Математические структуры и моделирование
2025. № 1 (73). С. 125–135

УДК 004.42
DOI 10.24147/2222-8772.2025.1.125-135

SOLID ПРОТИВ GRASP

Д.Н. Лавров[1]
к.т.н., доцент, e-mail: dmitry.lavrov72@gmail.com
Д.Д. Лаврова[2]
студент, e-mail: drlvrv@gmail.com

[1]Нижневартовский государственный университет, Нижневартовск, Россия
[2]Омский государственный университет им. Ф.М. Достоевского, Омск, Россия

Аннотация. В статье рассматривается использование принципов SOLID и шаблонов GRASP на этапе проектирования. Что лучше использовать для разработки? Даёт ли их использование одинаковую архитектуру? Что проще использовать с методической точки зрения для обучения студентов? Авторы дают своё видение, отвечая на данные вопросы.

Ключевые слова: SOLID, GRASP, шаблоны проектирования, разработка, обучение, архитектура.

Введение

При проектировании программного обеспечения на основе объектно-ориентированного подхода с целью получения масштабируемой архитектуры на практике используют принципы SOLID и/или шаблоны проектирования GRASP. Часто можно видеть, что SOLID и GRASP используются совместно. Но можем ли мы обойтись только одним? Достаточно ли нам принципов SOLID? А может быть, GRASP полнее отвечает на вопросы, которые возникают при проектировании архитектуры программного обеспечения? А может, SOLID – это и есть GRASP, сформулированный другими словами (и наоборот)? А если их используют совместно, можно ли сформулировать простой и понятный набор новых принципов, объединяющий два подхода? Интересен также методический аспект: при преподавании дисциплин программной инженерии с чего лучше начать с SOLID или с GRASP? Или можно дать лишь один из подходов? В данной статье представлен авторский анализ и выводы по некоторым из вышеперечисленных вопросов.

Прежде чем приступить, определимся с ключевыми понятиями SOLID и GRASP.

1. Краткая историческая справка и основные понятия

Впервые термин «шаблон проектирования» в отношении объектно-ориентированных анализа и проектирования появился в знаменитой книге 1994 г. «Банда четырёх» (Gang of Four – GoF) [1]. В последующем за шаблонами

этого каталога закрепилось название – шаблоны GoF. Простыми словами, шаблон проектирования – это пара «проблема–решение».

Шаблоны **GRASP (General Responsibility Assignment Software Patterns)** появились в 1997 г. Крэг Ларман систематизировал и собрал первую версию каталога шаблонов GRASP. Фактически это было обобщением принципов распределения обязанностей. В отличие от шаблонов GoF, часто дававших конкретное архитектурное решение, шаблоны GRASP больше были похожи на принципы. Проблема формулировалась в максимально общем виде. Формулировка решения следовала такому же принципу. Формулировки и сам каталог шаблонов эволюционировали, некоторые шаблоны были переосмыслены и заменены. С последней официальной версией 2004 г. можно ознакомиться в [2].

Каталог GRASP в настоящее время состоит из 9 шаблонов. Дадим их краткие формулировки:

1. **Information Expert (IE)** – Информационный эксперт:
   Ответственность должна быть назначена классу, который обладает всей необходимой информацией для выполнения задачи.
   Пример: Класс, хранящий данные о заказе, должен отвечать за расчёт его стоимости.
   Замечания: редко используется в одиночку, как правило, в комбинации с HC. Иначе мы получаем «Божественный класс».
2. **Creator (Cr)** – Создатель:
   Класс, который создаёт объекты другого класса, должен быть ответственным за их создание.
   Пример: Класс Order может создавать объекты класса OrderItem.
   Замечания: Также на практике используется с P и PV, что приводит к масштабируемым решениям.
3. **Controller (Ctrl)** – Контроллер:
   Ответственность за обработку системных событий должна быть назначена классу, представляющему систему или сценарий использования.
   Пример: Класс OrderController обрабатывает запросы, связанные с заказами.
4. **Low Coupling (LC)** – Низкая связанность:
   Классы должны быть слабо связаны друг с другом, чтобы изменения в одном классе минимально влияли на другие.
   Пример: Использование интерфейсов для уменьшения зависимости между классами.
5. **High Cohesion (HC)** – Высокое зацепление:
   Классы должны иметь чётко определённые и тесно связанные обязанности, быть сфокусированы на одной задаче.
   Пример: Класс, отвечающий только за логику аутентификации, а не за отправку email.
6. **Polymorphism (P)** – Полиморфизм:
   Использование полиморфизма для обработки альтернативных вариантов поведения на основе типа объекта.

Пример. Разные классы PaymentMethod (CreditCard, PayPal) реализуют метод pay().

7. **Pure Fabrication (PF)** – Чистая выдумка:
   Создание искусственных классов для выполнения задач, которые не соответствуют ни одной из существующих сущностей.
   Пример: Класс Logger, который отвечает только за логирование.
8. **Indirection (I)** – Косвенность:
   Использование промежуточного объекта для уменьшения связанности между компонентами.
   Пример: Внедрение зависимостей через интерфейсы.
9. **Protected Variations (PV)** – Защита от изменений:
   Изоляция компонентов от изменений в других частях системы.
   Пример: Использование абстракций для защиты от изменений в реализации.

SOLID – это набор из пяти принципов объектно-ориентированного программирования и проектирования, которые помогают создавать гибкие, поддерживаемые и масштабируемые системы. Предложил основные формулировки и популяризировал принципы Роберт Мартин в своей работе 2003 г. [3]. Аббревиатура SOLID расшифровывается по первым буквам принципов, в неё входящих.

SOLID принципы так же эволюционировали, как и GRASP. Современные формулировки принципов SOLID сохраняют свою суть, но адаптированы к текущим реалиям разработки, включая микросервисы, функциональное программирование и компонентно-ориентированные подходы. Вот как они выглядят сегодня:

1. **Single Responsibility Principle (SRP)** – Принцип единственной ответственности.
   Классическая формулировка: Класс должен иметь только одну причину для изменения.
   Современная формулировка: Модуль (класс, функция, компонент) должен быть ответственен только за одну задачу или функциональность.
   Акцент: высокое зацепление (cohesion) внутри модуля; разделение задач для упрощения тестирования и поддержки.
   Пример: В микросервисной архитектуре каждый сервис отвечает за одну бизнес-задачу (например, аутентификация, оплата).
2. **Open/Closed Principle (OCP)** – Принцип открытости/закрытости.
   Классическая формулировка: Программные сущности должны быть открыты для расширения, но закрыты для модификации.
   Современная формулировка: Модули должны быть спроектированы так, чтобы их можно было расширять новым поведением без изменения существующего кода.
   Акцент: Использование абстракций (интерфейсов, абстрактных классов); поддержка плагинов и расширений.
   Пример: Добавление новой функциональности через наследование или внедрение зависимостей (Dependency Injection).
3. **Liskov Substitution Principle (LSP)** – Принцип подстановки Барбары Лисков.

Классическая формулировка: Объекты в программе должны быть заменяемы экземплярами их подтипов без изменения корректности программы.
Современная формулировка: Наследующие классы или компоненты должны дополнять, а не нарушать поведение базовых классов.
Акцент: контракты интерфейсов должны соблюдаться; наследование должно быть логичным и предсказуемым.
Пример: Если базовый класс Bird имеет метод fly(), то класс Penguin не должен наследовать его, так как пингвины не летают.

4. **Interface Segregation Principle (ISP)** – Принцип разделения интерфейса.
Классическая формулировка: Клиенты не должны зависеть от интерфейсов, которые они не используют.
Современная формулировка: Интерфейсы должны быть узкоспециализированными и соответствовать конкретным потребностям клиентов.
Акцент: избегание «жирных» интерфейсов.
Пример: Вместо одного интерфейса Printer с методами print(), scan(), fax() создать отдельные интерфейсы Printable, Scannable, Faxable.
5. **Dependency Inversion Principle (DIP)** – Принцип инверсии зависимостей.
Классическая формулировка: Модули верхнего уровня не должны зависеть от модулей нижнего уровня. Оба типа модулей должны зависеть от абстракций.
Современная формулировка: Зависимости должны строиться на основе абстракций (интерфейсов или абстрактных классов), а не конкретных реализаций.
Акцент: внедрение зависимостей (Dependency Injection); уменьшение связанности между компонентами.
Пример: Использование DI-контейнеров для управления зависимостями в микросервисах.

SOLID принципы применимы как в объектно-ориентированном программировании, так и в современных парадигмах, таких как микросервисы и компонентно-ориентированная разработка.

2. Методика сравнения

Сложность сравнения принципов SOLID и шаблонов GRASP заключается не только в том, что количество SOLID принципов не совпадает с количеством шаблонов GRAPS (пять против девяти), но ещё и в том, что для получения качественного «чистого» кода, как в GRASP, так и в SOLID недостаточно использовать только один принцип. Применение только одного принципа может привести к абсурдным ситуациям, которые называются антипаттернами (антишаблонами) проектирования и описаны в каталоге [4]. Например очевидно, что шаблон Information Expert нельзя использовать без шаблона High Cohesion, в противном случае можно получить God Object («<Божественный объект») [4] или перегруженный контроллер [2].

Ещё одной проблемой является направленность GRASP и SOLID. GRASP предназначен для ответов на вопросы о распределении ответственности, в то время как SOLID – на особенности архитектуры.

Основой сравнения будут два вопроса:

1. Какие принципы SOLID используются при реализации шаблонов GRASP?
2. Какие шаблоны GRASP используются при реализации принципов SOLID?

3. Какие принципы SOLID используются при реализации шаблонов GRASP?

Рассмотрим все шаблоны GRASP и покажем, через какие принципы SOLID получается их корректная реализация.

1. Information Expert (IE) – Крэг Ларман указывает [2]: применение основного правила назначения ответственности должно комбинироваться с HC и LC для достижения повторной используемости кода. Тогда в SOLID получаем наилучшее приближение к данной цели с использованием принципов SRP, ISP.

   Итог в виде символической записи:

   $$IE \approx SRP \wedge ISP \wedge \boxed{\text{кто хранит данные для выполнения задачи?}}$$

   Эта формула отражает, что:
   - SRP гарантирует, что класс фокусируется на данных и их обработке;
   - ISP разделяет интерфейсы для работы с данными, избегая избыточных зависимостей;
   - для смысловой полноты задаётся дополнительный вопрос: «Кто хранит данные для выполнения задачи?»
2. Creator (Cr). Основной принцип Creator – поручить создание объекта тому, кто его агрегирует, содержит, является контейнером для него, регистрирует или использует, имеет данные для инициализации. Агрегирование в первом приоритете.

   Принципы SOLID, которые при этом реализуются:

   $$Cr \approx SRP \wedge ISP \wedge DIP \wedge \boxed{\text{Кто агрегирует?}}$$

   Creator лишь в самых простых случаях используется в одиночку, обычно он работает с IE, HC, LC, P, PV. А в SOLID это перечисленные выше методы: SRP, ISP, DIP.

   Принцип Creator реализует SRP (класс создаёт то, чем управляет), ISP (узкие интерфейсы для фабрик) и DIP (зависимость от абстракций). Формула $Cr \approx SRP \wedge ISP \wedge DIP$ отражает, что агрегирующий класс:
   - имеет единственную ответственность – SRP;
   - использует минимальные интерфейсы для создания объектов – ISP;
   - работает с абстракциями, а не конкретными классами – DIP.
3. Controller (Ctrl) отвечает на вопрос: кто должен обрабатывать внешние так называемые системные события? Одно из часто используемых решений – это контроллер прецедента. Прецедент – это вариант использования (use case), в

некотором смысле обобщённый сценарий работы с разрабатываемой системой. Чтобы не получить «перегруженный» контроллер, Крэг Ларман предлагает использовать Pure Fabrication, разделяя на несколько контроллеров и реализуя принцип ISP.

$$Ctrl \approx SRP \wedge ISP \wedge \boxed{\text{К какому прецеденту относится контроллер?}}$$

4. Low Coupling (LC) требует минимального количества связей, чтобы обеспечить повторную используемость кода и ограничить изменения в коде всего проекта, вызванные локальными правками.

   Покажем верность символической записи:

$$LC \approx LSP \wedge ISP \wedge DIP \wedge SRP$$

   LSP (Принцип подстановки Лисков) – Подтипы должны заменять базовые типы без нарушения корректности программы. Это позволяет клиентам зависеть от абстракций (например, интерфейсов), а не от конкретных классов. Уменьшает сцепление между клиентом и реализацией, так как замена подтипов не требует изменений в коде, использующем базовый тип.

   Пример: Если класс Dog наследует Animal, то любой код, работающий с Animal, должен корректно работать и с Dog. Это снижает зависимость клиента от конкретных реализаций.

   ISP (Принцип разделения интерфейсов) – Клиенты не должны зависеть от интерфейсов, которые они не используют. Дробление больших интерфейсов на мелкие и специализированные уменьшает количество ненужных зависимостей между классами. Клиенты зависят только от тех методов, которые им действительно нужны, что снижает сцепление.

   Пример: Вместо интерфейса Worker с методами work(), eat(), sleep() создайте отдельные интерфейсы: Workable, Eatable, Sleepable. Класс Robot будет реализовывать только Workable, избегая зависимостей от нерелевантных методов.

   DIP (Принцип инверсии зависимостей) – Модули высокого уровня не должны зависеть от модулей низкого уровня. Оба должны зависеть от абстракций. Зависимости направляются на абстракции (интерфейсы или абстрактные классы), а не на конкретные реализации. Это делает систему гибкой: замена реализации не влияет на клиентский код и снижает связность.

   Пример: Класс PaymentService зависит от абстракции PaymentGateway, а не от конкретных классов PayPalGateway или SberGateway. Это снижает сцепление между сервисом и платёжными системами.

   SRP (Принцип единственной ответственности) – Класс должен иметь только одну причину для изменения. Классы с одной ответственностью имеют меньше зависимостей, так как их функциональность узконаправлена. Изменения в таком классе не затрагивают другие части системы.

   Пример: Класс UserValidator отвечает только за проверку данных пользователя, а UserDatabase — за сохранение в базу. Это разделение уменьшает сцепление между валидацией и работой с базой данных.

Подытожим. Почему именно эта комбинация? LSP, ISP и DIP работают на уровне управления зависимостями:

- LSP обеспечивает безопасную замену реализаций через полиморфизм.
- ISP дробит интерфейсы, минимизируя ненужные связи.
- DIP инвертирует зависимости, делая систему модульной.
- SRP дополняет их на уровне структуры классов: локализует ответственность, предотвращая «распухание» зависимостей.

Это создаёт систему, где компоненты слабо связаны, но сильно сфокусированы на своих задачах, что повышает её гибкость, расширяемость и устойчивость к изменениям.

5. High Cohession (HC)

$HC \approx SRP \wedge$ Логическое/Функциональное/Коммуникационное зацепление

SRP задаёт базовое правило: одна ответственность = фокус на одной задаче.

Более широкая трактовка зацепления требует, чтобы все элементы класса (методы, данные) были связаны общей логикой, а не просто объединены в одном месте.

HC наиболее близок к SRP, но имеет свои особенности.

Анализ дополнительных шаблонов GRASP осуществлялся аналогично.

6. Polymorphism (P) – Полиморфизм реализуется через SOLID:
   - LSP: объекты подтипов должны заменять базовые типы без нарушения логики программы;
   - OCP: система открыта для расширения (через новые подтипы), но закрыта для модификации;
   - DIP: система открыта для расширения (через новые подтипы), но закрыта для модификации:

$$P \approx LSP \wedge OCP \wedge DIP$$

7. Pure Fabrication (PF) реализуется через
   - SRP: создаётся искусственный класс, который берёт на себя одну конкретную задачу (например, логирование или кэширование);
   - ISP: интерфейсы для таких классов делаются узкоспециализированными:

$$PF \approx SRP \wedge ISP$$

8. Indirection (I) реализуется через DIP: Внедряется абстракция (интерфейс) между компонентами, чтобы уменьшить прямую зависимость:

$$I \approx DIP$$

9. Protected Variation (PV) реализуется через:

– OCP защищает части системы от изменений, а вариации вынесены в подтипы или плагины;
– DIP (зависимость от абстракций) позволяет подменять реализации без изменения клиентского кода:

$$PV \approx OCP \wedge DIP$$

4. SOLID реализуем применением GRASP

1. **Принцип единственной ответственности (SRP)** выполняется применением комбинации шаблонов Information Expert и High Cohesion.
   - **Information Expert** назначает ответственность классу, который владеет данными для выполнения задачи.
   - **High Cohesion** гарантирует, что класс фокусируется на одной задаче (высокое зацепление, сфокусированность).

$$SRP \approx IE \wedge HC$$

2. **Принцип открытости/закрытости (OCP)** реализуется через Protected Variation и Polymorphism.
   - **Protected Variation** защищает систему от изменений через абстракции.
   - **Polymorphism** позволяет расширять поведение через подтипы.

$$OCP \approx P \wedge PV$$

3. **Принцип подстановки Лисков (LSP)** реализуется через Polymorphism.
   - **Polymorphism** обеспечивает заменяемость подтипов базовым типом без нарушения логики.

$$LSP \approx P$$

4. **Принцип разделения интерфейсов (ISP)** – через Low Coupling и Pure Fabrication.
   - **Low Coupling**: разделение интерфейсов уменьшает зависимости.
   - **Pure Fabrication**: создание узких интерфейсов для конкретных клиентов.

$$ISP \approx LC \wedge PF$$

5. **Принцип инверсии зависимостей (DIP)** реализуется через Indirection и Protected Variation.
   - **Indirection** вводит абстракцию (например, интерфейс) между компонентами.
   - **Protected Variation** защищает от изменений через зависимость от абстракций.

$$DIP \approx I \wedge PV$$

5. Анализ результатов

Если объединить результаты, получим следующую таблицу.

Таблица 1. Результаты сопоставления GRASP и SOLID

	IE	Cr	Ctrl	LC	HC	P	PF	I	PV
SRP	++	−+	−+	−+	++	−−	−+	−−	−−
OCP	−−	−−	−−	−−	−−	++	−−	−−	++
LSP	−−	−−	−−	−+	−−	++	−−	−−	−−
ISP	−+	−+	−+	++	−−	−−	++	−−	−−
DIP	−−	−+	−−	−+	−−	−+	−−	++	++

Первый плюс в ячейке таблицы – применение шаблона GRASP для соблюдения принципа SOLID. Второй плюс – это выполнение принципа SOLID при использовании GRASP. Теоретически связь должна быть двунаправленной, но из-за наличия субъективности в наших оценках и рассуждениях возможны «нечёткие» суждения. « ++ » означает, что связь проявилась в обе стороны.

Из таблицы видно, что Creator и Controller не имеют двойной связи с SOLID, а следовательно, GRASP явно более широко описывает принципы проектирования, он более гибок. Как ни странно, но при применении на начальных этапах обучения объектно-ориентированного проектирования это скорее вредит. А значит, обучение следует начинать с SOLID принципов. SOLID принципы при внимательном рассмотрении более конкретны, они кратко формулируются и легки в запоминании. Рекомендуется пользоваться всегда современными трактовками.

Ещё одним вариантом в обучении является использование вначале только первых трёх принципов распределения обязанностей: IE, Cr, Ctrl – совместно с пятью принципами SOLID: SRP, OCP, LSP, ISP, DIP. Это также даст отличный результат.

В некотором смысле SOLID-принципы эквивалентны набору шаблонов GRASP: HC, P, PF, I, PV (см. табл. 1). А этот набор как раз и влияет на архитектуру разрабатываемого проекта на основе GRASP.

В то же время если мы хотим ограничиться изучением только одного из наборов, то следует выбирать GRASP. Кроме того, GRASP был использован К. Ларманом в прецедентной разработке OpenUP и привязан к одному из его этапов, и, следовательно, разумно применять GRASP в подобного вида прецедентных подходах.

Заключение

SOLID при использовании современных формулировок и трактовок проще как в изучении, так и в использовании. SOLID говорит, что делать, но не как. В свою очередь GRASP лучше объясняет конкретные действия по достижению целей проектирования (в аспекте распределения обязанностей), но для получения качественного кода нужно использовать комбинации шаблонов GRASP, которые нужно уметь подбирать для каждой конкретной ситуации. Это даёт высокую вариативность и

усложняет применение GRASP на начальном этапе обучения. Некоторые из этих комбинаций достаточно хорошо приближаются к конкретным принципам SOLID, а точнее, к решениям, полученным на основе этих принципов.

Получается, что SOLID и GRASP дают нам возможность проектировать программное обеспечение, находясь как бы в различных системах координат, в тех, которые больше подходят для конкретного разработчика и задачи, над которой он работает.

В этом смысле SOLID и GRASP при правильном применении будут давать одинаково «чистый» код, хорошо расширяемый и стабильный к изменениям.

При сопоставлении SOLID и GRASP выяснилось, что их совместное использование при распределении обязанностей (применяем GRASP-шаблон с сохранением принципов SOLID) даёт и хорошие архитектурные решения, без задействования других шаблонов GRASP, без комбинирования. Три первых шаблона GRASP в совокупности с пятью SOLID позволяют получить те же архитектурные решения при распределении обязанностей, что и полный комплект GRASP, но являются более понятными в применении.

В заключение отметим, что SOLID – это принципы, а не догмы. Не всегда нужно слепо следовать им. Иногда упрощение кода (даже с небольшим нарушением SOLID) может быть оправдано.

GRASP – это шаблоны, а не правила. Они предоставляют рекомендации, но не всегда являются единственным правильным решением. Лучшее решение – это компромисс между различными факторами, такими как простота, гибкость, поддерживаемость и стоимость. Всегда оценивайте последствия своих решений и выбирайте вариант, который лучше всего соответствует потребностям вашего проекта.

Литература

1. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приёмы объектно-ориентированного проектирования. Паттерны проектирования: пер. с англ. СПб.: Питер, 2001. 368 с.
2. Larman C. Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd ed.). Prentice Hall, 2004. 736 p.
3. Martin R.C. Agile Software Development, Principles, Patterns, and Practices. Prentice Hall, 2003. 552 p.
4. Brown W.J., Malveau R.C., McCormick H.W. III, Mowbray T.J. AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis. John Wiley & Sons, 1998. 384 p.

SOLID VS GRASP

D.N. Lavrov[1]
Ph.D. (Techn.), Associate Professor, e-mail: dmitry.lavrov72@gmail.com
D.D. Lavrova[2]
Student, e-mail: drlvrv@gmail.com

[1]Nizhnevartovsk State University, Nizhnevartovsk, Russia
[2]Dostoevsky Omsk State University, Omsk, Russia

Abstract. The article discusses the use of SOLID principles and GRASP patterns at the design stage. Which is better to use for development? Does their use provide the same architecture? Which is easier to use from a methodological point of view for teaching students? In the article, the authors give their vision, answering these questions.

Keywords: SOLID, GRASP, design pattern, development, training, architecture.

Дата поступления в редакцию: 15.02.2025