

Российская Академия наук
Санкт-Петербургский академический университет —
Научно-образовательный центр нанотехнологий
Санкт-Петербургская кафедра философии РАН

История и методология формальной верификации программ

Реферат аспиранта СПбАУ
Жаворонкова Эдгара Андреевича

Научный руководитель
к.ф.-м.н., доцент
Москвин Денис Николаевич

Санкт-Петербург
2018

Содержание

1	Введение	3
2	Понятие качества программного обеспечения	4
3	Виды методов обеспечения качества программного обеспечения	8
3.1	Ручные методы	8
3.2	Статические методы	10
3.3	Динамические методы	12
4	Формальная верификация программ	14
4.1	Историческая справка	14
4.2	Теории типов и λ -исчисление	15
4.3	Проверка моделей	15
5	Список литературы	16

1. Введение

Вычислительная техника и программное обеспечение¹ уже настолько прочно вошли в обиход человечества, что невозможно представить нашу жизнь без них. Программное обеспечение решает задачи автоматизации процессов в самых разных областях, начиная от простейших вычислений и заканчивая медициной или атомной энергетикой. В зависимости от задачи, возложенной на программное обеспечение, отличаются и предъявляемые к нему требования. В том числе, требования к надежности и, как следствие – качеству. Кроме того, нельзя забывать и о постоянно растущей сложности программного обеспечения, из-за которой зачастую так легко допустить критическую ошибку при разработке.

В этом эссе я попробую привести исторические моменты, так или иначе связанные с вопросом обеспечения качества программного обеспечения и показать их методологический аппарат. Мы увидим различные трактовки понятия качества программного обеспечения, увидим, какие существуют методы для обеспечения качества ПО и поближе познакомимся в один из них – формальной верификацией.

¹Я буду в равной степени употреблять термин «программное обеспечение» и соответствующую аббревиатуру – ПО

2. Понятие качества программного обеспечения

Существуют различные подходы к трактовке понятия качества программного обеспечения. Международный стандарт ISO 8402:94, например вводит качество программного обеспечения как «весь объем признаков и характеристик программ, который относится к их способности удовлетворять установленным или предполагаемым потребностям» – [6]. Стандарт IEEE Std 610.12-1990 определяет качество ПО как «степень, в которой система, компонент или процесс удовлетворяют потребностям или ожиданиям заказчика или пользователя» – [1]

Однако исторически, первой записью о качестве продукта можно считать определение Уолтера Шухарта (англ. Walter Andrew Shewhart) в книге [10] качества как сущности, состоящей из двух аспектов. Один из них имеет дело рассмотрением качества как объективного факта, не зависящего от существования человека. Другой же, напротив, имеет дело с чувственными ощущениями реальности. Тем самым, Шухарт подводит нас к мысли, что у качества есть субъективная сторона.

Далее мысль Шухарта развил Джеральд Вайнберг (англ. Gerald Marvin Weinberg) в своей работе 1992 года [11], говоря о качестве, как о «значимом для какого-либо человека». Отсюда полезно задаться вопросом, какие люди будут оценивать качество отдельно взятого программного продукта и что будет ценным для них?

Отголосок этой мысли есть и в международном стандарте ISO/IEC 9126 – [7], который вводит так называемую модель качества, которая описывает различные подходы к нему в течение всего жизненного цикла ПО.

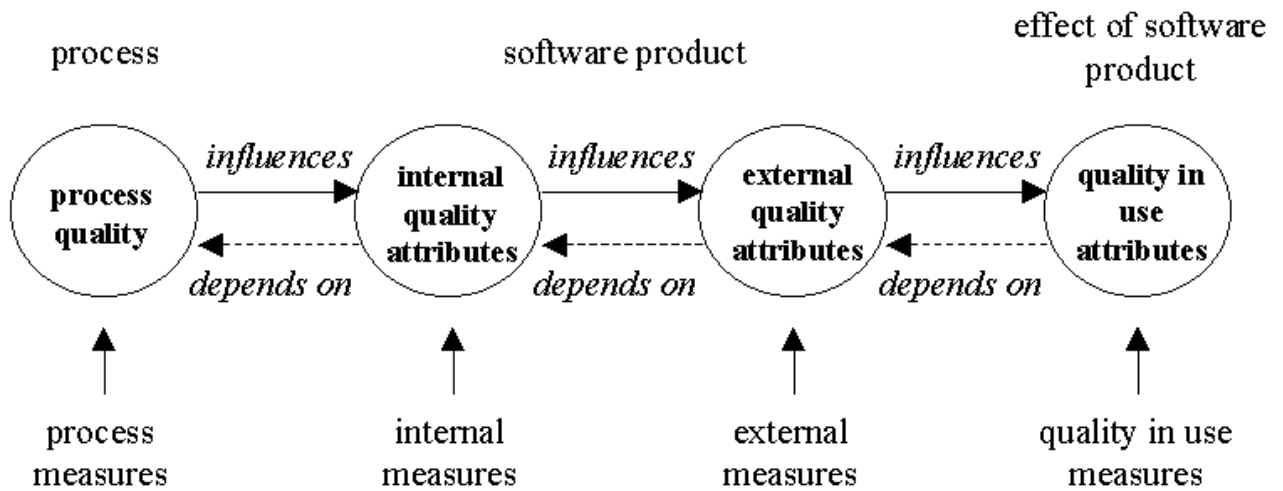


Рис. 1: Модель качества, согласно ISO/IEC FDIS 9126-1

Уже при первом взгляде на эту модель становится понятно, что на различных этапах жизненного цикла ПО используются различные, понятия качества. Так, в ходе непосредственно разработки мы говорим о качестве *процесса* разработки, которое напрямую влияет на внутреннее качество программного продукта.² Внутреннее качество продукта, в свою очередь, влияет на внешнее качество продукта, которое, затем, определяет качество продукта с точки зрения пользователя.

Может возникнуть вопрос, чем отличаются внутреннее качество продукта, от внешнего? Можно считать, что внутреннее качество программного обеспечения связано с некоторыми внутренними метриками(метриками, которые можно получить не запуская программу). Внешнее качество, соответственно, связано с внешними метриками(например, как ведет себя запущенная в некотором внешнем окружении программа).

В рамках внутреннего и внешнего качества стандарт предлагает следующие характеристики, приведенные на рисунке ниже:

²Кроме того, я буду использовать термин «программный продукт», считая его синонимом для программы или программного обеспечения. Формально, это конечно же, неверно, но в нашем контексте это не играет большой роли.

Software Quality Characteristics



Рис. 2: Характеристики качества ПО, согласно ISO/IEC FDIS 9126-1

1. Функциональность (*Functionality*) определяется способностью ПО решать возложенные на него задачи, соответствующие потребностям пользователя при заданных условиях использования. То есть эта характеристика говорит о том, что программа работает исправно точно и безопасно.
2. Надежность (*Reliability*) определяется способностью программы штатно завершаться, восстанавливаться в случае сбоев в работе. Другими словами – это способность программы выполнять требуемые задачи в обозначенных условиях, в течение обозначенных сроков.
3. Удобство использования (*Usability*) означает легкость в понимании и изучении ПО для пользователя. Эта характеристика больше относится к качеству ПО с точки зрения пользователя.

4. Эффективность(*Efficiency*) определяется способностью программы обеспечивать требуемый уровень производительности в условиях обозначенного времени или используемых ресурсов.
5. Удобство сопровождения(*Maintainability*) – это легкость, с которой программу можно анализировать, тестировать изменять с целью исправления каких-либо дефектов или для адаптации к новому окружению.
6. Переносимость(*Portability*) – это способность программного обеспечения запускаться и работать вне зависимости от программного или аппаратного окружения.

Нас будет интересовать в большей степени внутреннее качество ПО и способы его обеспечения, однако мы чуть-чуть коснемся и внешнего качества. Качество процесса разработки, равно как и качество ПО с точки зрения пользователя мы постараемся оставить за кадром.

3. Виды методов обеспечения качества программного обеспечения

В этой главе мы рассмотрим некоторые методы обеспечения качества ПО и классифицируем их. Однако перед этим полезно представить себе, что такое обеспечение качества программного обеспечения.

В предыдущей главе мы увидели некоторое количество трактовок определения качества ПО, с точки зрения стандартов, регламентирующих процесс разработки. Обеспечением качества, можно полагать процесс или результат формирования требуемых свойств продукта по мере его создания а также – поддержания этих свойств по мере внесения изменения в продукт.

Отметим, что это довольно универсальное определение, которое можно применить, как и к обеспечению качества программного обеспечения, так и к обеспечению качества любой другой продукции.

3.1. Ручные методы

Ручные методы обеспечения качества программного обеспечения представляют собой с одной стороны, самые простые с точки зрения применения, но, с другой стороны, самые ненадежные способы удостовериться в том, что ПО выполняет возложенные на него задачи. Под ручными методами мы будем понимать все методы, которые не предполагают или не поддаются какой-либо автоматизации.

К таким методам можно отнести, например, код-ревью или тестирование программного обеспечения.

Код-ревью или просмотр кода – это инженерная практика, заключающаяся в том, что код программы просматривается одним или несколькими разработчиками с целью обнаружения дефектов или же совершенствования навыков разработчика.

Исторически, первым упоминанием код-ревью можно считать работу Майкла Фагана(англ. Michael Fagan) [4], в которой он предлагает методологию нахождения дефектов в программном коде путем его инспекций.

По Фагану, процесс инспекции представляет собой сравнение выхода каждой операции с некоторым выходным критерием. Кроме выходных

критериев Фаган вводит входные критерии – критерии, которым должна удовлетворять операция перед ее началом.

Более подробно, процесс инспекции состоит из следующих операций:

1. Планирование(*Planning*). Включает в себя подготовку материалов(кода), места для встречи и встречи участников.
2. Обзор(*Overview*). Включает ознакомление участников процесса с материалами и назначение ролей.
3. Подготовка(*Preparation*). Включает непосредственный обзор участниками материалов с целью выяснения возможных дефектов.
4. Обсуждение(*Inspection meeting*). Включает непосредственное нахождение дефектов.
5. Исправления(*Rework*). Найденные дефекты исправляются их авторами
6. Заключение(*Follow-up*). Финальный этап, на котором участники удостоверяются, что все найденные дефекты исправлены и процесс исправлений не породил новых.

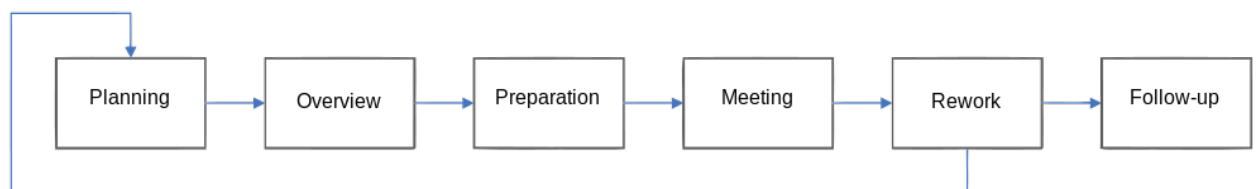


Рис. 3: Процесс инспекции кода по Фагану

Помимо инспекции кода существует еще и большое количество видов тестирования программного обеспечения. Как-то:

1. Функциональное тестирование
2. Системное тестирование
3. Тестирование производительности

4. Регрессионное тестирование
5. Модульное тестирование
6. и т.д.

Нас больше интересует так называемое ручное тестирование в ходе которого инженер использует продукт, моделируя действия конечного пользователя. Предыдущие виды тестирования так или иначе используют программные средства для запуска или анализа кода, поэтому его можно отнести скорее к динамическим методам.

Основная идея ручного тестирования заключается в том, что бы посмотреть на систему с точки зрения пользователя и увидеть недостатки в качестве ПО, которые не сразу заметны в ходе разработки. Этот вид тестирования довольно неплохо работает, например, для пользовательских интерфейсов.

Более или менее очевидно, что ручные методы плохо справляются со сложными системами, либо же требуют слишком много времени и усилий. Поэтому далее мы рассмотрим две группы методов, которые используют программные средства и являются более выразительными в том смысле, что позволяют проверить более строгие инварианты в программном коде.

3.2. Статические методы

Под статическими методами понимают все методы, которые используют различные артефакты, полученные при проектировании или разработке. Такими артефактами, например, являются требования, спецификации или сам программный код. Важная особенность статических методов заключается в том, что они никоим образом не используют результат выполнения программы. К таким методам относят например, статический анализ кода или формальную верификацию.

Статический анализ кода это процесс выявления ошибок и недочетов в исходном коде, который можно рассматривать как автоматизированный вариант код-ревью. Помимо собственно выявления ошибок, статический анализ позволяет решать задачи рекомендаций по оформлению кода и подсчет метрик кода(например цикломатической сложности или сцепления между двумя программными модулями).

Статические анализаторы обладают тем преимуществом, что позволяют находить огромное количество ошибок еще на этапе программирования, что существенно снижает стоимость их исправления. Кроме того статический анализатор обеспечивает:

1. Полное покрытие кода. Участки кода, которые редко получают управление обычно остаются без внимания в ходе код-ревью, между тем являясь потенциальным источником ошибок.
2. Так как статический анализ не использует результат запуска программы, то он не зависит от окружения, в котором она может исполняться, тем самым позволяя находить ошибки, которые проявляются при смене окружения(компилятора, аппаратной платформы и т. д.)
3. Не секрет, что человек часто опечатывается по невнимательности и такие ошибки очень сложно обнаружить. Статические анализаторы с легкостью находят ошибки такого рода и позволяют сэкономить огромное количество времени.

Несмотря на кажущуюся современность, история статического анализа кода начинается примерно в 1970-ых годах, с появления утилиты `lint` в операционной системе Unix, которая описана в работе Стивена Джонсона(англ. Stephen C. Johnson) [8]. Именно ее можно считать первым статическим анализатором. С современной точки зрения она была довольно примитивной, но именно она дала начало развитию статических анализаторов.

Формальные методы, в отличии от статического анализа кода, опираются на математический аппарат в ожидании, что его использование существенно повышает надежность разрабатываемой системы. При этом, они довольно сложны и зачастую основываются на не всегда достижимых в реальности предположениях. Формальные методы можно применять на трех уровнях:

1. Нулевой. Разрабатывают формальную спецификацию, как артефакт, с оглядкой на который в дальнейшем идет разработка. Очевидно, в этом случае мы все еще не можем гарантировать, что написанный

код соответствует спецификации, для чего нужен следующий уровень.

2. Первый. Программный код **выводится** из формальной спецификации автоматически, что позволяет неформально рассуждать о том, насколько он соответствует спецификации.
3. Второй. Полностью формализованные доказательства выводятся и проверяются автоматически.

Формальная верификация занимает первый уровень применения формальных методов. Некоторые конкретные механизмы мы рассмотрим в следующей главе, а здесь же просто скажем, что она собой представляет. Теоретически, основой формальной верификации служит доказательство на некой абстрактной математической модели свойств программной системы в предположении о том, что эта модель адекватно отражает природу вышеозначенной системы. Зачастую для моделирования используют:

1. Формальную семантику языка программирования
2. Теории и типов
3. Абстрактные автоматы
4. Вычислительные формализмы, такие как машина Тьюринга или Поста или λ -исчисление
5. и т.д.

3.3. Динамические методы

К динамическим методам мы отнесем все методы, которые используют результат запуска программы. Основная цель таких методов – измерить производительность программы в заданных условиях или обнаружить ошибки. Примерами таких методов являются профилирование и тестирование программного обеспечения.

Профилирование представляет собой сбор и измерение характеристик работы программы с целью последующей ее оптимизации. Характеристиками могут выступать время работы программы, используемые ресурсы компьютера, частота вызова определенных функций и так далее.

Инструмент, который осуществляет профилирование называется профилировщиком(*profiler*).

Первым профилировщиком можно считать утилиту *prof* из операционной системы Unix, которая занималась тем, что собирала данные о том, сколько времени и с какой частотой происходили вызовы функций в программах, написанных под эту операционную систему. Дальнейшие профилировщики, такие как *gprof* развивали эту технику, за счет построения и анализа графа вызовов(*call graph*) функций в программе, что позволяло оценивать, насколько часто та или иная функция вызывается относительно других и какое место она занимает в цепочке вызовов. – [5].

Современные профилировщики позволяют производить так называемое инструментирование – модификацию кода программы, с целью сбора данных для анализа. Кроме профилирования, инструментирование полезно в контексте анализа кода в средах разработки.

Тестирование программного обеспечения же, предполагает запуск программы с целью нахождения ошибок. Важным моментом является тот факт, что тестирование именно **находит** ошибки, в отличие от формальных методов, вкратце описанных ранее, которые **доказывают** отсутствие ошибок по отношению к спецификации программы. На этот счет широко известна цитата Эдсгера Дейкстры(нидерл. Edsger Wybe Dijkstra) – «тестирование показывает не отсутствие ошибок, но их наличие» из [2].

Мы уже приводили некоторые виды тестирования программного обеспечения в первом разделе этой главы, поэтому не будем повторяться, скажем лишь, что несмотря на кажущуюся ненадежность, тестирование все еще остается одним из самых популярных методов обеспечения качества программного обеспечения, который позволяет убедиться в том, что программа на базовом уровне отвечает требованиям, предъявленным к ней в спецификации.

Мнообразие видов тестирования позволяет получать большое количество информации для анализа, а возможность автоматизации позволяет удобно встроить тестирование в жизненный цикл разработки. Например – автоматический запуск тестов при пересборке программы, что позволяет быстро находить ошибки в программе еще на этапе разработки.

4. Формальная верификация программ

В этой главе мы несколько подробнее обсудим некоторые методы формальной верификации программ. Мы увидим, на что они опираются, в каких предположениях действуют и какова их выразительная сила. Кроме того, мы обсудим, где лежат корни идеи формальной верификации программ.

4.1. Историческая справка

Согласно [9], еще в пятидесятых годах прошлого века Мартином Девисом(англ. Martin Davis) было представлено первое формальное доказательство, полученное автоматически – доказательство того, что произведение двух четных чисел четно в арифметике Пресбургера. Спустя короткое время, уже в конце шестидесятых, начали появляться первые автоматические доказательства теорем, которые использовались для верификации программ, написанных на таких языках, как Pascal, Ada и Java.

Еще позже, в 1972 году, Робинот Милнером(англ. Robin Milner) был создана система проверки доказательств LCF(Logic for Computable Functions), которая положила начало современным системам автоматического доказательства теорем, таким как HOL или Coq. На текущий момент, Coq является одним из самых популярных инструментов формальной верификации и автоматического доказательства теорем.

У доказателей теорем есть один недостаток, который заключается в том, что если некоторое утверждение **не** является теоремой в некоторой логике, то доказатель ничего не сможет вам об этом сказать.

Для верификации конкурентных(многопоточных) программ широко используются другие методы, такие как проверка моделей или Сети Петри. Их история начинается в восьмидесятых годах с работ Аллена Эмерсона и Эдмунда Кларка(англ. E. Allen Emerson и Edmund M. Clarke) о применении в этой области темпоральной логики, например – [3]

Если оторваться от контекста индустрии разработки ПО, то принцип верификации был выдвинут еще в тридцатых годах. Студенты и преподаватели кафедры философии индуктивных наук Венского университета собирали семинар, более известный как Венский Кружок. Именно там под влиянием идей еще Эрнста Маха(нем. Ernst Mach), выдвинувшего мысль

о том, что «суждения, которые не могут быть ни проверены, ни отвергнуты не имеют отношения к науке» – цитата по [12] был сформулирован принцип верификационизма, утверждавший о том, что к реальному миру имеют отношения лишь те суждения, которые можно проверить экспериментом.

4.2. Теории типов и λ -исчисление

4.3. Проверка моделей

5. Список литературы

- [1] *IEEE Std 610.12-1990: IEEE Standard Glossary of Software Engineering Terminology*. IEEE, 1990.
- [2] John N Buxton and Brian Randell. *Software Engineering Techniques: Report on a Conference Sponsored by the NATO Science Committee*. NATO Science Committee; available from Scientific Affairs Division, NATO, 1970.
- [3] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, UK, 1982. Springer-Verlag.
- [4] Michael E Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 38(2/3):258, 1999.
- [5] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof: A call graph execution profiler. *SIGPLAN Not.*, 17(6):120–126, June 1982.
- [6] ISO. *ISO 8402:1994 Quality management and quality assurance – Vocabulary*. ISO, 1994.
- [7] ISO/IEC. *ISO/IEC 9126. Software engineering – Product quality*. ISO/IEC, 2001.
- [8] S. C. Johnson. Lint, a c program checker. In *COMP. SCI. TECH. REP*, pages 78–1273, 1978.
- [9] Eugenio G Omodeo and Alberto Policriti. *Martin Davis on Computability, Computational Logic, and Mathematical Foundations*, volume 10. Springer, 2017.
- [10] Walter Andrew Shewhart. *Economic control of quality of manufactured product*. ASQ Quality Press, 1931.
- [11] Gerald M Weinberg. *Quality software management*. New York, 1993.
- [12] Википедия. Венский кружок — Википедия, свободная энциклопедия, 2017. Дата обращения: 11 апреля 2018.