Белорусский государственный университет

Факультет прикладной математики и информатики

Дехтяренко Владислав Ярославович

**РЕФЕРАТ**

**на тему «Принципы объектно-ориентированного программирования и проектирования SOLID»**

студента 1 курса 2 группы

**Преподаватель**

*Баранов Геннадий Аркадьевич*

Минск, 2017

**Содержание**

Введение

Принципы объектно-ориентированного проектирования SOLID:

1. Принцип единственности ответственности
   1. Валидация данных
   2. «Объект Бога»
   3. Резюме
2. Принцип открытости / закрытости

2.1 Не использование абстракций

2.2 Проверка типа абстракции

2.3 Резюме

3. Принцип подстановки Лискоу

3.1 Ошибочное наследование

3.2 Проектирование на контрактной основе

3.3 Резюме

4. Принцип отделения интерфейса

4.1 Лишняя абстракция в наследовании

4.2 Резюме

5. Принцип инверсии зависимостей

5.1 Резюме

Заключение

Литература

**Введение**

Любая полезная программа постепенно изменяется, в нее вносятся новые возможности. Очевидно, что с кодом что-то не так, если для небольшого изменения приходится переписывать значительную часть системы. Под чистым кодом понимают код, который обеспечивает удобство сопровождения и безболезненность внесения изменений.

Чистый код должен быть не только оформлен определенным образом, но и иметь определенную структуру. Существует множество принципов, которым должен соответствовать хороший программный проект, а также ряд техник, позволяющих привести его в соответствие с этими принципами. Процесс очищения кода называют рефакторингом.

Далее будут описаны пять основных принципов, которым должен соответствовать хороший объектно-ориентированный проект — **SOLID** — это аббревиатура от пяти принципов проектирования, названия которых удачно зашифровал в этом слове Роберт Мартин.

Принципы SOLID:  
1. Принцип единственности ответственности **(*The Single Responsibility Principle*)**  
2. Принцип открытости/закрытости **(*The Open Closed Principle*)**  
3. Принцип подстановки Барбары Лисков **(*The Liskov Substitution Principle*)**  
4. Принцип разделения интерфейса **(*The Interface Segregation Principle*)**  
5. Принцип инверсии зависимостей **(*The Dependency Inversion Principle*)**

**Принципы объектно-ориентированного проектирования SOLID**

1. **Принцип единственности ответственности**

**Не должно быть больше одной причины для изменения класса**

Что является причиной изменения логики работы класса? Видимо, изменение отношений между классами, введение новых требований или отмена старых. Если у объекта много ответственности, то и меняться он будет очень часто. Таким образом, если класс имеет больше одной ответственности, то это ведет к хрупкости дизайна и ошибкам в неожиданных местах при изменениях кода.

Таким образом, принцип единственности ответственности можно переформулировать следующим образом: ***Конкретный класс должен решать конкретную задачу — ни больше, ни меньше.***

* 1. **Валидация данных**

**Проблема**

При создании проекта, перед разработчиками всегда стояла проблема валидации данных. Например, проверка введенного адреса эл. почты, длины имени пользователя, сложности пароля и т.п. Для валидации объекта резонно возникает первая реализация:

1: public class Product

2: {

3: public int Price { get; set; }

4:

5: public bool IsValid()

6: {

7: return Price > 0;

8: }

9: }

10:

11: // проверка на валидность

12: var product = new Product { Price = 100 };

13: var isValid = product.IsValid();

Такой подход является вполне оправданным в данном случае. Код простой, тестированию поддается, дублирования логики нет.

Теперь наш объект Product начал использоваться в каком-либо CustomerService, который считает действительным продукт с ценой более 100 тыс. рублей. Необходимо изменять метод валидации объекта продукта, например, таким образом:

1: public class Product

2: {

3: public int Price { get; set; }

4:

5: public bool IsValid(bool isCustomerService)

6: {

7: if (isCustomerService == true)

8: return Price > 100000;

9:

10: return Price > 0;

11: }

12: }

13:

14: // используем объект продукта в новом сервисе

15: var product = new Product { Price = 100 };

16: var isValid = product.IsValid(true);

**Решение**

При дальнейшем использовании объекта Product логика валидации его данных будет изменяться и усложняться. По принципу единственности ответственности следует отдать ответственность за валидацию данных продукта другому объекту. Причем надо сделать так, чтобы сам объект продукта не зависел от конкретной реализации его валидатора. Получаем код:

1: public interface IProductValidator

2: {

3: bool IsValid(Product product);

4: }

5:

6: public class ProductDefaultValidator : IProductValidator

7: {

8: public bool IsValid(Product product)

9: {

10: return product.Price > 0;

11: }

12: }

13:

14: public class CustomerServiceProductValidator : IProductValidator

15: {

16: public bool IsValid(Product product)

17: {

18: return product.Price > 100000;

19: }

20: }

21:

22: public class Product

23: {

24: private readonly IProductValidator validator;

25:

26: public Product() : this(new ProductDefaultValidator())

27: {

28: }

29:

30: public Product(IProductValidator validator)

31: {

32: this.validator = validator;

33: }

34:

35: public int Price { get; set; }

36:

37: public bool IsValid()

38: {

39: return validator.IsValid(this);

40: }

41: }

42:

43: // обычное использование

44: var product = new Product { Price = 100 };

45:

46: // используем объект продукта в новом сервисе

47: var product = new Product (new CustomerServiceProductValidator()) { Price = 100 };

Имеем объект Product отдельно, а любое количество различных валидаторов отдельно.

**1.2. «Объект Бога» (God Object)**

**Проблема**

Предел нарушения принципа единственности ответственности – God object («Объект Бога»). Этот объект способен выполнять все операции, которые только возможно. Например, он делает запросы к базе данных, к файловой системе, общается по протоколам в сеть и содержит большое количество бизнес-логики. Рассмотрим пример такого объекта, который называется ImageHelper:

1: public static class ImageHelper

2: {

3: public static void Save(Image image)

4: {

5: // сохранение изображение в файловой системе

6: }

7:

8: public static int DeleteDuplicates()

9: {

10: // удалить из файловой системы все дублирующиеся изображения и вернуть количество удаленных

11: }

12:

13: public static Image SetImageAsAccountPicture(Image image, Account account)

14: {

15: // запрос к базе данных для сохранения ссылки на это изображение для пользователя

16: }

17:

18: public static Image Resize(Image image, int height, int width)

19: {

20: // изменение размеров изображения

21: }

22:

23: public static Image InvertColors(Image image)

24: {

25: // изменить цвета на изображении

26: }

27:

28: public static byte[] Download(Url imageUrl)

29: {

30: // загрузка битового массива с изображением с помощью HTTP запроса

31: }

32:

33: // и т.п.

34: }

Кажется, что границы ответственности у него вообще нет. Он может сохранять в базу данных, причем знает правила назначения изображений пользователям. Может скачивать изображения. Знает, как хранятся файлы изображений и может работать с файловой системой.

Каждая ответственность этого класса ведет к его потенциальному изменению. Получается, что этот класс будет очень часто менять свое поведение, что затруднит его тестирование и тестирование компонентов, которые его используют. Такой подход снизит работоспособность системы и повысит стоимость ее сопровождения.

**Решение**

Решением является разделить этот класс по принципу единственности ответственности: один класс на одну ответственность.

1: public static class ImageFileManager

2: {

3: public static void Save(Image image)

4: {

5: // сохранение изображение в файловой системе

6: }

7:

8: public static int DeleteDuplicates()

9: {

10: // удалить из файловой системы все дублирующиеся изображения и вернуть количество удаленных

11: }

12: }

13:

14: public static class ImageRepository

15: {

16: public static Image SetImageAsAccountPicture(Image image, Account account)

17: {

18: // запрос к базе данных для сохранения ссылки на это изображение для пользователя

19: }

20: }

21:

22: public static class Graphics

23: {

24:

25: public static Image Resize(Image image, int height, int width)

26: {

27: // изменение размеров изображения

28: }

29:

30: public static Image InvertColors(Image image)

31: {

32: // изменить цвета на изображении

33: }

34: }

35:

36:

37: public static class ImageHttpManager

38: {

39: public static byte[] Download(Url imageUrl)

40: {

41: // загрузка битового массива с изображением с помощью HTTP запроса

42: }

43: }

**1.3 Резюме**

Принцип SRP является одним из простейших, но в то же время одним из труднейших в плане применения. Объединение ответственности является общепринятой практикой. Поиск и отделение этих ответственностей во многом определяют функции программного проекта. Остальные принципы SOLID часто связаны с принципом SRP.

1. **Принцип открытости/закрытости**

**Программные сущности (классы, модули, функции и т.д.) должны быть открыты для расширения, но закрыты для изменения**

Какую цель мы преследуем, когда применяем этот принцип? Как известно программные проекты в течение свой жизни постоянно изменяются. Изменения могут возникнуть, например, из-за новых требований заказчика или пересмотра старых. В конечном итоге потребуется изменить код в соответствии с текущей ситуацией.

С одной стороны, внесение изменений требует времени программистов и тестировщиков, которое является очень дорогим ресурсом в производстве ПО. С другой, бизнес должен достаточно быстро реагировать на рыночные изменения и время здесь представляется очень важным конкурентным преимуществом.

Отсюда можно сделать вывод, что нашей целью является разработка системы, которая будет достаточно просто и безболезненно меняться.

Принцип открытости/закрытость как раз и дает понимание того, как оставаться достаточно гибкими в условиях постоянно меняющихся требований: ***Все классы, функции и т.д. должны проектироваться так, чтобы для изменения их поведения***  ***нам не нужно было изменять их исходный код.***

* 1. **Не использование абстракций**

**Проблема**

Самый простой пример нарушения принципа открытости/закрытости – использование конкретных объектов без абстракций. Предположим, что у нас есть объект *SmtpMailer*. Для логирования своих действий он использует *Logger*, который записывает информацию в текстовые файлы.

1: public class Logger

2: {

3: public void Log(string logText)

4: {

5: // сохранить лог в файле

6: }

7: }

8:

9: public class SmtpMailer

10: {

11: private readonly Logger logger;

12:

13: public SmtpMailer()

14: {

15: logger = new Logger();

16: }

17:

18: public void SendMessage(string message)

19: {

20: // отсылка сообщения

21:

22: logger.Log(string.Format("Отправлено '{0}'", message));

23: }

24: }

И тоже самое происходит в других классах, которые используют *Logger*. Такая конструкция вполне жизнеспособна до тех, пока мы не решим записывать лог *SmptMailer*'a в базу данных. Для этого нам надо создать класс, который будет записывать все логи не в текстовый файл, а в базу данных:

1: public class DatabaseLogger

2: {

3: public void Log(string logText)

4: {

5: // сохранить лог в базе данных

6: }

7: }

А теперь самое интересное. Мы должны изменить класс *SmptMailer* из-за изменившегося бизнес-требования:

1: public class SmtpMailer

2: {

3: private readonly DatabaseLogger logger;

4:

5: public SmtpMailer()

6: {

7: logger = new DatabaseLogger();

8: }

9:

10: public void SendMessage(string message)

11: {

12: // отсылка сообщения

13:

14: logger.Log(string.Format("Отправлено '{0}'", message));

15: }

16: }

Но ведь по [принципу единственности ответственности](http://blog.byndyu.ru/2009/10/blog-post.html) не *SmptMailer* отвечает за логирование, почему изменения дошли и до него? Потому что нарушен наш принцип открытости/закрытости. *SmptMailer* не закрыт для модификации. Нам пришлось его изменить, чтобы поменять способ хранения его логов.

**Решение**

В данном случае защитить *SmtpMailer* поможет выделение абстракции. Пусть *SmtpMailer* зависит от интерфейса *ILogger*:

1: public interface ILogger

2: {

3: void Log(string logText);

4: }

5:

6: public class Logger : ILogger

7: {

8: public void Log(string logText)

9: {

10: // сохранить лог в файле

11: }

12: }

13:

14: public class DatabaseLogger : ILogger

15: {

16: public void Log(string logText)

17: {

18: // сохранить лог в базе данных

19: }

20: }

21:

22: public class SmtpMailer

23: {

24: private readonly ILogger logger;

25:

26: public SmtpMailer(ILogger logger)

27: {

28: this.logger = logger;

29: }

30:

31: public void SendMessage(string message)

32: {

33: // отсылка сообщения

34:

35: logger.Log(string.Format("Отправлено '{0}'", message));

36: }

37: }

Теперь смена логики логирования уже не будет вести к модификации *SmtpMailer*'а.

**2.2 Проверка типа абстракции**

**Проблема**

Этот пример в разных вариациях все не раз видели в коде. Его хоть в рамку можно вешать, как самое популярное нарушение проектирования. У нас есть иерархия объектов с абстрактным родительским классом *AbstractEntity* и класс *Repository*, который использует абстракцию. При этом вызывая метод *Save* у *Repository* мы строим логику в зависимости от типа входного параметра:

1: public abstract class AbstractEntity

2: {

3: }

4:

5: public class AccountEntity : AbstractEntity

6: {

7: }

8:

9: public class RoleEntity : AbstractEntity

10: {

11: }

12:

13: public class Repository

14: {

15: public void Save(AbstractEntity entity)

16: {

17: if (entity is AccountEntity)

18: {

19: // специфические действия для AccountEntity

20: }

21: if (entity is RoleEntity)

22: {

23: // специфические действия для RoleEntity

24: }

25: }

26: }

Из кода видно, что объект Repository придется менять каждый раз, когда мы добавляем в иерархию объектов с базовым классом AbstractEntity новых наследников или удаляем существующих. Условные операторы будут множится в методе Save и тем самым усложнять его.

**Решение**

Чтобы решить данную проблему, необходимо логику сохранения конкретных классов из иерархии AbstractEntity вынести в конкретные классы Repository. Для этого мы должны выделить интерфейс IRepository и создать хранилища AccountRepository и RoleRepository:

1: public abstract class AbstractEntity

2: {

3: }

4:

5: public class AccountEntity : AbstractEntity

6: {

7: }

8:

9: public class RoleEntity : AbstractEntity

10: {

11: }

12:

13: public interface IRepository<T> where T : AbstractEntity

14: {

15: void Save(T entity);

16: }

17:

18: public class AccountRepository : IRepository<AccountEntity>

19: {

20: public void Save(AccountEntity entity)

21: {

22: // специфические действия для AccountEntity

23: }

24: }

25:

26: public class RoleRepository : IRepository<RoleEntity>

27: {

28: public void Save(RoleEntity abstractEntity)

29: {

30: // специфические действия для RoleEntity

31: }

32: }

Теперь наши изменения будут локализованы в конкретных объектах.

**2.3 Резюме**

Во многих отношениях принцип открытости/закрытости лежит в основе объектно-ориентированного проектирования. Соответствие этому принципу порождает большие преимущества? Характерные для объектно-ориентированной технологии (т.е. гибкость, повторное использование и удобство эксплуатации). Но в то же время соответствие этому принципу не достигается лишь использованием объектно-ориентированного языка программирования. Не следует также прибегать к чрезмерным абстракциям для каждой части приложения. Вместо этого разработчики советуют применять абстракции только тех частей программы, которые подвержены частым изменениям. Вопрос предотвращения непродуманных абстракций имеет такое же значение, как и сама абстракция.

1. **Принцип подстановки Лискоу**

**Если для каждого объекта o1 типа S существует объект o2 типа T, который для всех программ P определен в терминах T, то поведение P не изменится, если o2 заменить на o1 при условии, что S является подтипом T.**

Значимость этого правила становится очевидной в случае, если рассмотреть последствия его нарушений. Предположим, что у нас есть функция *f*, содержащая в качестве аргумента указатель или ссылку на базовый класс В. Также представим, что существует производная от В (сокращённо D), которая при подстановке в функцию *f* под видом B вызывает изменения в поведении последней. В этом случае D игнорирует принцип LSP. Очевидно, что D представляет собой одну из «неустойчивостей» *f*.

Разработчики функции *f* могут подвергнуться искушению провести определенные тесты, которые покажут, что поведение функции не изменяется при подстановке в нее производной функции D. Такое тестирование отрицает основные принципы OCP, поскольку оно не охватывает весь диапазон значений производных от B. Обычно такие ошибки совершают неопытные разработчики (или решения принимаются в спешке).

**3.1 Ошибочное наследование**

**Проблема**

Мы хотим реализовать свой список с интерфейсом IList. Его особенностью будет то, что все записи в нем дублируются.

1: public class DoubleList<T> : IList<T>

2: {

3: private readonly IList<T> innerList = new List<T>();

4: public void Add(T item)

5: {

6: innerList.Add(item);

7: innerList.Add(item);

8: }

9: . . .

Данная реализация не представляет никакой опасности, если рассматривать ее изолированно. Взглянем на использование этого класса с точки зрения клиента. Клиент, абстрагируясь от реализаций, пытается работать со всеми объектами типа *IList* одинаково:

1: [Fact]

2: public void CheckBehaviourForRegularList()

3: {

4: IList<int> list = new List<int>();

5: list.Add(1);

6: Assert.Equal(1, list.Count);

7: }

8: [Fact]

9: public void CheckBehaviourForDoubleList()

10: {

11: IList<int> list = new DoubleList<int>();

12: list.Add(1);

13: Assert.Equal(1, list.Count); // ошибка

14: }

Поведение списка *DoubleList* отличается от типичных реализаций *IList*. Получается, что *DoubleList* не может быть заменен базовым типом. Это и есть нарушение принципа подстановки Лисков. Проблема заключается в том, что теперь клиенту необходимо знать о конкретном типе объекта, реализующем интерфейс *IList*. В качестве такого объекта могут передать и *DoubleList*, а для него придется выполнять дополнительную логику и проверки.

**Решение**

Правильным решением будет использовать свой собственный интерфейс, например, IDoubleList. Этот интерфейс будет объявлять для пользователей поведение, при котором добавляемые элементы удваиваются.

**3.2 Проектирование на контрактной основе**

Многие разработчики не очень хорошо представляют себе «разумное» и «рациональное» поведение объектов. Как узнать, чего клиенты ждут от вас? Существует технология, позволяющая это выяснить, не нарушая при этом принципа LSP. Эта методика была разработана Бертраном Мейером и называется проектированием по контракту (DBC – design by contract).

При использовании этой методики автор отдельного класса составляет для него так называемый контракт. Данный контракт информирует авторов клиентского кода об ожиаемом поведении данного класса. Здесь указываются предварительные и выходные условия для каждого применяемого метода. Для успешного применения технологии DBC во входных условиях не должно быть ошибок. По завершению работы DBC гарантирует отсутствие ошибок в выходных условиях.

**3.3 Резюме**

Принцип LSP – один из ведущих принципов OCP. Взаимозаменяемость подтипов позволяет расширять функциональные возможности модуля, основанного на базовом типе, не прибегая к его модификации. Разработчики должны доверять тому принципу без малейших колебаний. Таким образом, контракт базового типа не должен содержать ошибок и быть, по возможности, интуитивно понятным, если все его элементы сразу не содержатся в программном коде.

1. **Принцип отделения интерфейса**

**Клиенты не должны зависеть от методов, которые они используют**

Этот принцип позволяет преодолевать недостатки, связанные с чрезмерной «толщиной» интерфейсов. Классы, имеющие «тучные» интерфейсы, недостаточно компактны. Эти интерфейсы можно разбить на группы методов, и каждая группа обслуживает иной набор клиентов. Поэтому каждая группа клиентов использует определенную группу членов-функций.

Принцип ISP ориентируется на наличие объектов, нуждающихся в несвязанных интерфейсах; также учитывается, что клиенты не должны представлять их как отдельный класс. Вместо этого клиентам следует иметь представление об абстрактных базовых классах, имеющих связанные интерфейсы.

**4.2 Лишняя абстракция в наследовании**

**Проблема**

Речь идет о базовых классах, которые вынуждают своих наследников знать и делать слишком много. Печально известный пример – класс *[MembershipProvider](https://msdn.microsoft.com/ru-ru/library/system.web.security.membershipprovider.aspx)*. Для использования этого класса нужно реализовать 27 абстрактных методов и свойств.

Есть базовый класс для аудиторов *EntityAuditor*. Он унаследован от класса *AuditorBase*, который предоставляет ORM, и реализует метод *AuditEntityFieldSet* этого базового класса. Также *EntityAuditor* добавляет свой абстрактный метод *CreateLogRow*, который используется в методе *AuditEntityFieldSet* и должен быть переопределен в конкретных реализациях:

1: public abstract class EntityAuditor : AuditorBase

2: {

3:    public override void AuditEntityFieldSet(IEntityCore entity, int fieldIndex, object originalValue)

4:    {

5:        // ...

6:        CreateLogRow(...)

7:        // ...

8:    }

9:    protected abstract LogRowEntity CreateLogRow(decimal? fieldId, string oldValue, string newValue, IEntityCore entity);

10: }

После этого начинаем реализовывать наследников. Например, создадим аудитор для класса *Product*:

1: public class ProductAuditor : EntityAuditor

2: {

3:    protected override LogRowEntity CreateLogRow(decimal? fieldId, string oldValue, string newValue, IEntityCore entity)

4:    {

5:        // ...

6:    }

7: }

Сейчас добавлению наследников ничего не мешает. Теперь представим, что в методе *AuditEntityFieldSet* понадобилась дополнительная логика, при которой нужно вызвать метод *UpdateDuplicates*. Этот метод также является абстрактным и требует реализации в наследниках:

public abstract class EntityAuditor : AuditorBase

{

    public override void AuditEntityFieldSet(IEntityCore entity, int fieldIndex, object originalValue)

    {

       // ...

      CreateLogRow(...)

      UpdateDuplicates(...)

     // ...

    }

    protected abstract LogRowEntity CreateLogRow(decimal? fieldId, string oldValue, string newValue, IEntityCore entity);

    protected abstract void UpdateDuplicates(IEntityCore entity, decimal fieldId, object current);

}

public class ProductAuditor : EntityAuditor

{

    protected override LogRowEntity CreateLogRow(decimal? fieldId, string oldValue, string newValue, IEntityCore entity)

    {

        // ...

    }

    protected override void UpdateDuplicates(IEntityCore entity, decimal fieldId, object current)

    {

        // реализация

    }

}

public class AccountAuditor : EntityAuditor

{

    protected override LogRowEntity CreateLogRow(decimal? fieldId, string oldValue, string newValue, IEntityCore entity)

    {

        // ...

    }

    protected override void UpdateDuplicates(IEntityCore entity, decimal fieldId, object current)

    {

        // здесь ничего нет

    }

}

*EntityAuditor* требует реализации метода *UpdateDuplicates* даже в тех наследниках, где он не нужен, как, например, в *AccountAuditor*. Проблема в том, что частный случай (*UpdateDuplicates*), который используется только в половине наследников, мы сделали общим, т.е. обязательным для всех наследников нашего *EntityAuditor*. Получается, что чем больше наследников будет у *EntityAuditor*, тем больше бесполезного кода мы будем писать, тем больше наследники будут знать лишнего о своем базовом классе. Это может сильно помешать нам в дальнейшем при рефакторинге или изменении логики в *EntityAuditor*.

**Решение**

В данном случае решение очень простое. Если наследникам класса *EntityAuditor* не нужна функция *UpdateDuplicates*, то и реализовывать ее они не должны.

**Резюме**

«Тучные» классы могут приводить к причудливым и нежелательным стыкам между клиентами. Когда один клиент вносит изменение в подобный класс, это оказывает влияние на всех остальных клиентов. Поэтому клиенты должны зависеть от методов, которые ими вызываются. Чтобы достичь этого, интерфейс подобных «тучных» классов разбивается на большое количество интерфейсов, специфичных для каждого клиента. Каждый интерфейс, связанный с определённым клиентом, объявляет только функции, которые вызываются данным клиентом или клиентской группой. Затем «тучный» класс может наследовать все интерфейсы, связанные с определёнными клиентами, и реализовывать их. Таким образом, прерывается зависимость между клиентами и методами, к которым они не обращаются, что позволяет поддерживать независимость клиентов друг от друга.

1. **Принцип инверсии зависимостей**

**Модули высокого уровня не должны зависеть от модулей низкого уровня. Оба типа модулей обязаны зависеть от абстракций. Абстракции не должны зависеть от деталей. Деталям следует зависеть от абстракций.**

Почему в названии данного принципа проектирования используется слово «инверсия»? Причина в том, что более традиционные методы разработки ПО, такие как структурированный анализ и проектирование, обычно применяются ля создания программных структур, в которых наблюдается зависимость модулей высокого уровня от модулей низкого уровня. Политика применения подобных методов зависит от имеющихся деталей. Более того, эти методы служат для определения подпрограммой иерархии, уточняющей принципы обращения модулей высокого уровня к модулям низкого уровня.

Рассмотрим зависимость модулей высокого уровня от низкоуровневых модулей. Именно модули высокого уровня включают важные политические решения и бизнес-модели приложения. Они описывают суть приложения. Но до тех пор, пока эти модули зависят от модулей низкого уровня, изменения последних могут непосредственно влиять на функционирование модулей более высокого уровня.

В итоге формируется «замкнутый круг». Существуют модули высокого уровня, устанавливающие определенные политики, но на них влияют в целом детали модулей низкого уровня. Модули, содержащие описания бизнес-правил высокого уровня, должны иметь определенную независимость и приоритет перед модулями, включающими детали. Модули высокого уровня ни в коей мере не должны зависеть от модулей низкого уровня.

Более того, к высокоуровневым модулям, определяющим основные положения политик, приходиться обращаться повторно. Модули низкого уровня достаточно удобно использовать повторно в виде библиотек подпрограмм. Если модули высокого уровня зависят от модулей низкого уровня, довольно сложно использовать высокоуровневые модули в различных контекстах. Но если модули высокого уровня не зависят от модулей низкого уровня, высокоуровневые модули легко применять повторно. Этот принцип положен в основу схематического проектирования (framework design).

**Резюме**

Традиционное процедурное программирование формирует структуру зависимостей, при которой политика зависит от деталей. Это весьма неудобно, поскольку политики чутко реагируют на изменение в деталях. ООП инвертирует эту структуру зависимости, тогда и детали, и политики зависят от абстракции, а сервисные интерфейсы часто оказываются в собственности у клиентов.

Действительно, инверсия зависимостей является признаком хорошо выполненной объектно-ориентированной разработки. И не имеет значения, какой язык программирования при этом используется. Если зависимости инвертированы, речь идёт об объектно-ориентированной разработке. Если зависимости не инвертированы, имеет место процедурная разработка.

Принцип инверсной зависимости является фундаментальным механизмом низкого уровня, находящимся в основе многих выгодных аспектов, отличающих объектно-ориентированную технологию. При создании повторно применимых схем (frameworks) необходимо соответствующим образом применять эту технологию. Также этот подход имеет критическое значение при конструировании гибко трансформируемых кодов. Если абстракции и детали изолированы друг от друга, код значительно легче поддерживать в будущем.

**Заключение**

Среда профессиональной разработки ставит много трудных задач перед инженерами ПО. Полученное образование приучило их думать о решении задач, рассматривая их сверху вниз. В начальных заданиях в мире внутрикорпоративных приложений такой подход ещё будет уместен. Однако очень скоро обнаруживается, что функция верхнего уровня разрослась до неприличного размера. Внесение даже самых мелких изменений требует полного знания всей системы, и держать их под контролем почти невозможно. Следование принципам построения программного обеспечения поможет избежать развития его конструкции, опережающего расширение ее фундамента.

**Литература**

Роберт К. Мартин, Джеймс В. Ньюкирк, Роберт С. Косс – Быстрая разработка программ. Принципы, примеры, практика. — Издательский дом «Вильямс», 2004 — 164 - 232 с.

Блог Александра Бындю — <http://blog.byndyu.ru/2009/10/solid.html>

Блог Муратов Мурат — <http://blog.muradovm.com/2012/03/solid.html>

Роберт Мартин — <http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod>

Wikipedia — <https://goo.gl/9VyExX>

Microsoft Developer Network — <https://goo.gl/Dd2lYd>