УДК 004.05
С. В. Пономарев – магистрант кафедры компьютерной математики и программирования
В. В. Бураков (д-р техн. наук) – научный руководитель

СТАТИСТИЧЕСКИЙ АНАЛИЗ ОБЪЕКТНО-ОРИЕНТИРОВАННЫХ ПРОГРАММ

Оценка качества разрабатываемого программного продукта является одним из важнейших показателей, которые необходимо контролировать на всем жизненном цикле разработки. Игнорирование данного показателя или недооценка его важности часто приводит к срывам сроков выпуска, увеличением стоимости разработки.

Не существует единого подхода к оценке качества, однако все они в своей основе используют различные метрики и в зависимости от целей, оценивают качество путем определение таких показателей как тестируемость(покрытие тестами), сложность потока управления, зависимости между модулями программы и другие. Основными их недостатками является сильная ограниченность их применения специфическими требованиями. Комплексные оценки качества требует экспертной оценки со стороны достаточно квалифицированного персонала. Именно поэтому требуется автоматизированный инструмент оценивания качества разрабатываемых программных продуктов.

В процессе работы над проектами, было разработано множество подходов к разработке программного обеспечения, которые удовлетворяют те или иные потребности разработчиков. Однако, в связи с необходимостью наиболее быстрого выпуска проектов и постоянно изменяющимися требованиями, многие классические методологии оказались весьма неэффективными. Среди руководителей проектов популярность стали набирать гибкие методологиями разработки, которые основаны на использовании итеративной модели разработки и изначально были нацелены на постоянные изменения в требованиях к конечному продукту. Манифест гибкой разработки был принят множеством влиятельных разработчиков, представителей разных методологий, которые на тот момент уже нашли применение и подтвердили свою состоятельность на успешно завершенных проектах. На данный момент гибкие методологии разработки получили широкое распространение, как в малых, так и в крупных командах разработчиков. На основе принципов гибких методологий были выделены признаки плохой архитектуры приложений, наличие которых сводит на минимум большинство преимуществ гибких методологий [1]:

— жесткость (rigidity). Это характеристика программы, затрудняющая внесение в нее даже самых простых изменений. Зачастую возникает вследствие сильной связности, когда небольшое изменение в одном модуле, влечет за собой каскадные изменения во множестве зависимых модулях. Чем больше изменений придется внести в зависимые модули, тем более жесткой считается система. Данная характеристика показывает, насколько система поддается изменениям;

— хрупкость (fragility). Это свойство программы повреждаться во многих местах при внесении в нее единственного изменения. Как и прошлый признак, показывает, насколько система поддерживает изменения. Важным отличием является разрушение системы даже в модулях, не имеющих явной связи с текущим. Часто может возникать вследствие копирования и вставки исходного кода программы в разные места. При необходимости внести изменения в такой код в одном месте, может разрушиться логика в совершенно других модулях приложения. Такие ошибки часто возникают во время выполнения, что делает их трудно обнаружимыми и более опасными;

— косность/неподвижность (immobility). Характеризует сложность повторного использования модуля в других проектах. Возникает при необходимости использования некоторого модуля приложения вне текущего проекта, когда вместе с модулем требуется переносить множество его зависимостей. Использование модуля, обладающего высокой косностью, может потребовать значительных усилий на его интеграцию и развертывание в составе нового продукта, а иногда может и вовсе свести на нет возможность повторного его использования;

— вязкость (viscosityofdesign). Показывает насколько сложно вносить изменения в программу, учитывая текущие требования архитектуры. Если при добавлении новой функциональности в разрабатываемый проект соблюдать текущую архитектуру становится все труднее, это может свидетельствовать о повышении вязкости системы. Система становится наиболее вязка, когда вносить изменения с

учетом архитектуры становится более трудоемким, чем решать проблему обходным путем. Повышение уровня вязкости приводит к несоблюдению архитектуры и, как следствие, к ее ухудшению;

— ненужная сложность. Проявляется, когда в проекте используется не вся созданная инфраструктура, вследствие дальновидного планирования, когда разработчик пишет код, который не используется на данный момент, но возможно будет использован в будущем, тем самым повышает сложность архитектуры приложения, которая не приносит непосредственного преимущества в данный момент;

— ненужные повторения. Характеризует использование похожего кода, который различается небольшими изменениями, в различных частях программы. При нахождении ошибок или необходимости внести изменения в такие блоки кода, требует каскадного изменения во всех похожих местах, однако, так как блоки не являются идентичными, увеличивается поиск всех таких блоков, а так же их изменения. Дополнительной на практике опасностью служит низкая приоритетность при исправлении такого рода зависимостей, как правило, они игнорируются, вследствие работоспособности текущего кода. Зачастую ненужные повторения свидетельствует об отсутствии или неправильности построенных абстракций для таких блоков;

— непрозрачность. Трудность модуля для понимания. Код должен писаться для людей, а не машин. Так как код пишется один раз, а читается неоднократно, следует более тщательно подходить к понятности кода. Зачастую о высоком уровне непрозрачности может свидетельствовать большое количество комментариев в исходном коде. Так как современные языки программирования могут предложить достаточно высокий уровень языковых средств для выражения свои намерений, необходимость в комментировании текста программы зачастую свидетельствует о неявном потоке управления и т п. [2] Непонятность кода может быть связана с преждевременной оптимизацией. Высокая степень непрозрачности увеличивает время входа новых разработчиков в проект, повышает сложность внесения изменений, может снижать повторное использование.

Для избавления от признаков плохой архитектуры приложений было разработано множество принципов проектирования программ. Наиболее широкое признание и применение нашли объектно-ориентированные принципы проектирования классов SOLID,предложенные Робертом Мартином, однако их авторство принадлежит многим влиятельным разработчикам на основе их многолетнего опыта проектирования, управления проектами и разработке программных продуктов. Соблюдение данных принципов позволяет исключить признаки плохого дизайна и сделать разработку максимально эффективной при применении гибких методологий. [1] Для формализации оценки качества проектирования объектно-ориентированных программ на основе SOLID, необходимо к каждому из принципов, поставить в соответствие некоторую метрику или набор метрик, позволяющих измерить их численно. Далее приведено краткое описание принципов SOLID с подобранными метриками. [1]

Принцип единственной ответственности(single responsibility principle; SRP). На каждый объект должна быть возложена одна единственная обязанность. У класса должна быть только одна причина для изменения. Применимые метрики:

- недостаток связности в методах(lack of cohesion in methods; LCOM);
- центростремительное сцепление (afferent coupling; CA);
- центробежное сцепление (efferent coupling; CE);
- relational cohesion (H);
- нестабильность (instability; I).

Принцип открытости/закрытости (open/closed principle; OCP). Программные сущности должны быть открыты для расширения, но закрыты для изменения. Данный принцип можно охарактеризовать следующим правилом, для модификации программы необходимо писать новый код, а не изменять существующий. Применимые метрики:

- количество зависимостей класса от конкретных классов;
- количество зависимостей класса от абстракций.

Принцип замещения (подстановки) Лисков (Liskov substitution principle; LCP).Объекты в программе могут быть заменены их наследниками без изменения свойств программы. Данный принцип является важнейшим критерием для оценки качества принимаемых решений при построении иерархий

наследования. Его несоблюдение приводит к хрупкости в системе, так как при передаче объекта такого класса мы будем иметь ошибочное поведение. Расчет данного принцип можно формализовать, используя методологию проектирования по контракту. Применимые метрики: количество операций, переопределяемых подклассом (number of operations overridden by a subclass; NOO).

Принцип разделения интерфейса (interface segregation principle; ICP). Много специализированных интерфейсов лучше, чем один универсальный. Клиенты не должны зависеть от методов, которые они не используют. Несоблюдение принципа приводит к увеличению жесткости системы, так как при изменении интерфейса возникает необходимость изменить не только те классы, непосредственно для которых данное изменение было необходимо, но и так же изменения потребуется внести в вынужденно зависящие от производимого изменения классы. Применимые метрики:

- количество открытых методов в классе;
- недостаток связности в методах(lack of cohesion in methods; LCOM).

Принцип инверсии зависимостей (dependency inversion principle; DIP). Модули верхних уровней не должны зависеть от модулей нижних уровней. Оба типа модулей должны зависеть от абстракций. Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций. Позволяет исключить ненужные зависимости, тем самым снизить жесткость, хрупкость и косность системы. Применимые метрики:

- количество зависимостей класса от конкретных классов,
- количество зависимостей класса от абстракций.

Использование принципов SOLIDв определении качества разрабатываемых программных продуктов сочетает в себе опыт множества экспертов в области проектирования и разработки программного обеспечения и позволяет производить комплексную оценку состояния проекта на различных стадиях его разработки. Используя методы статистического анализа и множество данных полученных из эталонных программ, с открытым исходным кодом, можно определить допустимые интервалы изменения значений метрик для каждого из принципов, с целью проверки качества проектирования и предложением рекомендаций по улучшению кода, что позволяет построить автоматизированный инструмент оценивания качества разрабатываемых программных продуктов. Использование такого инструмента позволит обеспечить сокращение трудозатрат и сроков создания программного обеспечения, повысив его качество.

Библиографический список

1. *Мартин, Р., Мартин, М.* Принципы паттерны и методологии гибкой разработки на языке C# / СПб.: Символ-Плюс, 2011. 768 с.
2. *Мартин, Р.* Чистый код: создание, анализ и рефакторинг /СПб.: Питер, 2011. 464 с.