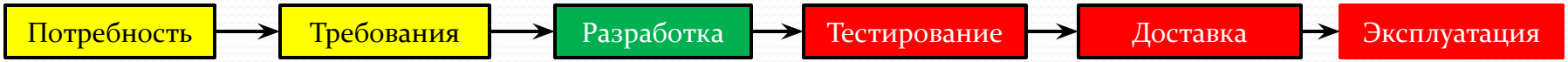
Гунба С.В.

Архитектура ПО

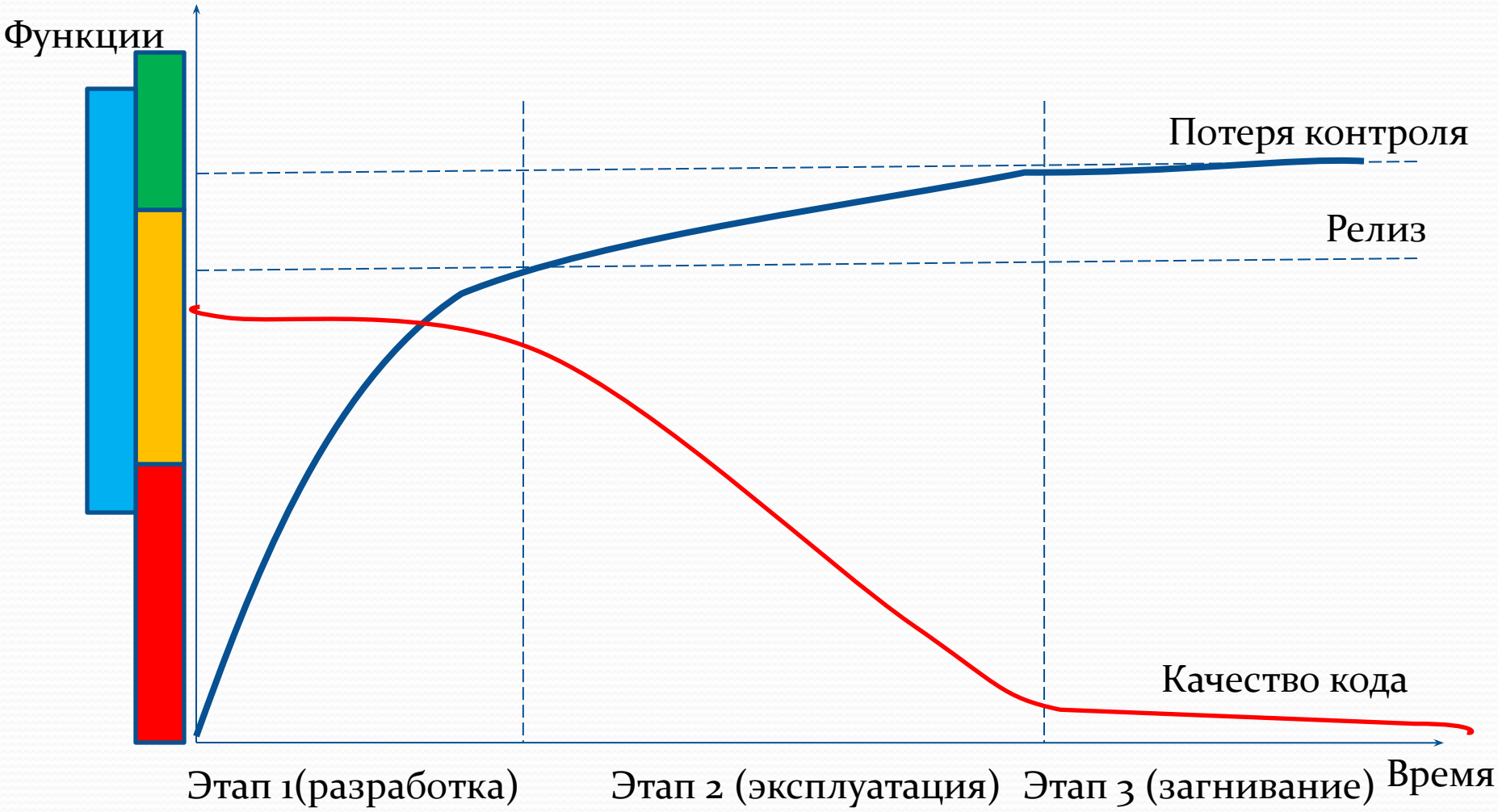
DevOps итоги



Узкими местами потока создания ценности являются:

1. **Тестирование**
    1. Большое время обратной связи «разработка–>тест-> дефект»
    2. Много времени занимает «ретроспективное тестирование»
    3. Тестирование поверхностное (не тестируются негативные сценарии)
2. Развертывание
    1. Ручная доставка новых версий пользователю
    2. Нет понятия «Релиз» системы
3. Эксплуатация
    1. Нет данных о реальной работе системы на объектах

Развитие функционала



Качество кода

Качество кода – достаточно сложное понятие. Мы рассмотрим 3 параметра:

1. Работоспособность – способность выполнять задачи
2. Модифицируемость – возможность вносить изменения без потери работоспособности (*Архитектурное понятие*)
3. Минимальное число зависимостей – взаимосвязей между частями приложения (*Архитектурное понятие*)

Причины падения качества кода:

1. Изначально некачественная разработка (при разработке добиваемся только работоспособности, не следим за *архитектурой*)
2. Затыкание дыр – быстрое устранение дефектов без анализа последствий для *архитектуры* проекта
3. Наличие «неизменяемых функций»– например подключение новой СУЛ, которой нет в доступе. Изменять такие фрагменты страшно.
4. Отсутствие качественного тестирования после модификации или введения нового функционала.
5. Введение «голубого функционала» который не нужен впоследствии и только загрязняет код

Запахи плохого дизайна архитектуры

Жесткость

Жесткость – это характеристика программы, затрудняющая внесение в нее изменений, даже самых простых. Дизайн жесткий, если единственное изменение вызывает целый каскад других изменений в зависимых модулях. Чем больше модулей приходится изменять, тем жестче дизайн.

Хрупкость

Хрупкость – это свойство программы повреждаться во многих местах при внесении единственного изменения. Зачастую новые проблемы возникают в частях, не имеющих концептуальной связи с той, что была изменена.

Косность

Дизайн является косным, если он содержит части, которые могли бы оказаться полезны в других системах, но усилия и риски, сопряженные с попыткой отделить эти части от оригинальной системы, слишком велики.

Вязкость

Сталкиваясь с необходимостью внести изменение, разработчик обычно находит несколько способов сделать это. Одни сохраняют дизайн, другие – нет(хак). Если сохраняющие дизайн подходы оказывается реализовать труднее, чем «хак», то вязкость дизайна высока.

Запахи плохого дизайна архитектуры

Ненужная сложность

Дизайн попахивает ненужной сложностью, если содержит элементы неиспользуемые в текущий момент. Это часто случается, когда разработчики стараются предвидеть будущие изменения требований и вставлять в программу средства для их поддержки.

Ненужные повторения

Если в системе есть дублирующийся код, то задача ее изменения может потребовать значительных усилий. Ошибки, обнаруженные в повторяющемся блоке, должны быть исправлены во всех его копиях.

Непрозрачность

Непрозрачность – это трудность модуля для понимания. Код может быть ясным и выразительным или темным и запутанным. Код, эволюционирующий со временем, постепенно становится все более и более непрозрачным. Дабы свести непрозрачность к минимуму, нужно постоянно следить, чтобы он оставался ясным и выразительным.

Методы борьбы с ухудшением качества кода

Рефакторинг – изменение внутренней структуры ПО без изменения его поведения.

В рефакторинг входит:

1. Переименование классов, функций, переменных.
2. Переработка алгоритмов, при условии сохранения входных и выходных данных
3. Переработка архитектуры

Рефакторинг проводится на разных уровнях приложения:

1. На уровне минимальных «единиц» кода (на уровне function или class)
2. На уровне крупных компонентов (на уровне бизнес логики, канального уровня, интерфейсного уровня)
3. На уровне всего приложения (рефакторинг архитектуры)

Рефакторинг на уровне минимальных «единиц» кода

Выводы

- Основной причиной загнивания проекта является ухудшение качества кода и «размытие» архитектуры.
- Признаками «размытия» архитектуры являются «запахи»
- Для сохранения качества кода необходимо проводить рефакторинг на различных уровнях
- Неотъемлемой частью рефакторинга является тестирование!
- При этом тестирование производится многократно на каждом этапе и на каждом уровне рефакторинга.

А это значит что тестирование следует автоматизировать!!!

А возможность автоматизации тестирования зависит от архитектуры приложения и от архитектуры его компонентов.

Архитектура ПО

Архитектура ПО – это внутренний атрибут качества ПО и разрабатывается она в зависимости от специфики разрабатываемого ПО.

Общепринятого определения понятия «Архитектура ПО» не существует.

Мы будем рассматривать Архитектуру ПО как искусство или науку проведения **границ** и установления **зависимостей**.

Факты об архитектуре ПО:

1. Разработка архитектуры требует времени
2. Поддержание архитектуры требует времени
3. Архитектура это долгосрочная инвестиция! Затраты на хорошую архитектуру окупятся при дальнейшей разработке проекта.

Границы

Границы отделяют программные элементы друг от друга и избавляют их от необходимости знать, что находится по ту сторону.

Границы часто только умозрительны.

«Физические» воплощения границ:

1. Функция
2. Класс (понятие из ООП)
3. Интерфейс (java)
4. Программный модуль (*.h + *.c)
5. Компонент (скомпилированный модуль, dll и тд)

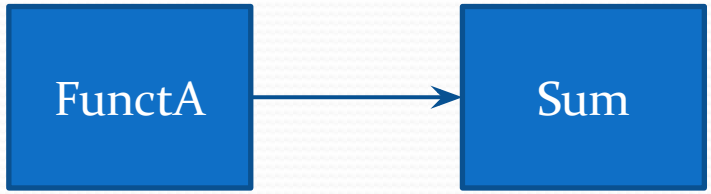
Пересечение границы – это вызов функции (метода)

Зависимости

Функции (классы) считаются зависимыми если одна функция вызывает другую функцию.

Зависимость обозначается стрелкой направленной в сторону зависимости.

```
void FunctA (){
  m= Sum (x,y);
}
```

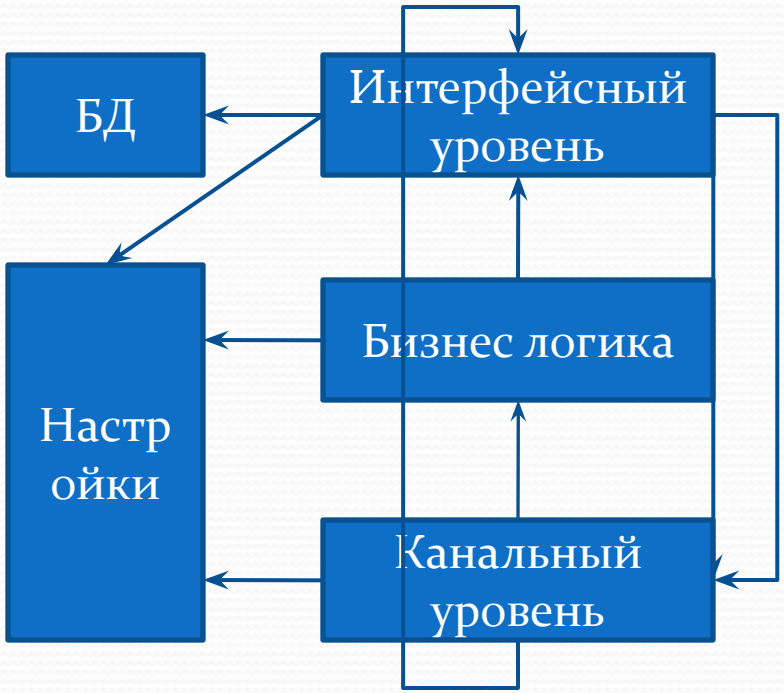


Важно понимать! Изменение поведения функции **Sum** окажет влияние на поведение функции **FunctA**

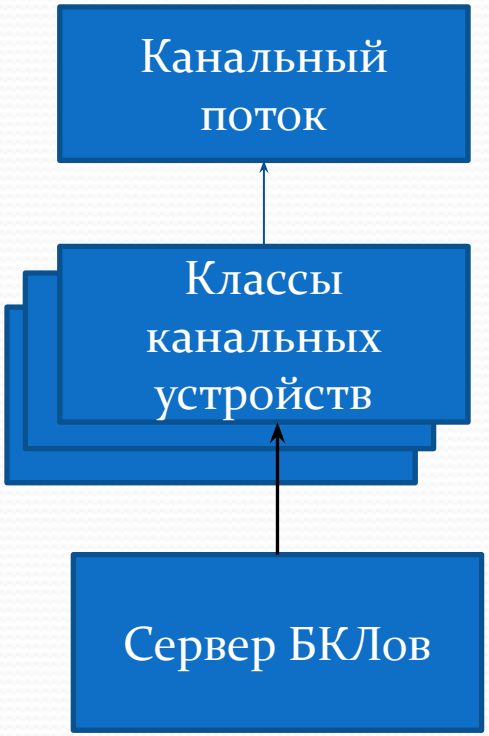
Изменение поведения функции FunctA не окажет никакого влияния на функцию Sum.

Уровень абстракции

Уровень абстракции — один из способов сокрытия деталей реализации определенного набора функциональных возможностей. Применяется для управления сложностью проектируемой системы при декомпозиции когда система представляется в виде иерархии уровней абстракции.



Общая структура ПО «КУПОЛ»



Канальный уровень

Гексагональная архитектура

В приложении выделяют основные слои:

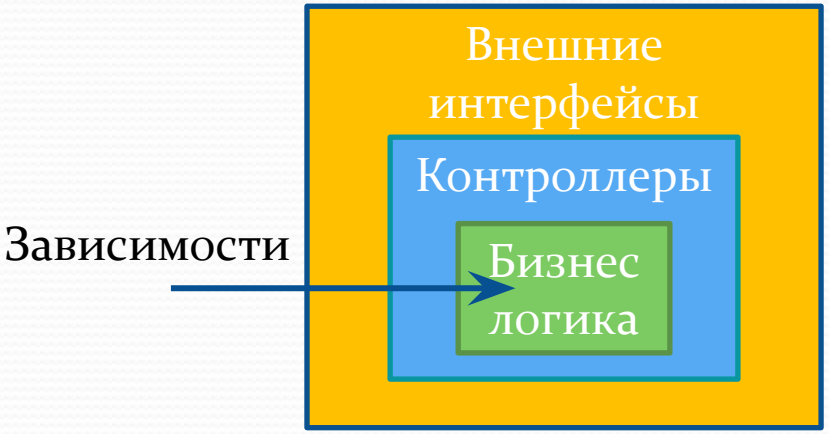
Внешние интерфейсы – обеспечивают взаимодействие приложения с внешним миром (ввод / вывод данных, взаимодействие с другими системами или частями приложения)

Контроллеры – обеспечивает поток управления, преобразование данных от интерфейса в бизнес логику, взаимодействие с бизнес логикой и управление выводом.

Бизнес логика – реализует основную цель приложения, содержит сложные алгоритмы обработки данных.

При этом **зависимости** направлены от внешних слоев вовнутрь.

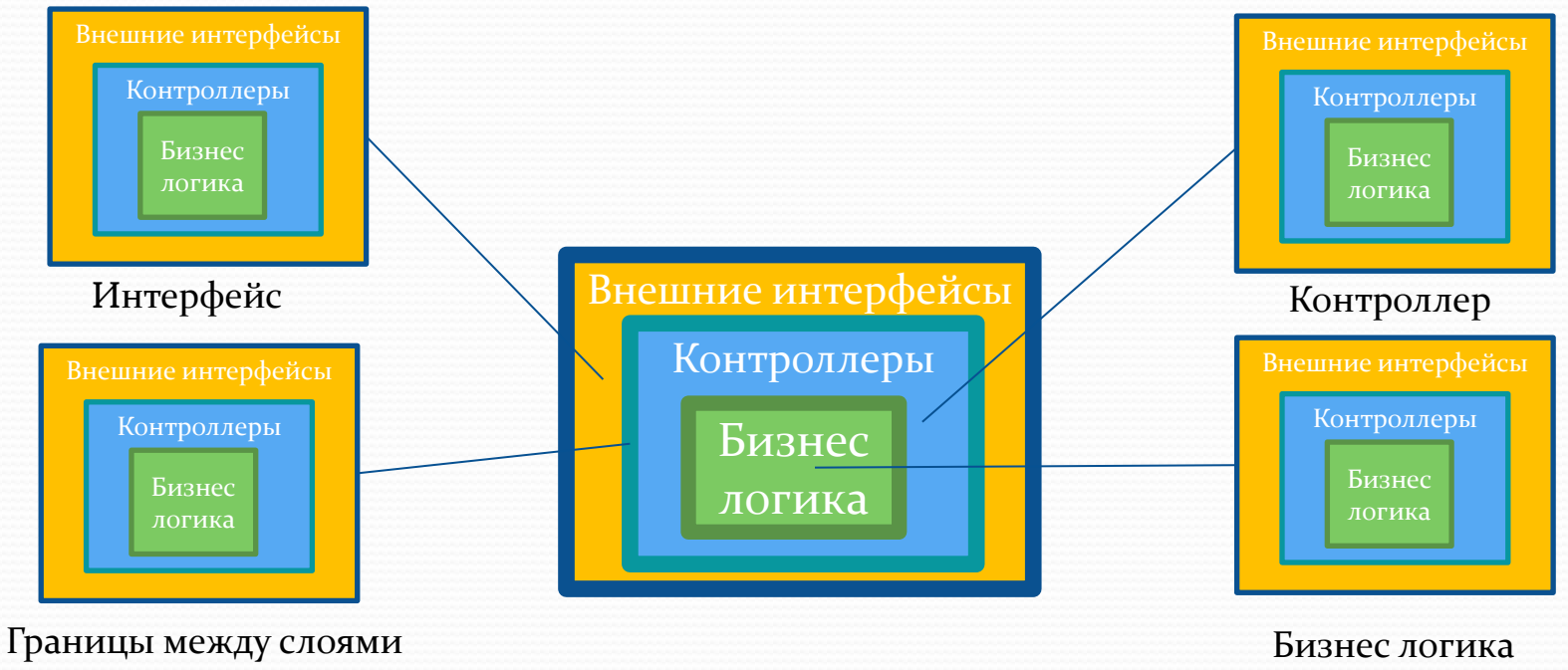
На границах между слоями должны быть **интерфейсы**



Гексагональная архитектура ПО на различных уровнях

Классы, компоненты и границы из которых состоит приложение на различных уровнях, так же построены согласно гексагональной архитектуре.

Например поле ввода IP адреса в рамках приложения это «внешний интерфейс». А для «внешнего интерфейса» это отдельный класс, который обладает интерфейсом, логикой (определение корректности IP) контроллером.



Принципы построения ПО по гексагональной архитектуре

1. Бизнес логика не должна иметь внешних зависимостей
2. Бизнес логика должна быть аналогом «математической функции». Другими словами иметь четкие входные данные и четкие выходные данные.
3. Все «внешние» взаимодействия должны производиться через контроллер
4. Контроллер должен быть максимально простым, вся сложная логика должна быть в бизнес логике
5. В общем случае контроллер не должен быть зависимым от слоя внешнего интерфеса (для встраиваемого ПО есть исключения)
6. Другими словами: Бизнес логика не должна ничего знать о контроллере и внешнем интерфейсе, контроллер ничего не должен знать об реализации внешнего интерфейса(быть отвязанным), внешний интерфейс ничего не должен знать о бизнес логике.

Принципы SOLID

1. **SRP Принцип единой ответственности** – каждый модуль должен иметь только одну причину для изменения. Иметь одну ответственность.
2. **OCP**: Open-Closed Principle — принцип открытости/закрытости. Добавление новых функций должно вводится дополнением нового кода, а не изменением старого.
3. **LSP**: Liskov Substitution Principle — принцип подстановки Барбары Лисков. Сохранения интерфейсов между компонентами.
4. **ISP**: Interface Segregation Principle — принцип разделения интерфейсов. Этот принцип призывает разработчиков программного обеспечения избегать зависимости от всего, что не используется.
5. **DIP**: Dependency Inversion Principle — принцип инверсии зависимости. Код, реализующий высокоуровневую политику, не должен зависеть от кода, реализующего низкоуровневые детали. Напротив, детали должны зависеть от политики.

Принцип единой ответственности

- Оригинальное описание: «*У класса должна быть только одна причина для изменения.*»

Другими словами у класса (функции) должна быть только одна ответственность. Класс (функция) должны выполнять только одно действие.

Пример

```
void FunctionF(k,x,a){
 y = k*x + a;  // Вычисление функции
Printf(“y = “,y );}//Вывод на экран
```

Следствия :

В коде не должно быть дублирования. Если есть дублирование, то необходимо от него избавиться через функцию, макрос или наследование. Первый признак дублирование – copy/paste кода.

Связанное понятие «**Зацепление**» - мера «Специализированности» класса или функции. Высокое зацепление – принцип соблюдается, низкое – нет.

Принцип открытости/закрытости (OCP)

Оригинальное звучание: *Программные сущности (классы, модули, функции и т. п.) должны быть открыты для расширения, но закрыты для модификации.*

У модулей, согласованных с принципом OCP, есть две основных характеристики.

1. Они *открыты для расширения*. Это означает, что поведение модуля можно расширить. Когда требования к приложению изменяются, мы добавляем в модуль новое поведение, отвечающее изменившимся требованиям. Иными словами, мы можем изменить состав функций модуля.

2. Они *закрыты для модификации*. Расширение поведения модуля не сопряжено с изменениями в исходном или двоичном коде других модулей ПО.

Это принцип больше относится к ООП и реализуется с помощью интерфейсов и полиморфизма.

Понятие «**Связанность**» - это взаимодействие между классами (модулями). Если взаимодействие производится по строго определённым интерфейсам, то связанность низкая и это хорошо. А если взаимодействие производится на уровне переменных, то связанность высокая и это плохо.

Принцип подстановки Лисков (LSP)

Оригинальное описание: *Должна быть возможность вместо базового типа подставить любой его подтип*.

- Этот принцип относится к ООП.

Описание другими словами: Потомки базового класса должны поддерживать интерфейсы родителя.

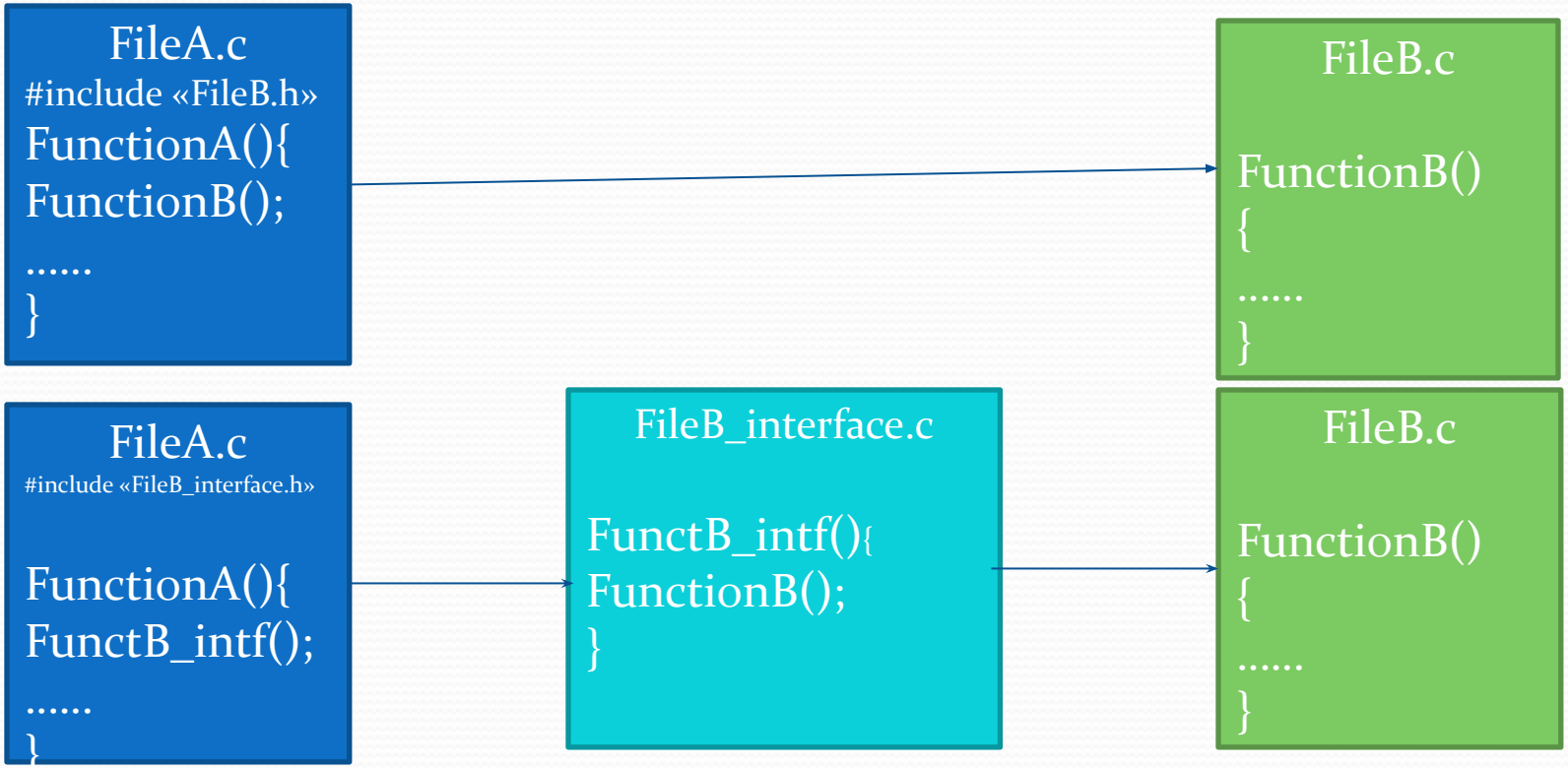
Признаки нарушение LSP: Потомок исключает функциональность базового класса и его нельзя использовать в приложении вместо базового.

Принцип инверсии зависимости (DIP) в ООП

Оригинальное звучание:

- *Модули верхнего уровня не должны зависеть от модулей нижнего уровня. И те и другие должны зависеть от абстракций.*
- *Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.*

Принцип инверсии зависимости (DIP) в Си



Еще можно разделить зависимости с помощью указателей на функции. Из функции FunctionA() вызывать функцию FunctionB() через указатель, предварительно его инициализировав.

Шаблоны проектирования

Pattern design – наборы типовых решений часто встречающихся задач при разработке ПО. Их также называют идиомами.

Шаблонов проектирования достаточно много, они имеют разную степень абстракции и разное назначение.

Мы интуитивно применяем шаблоны, не зная что это шаблон.

Гексагональная архитектура это так же шаблон.

В рамках этой беседы шаблоны рассматривать не будем. Просто необходимо знать что они есть.

Пример

Задача:

По внешнему тактовому сигналу (inStrob = 1) считать значение с портов A, B, C. Вычислить сумму и подсветить соответствующий светодиод.

№	Значение A+B+C	Отображение
1	0..50	Горит зеленый
2	51..150	Горит желтый
3	>150	Горит красный

Светодиод зажигается 1 в советующем бите

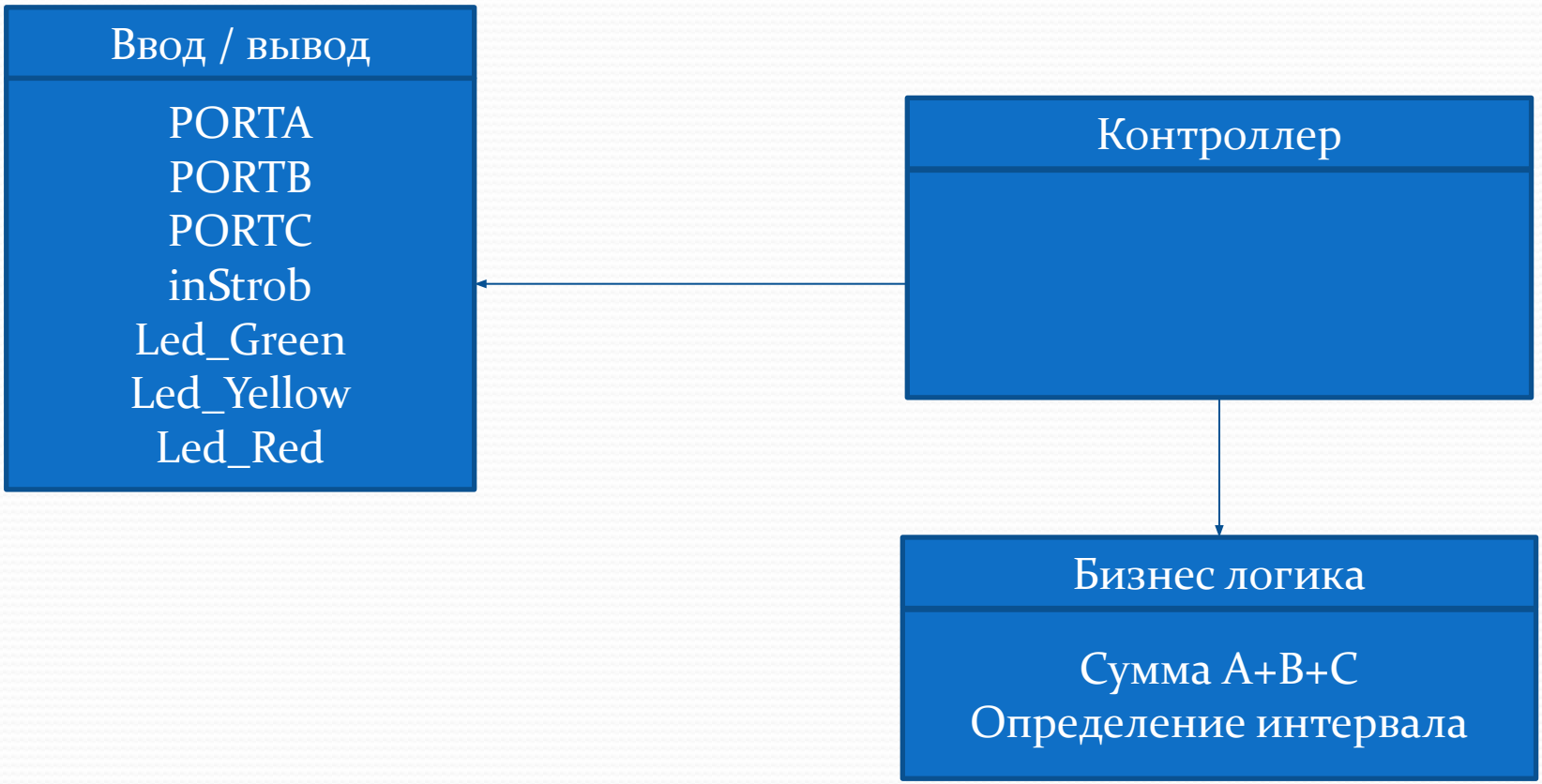
Led_Green, Led_Yellow, Led_Red

Пример решение1

Код	Комментарии
void Handle_ABC(){	
if ((inStrob == 1) && (inStrob_mem == 0)){	// отслеживаем строб / 1
Led_Green = 0;	//Гасим индикацию /2
Led_Yellow = 0;	
Led_Red = 0;	
if (PORTA+PORTB+PORTC < 51){	//Вычисляем сумму, проверяем условие /3
Led_Green = 1;	Зажигаем зеленый /4
}else{	
if (PORTA+PORTB+PORTC < 151){	//Вычисляем сумму, проверяем условие
Led_Yellow = 1;	//Зажигаем желтый
}else{	
Led_Red = 1;	// зажигаем красный
}	
}	
}	
inStrob_mem = inStrob;	//запоминаем предыдущее значение
}	строба /5

Функция Handle_ABC() выполняет 5 действий

Переход к гексагональной архитектуре



Модуль ввода /вывода (hal_un.c)

unsigned char hal_GetStrob (){
unsigned char res;
res = 0;
if (inStrob == 1){
res = 1;
}
return res;
}
	Запрос строба

unsigned char hal_getA (){
return PORTA;
}
	Запрос порта А

unsigned char hal_getB (){
return PORTB;
}
	Запрос порта В

unsigned char hal_getC (){
return PORTC;
}
	Запрос порта С

void hal_HideAll(){
Led_Green = 0;
Led_Yellow = 0;
Led_Red = 0;
}
	Тушит все светодиоды

void hal_Show_GR(){
Led_Green = 1;
}
	Зажигает зеленый

void hal_Show_YL(){
Led_Yellow = 1;
}
	Зажигает желтый

void hal_Show_RD(){
Led_Red = 1;
}
	Зажигает красный

Бизнес логика (logic.c)

```
unsigned char Logic(unsigned char A, unsigned char B,
unsigned char C){
   unsigned short Sum;
   unsigned char Res;
   Sum = LogicCalc(A,B,C);
   Res =  LogicCheckInterval (Sum);
   return Res;
}


unsigned short LogicCalc (unsigned char A, unsigned
char B, unsigned char C){
   return A+B+C;
}
unsigned char  LogicCheckInterval (unsigned short
Sum){
   unsigned char res;
   res = 0;
   if (Sum < 51){
      res = 0;
   }else{
      if (Sum < 151){
         res = 1;
      }else {
         res = 2;
      }
   }
   return res;
}
```

Функция реализует бизнесс логику
На входе 3 значения, на выходе код светодиода
0- Зеленый
1- Желтый
2- Красный

Вычисляет сумму

Определяет какой светодиод зажигать.

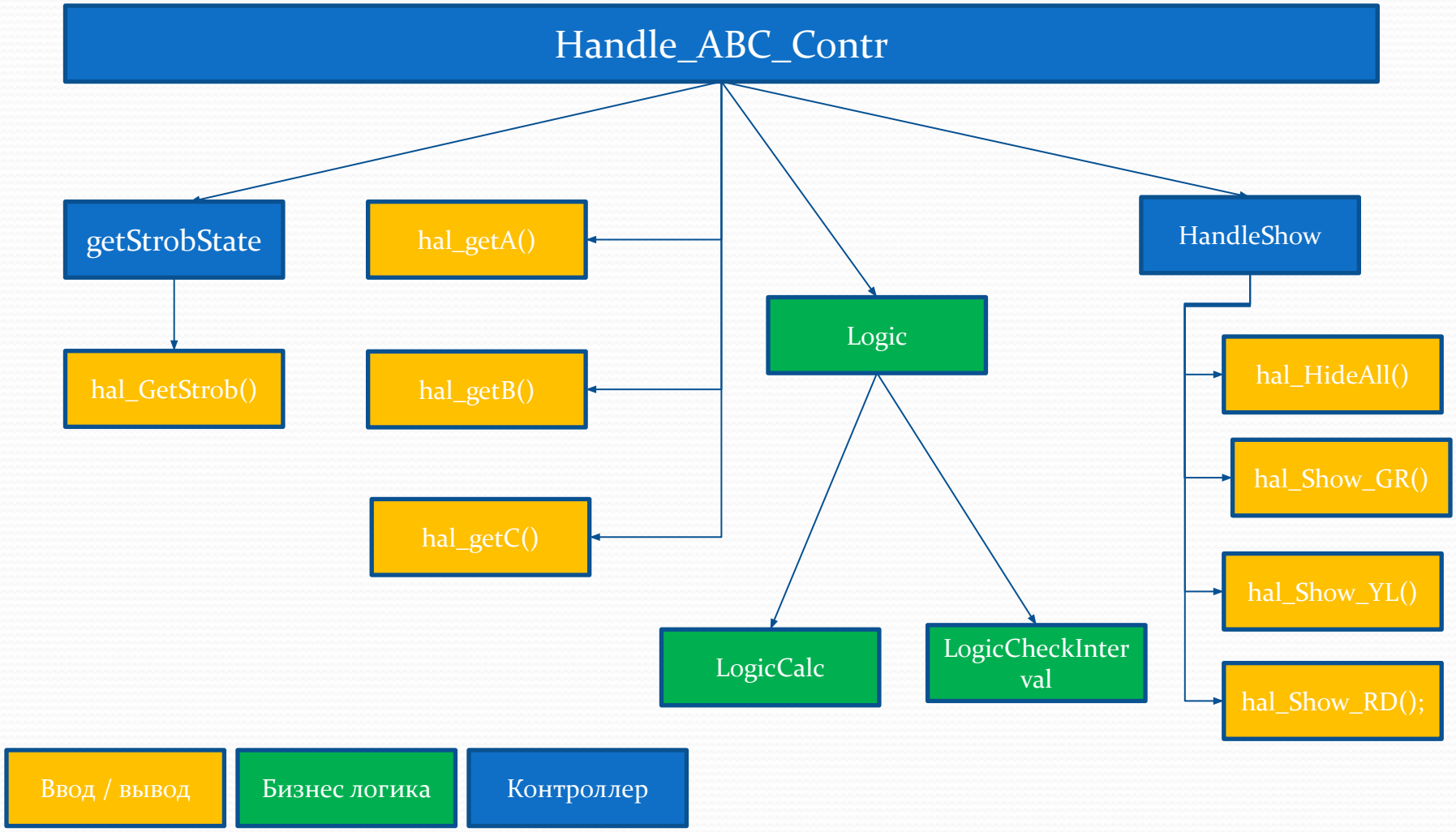
Контроллер

void Hande_ABC_Contr(){
unsigned char A,B,C,LogicRes;
if (getStrobState()){
A = hal_getA();
B = hal_getB();
C = hal_getC();
LogicRes = Logic(A,B,C);
HandleShow(LogicRes);
}
}
	Основной поток управления

unsigned char getStrobState(){
unsigned char res;
res = 0;
if ((hal_GetStrob() == 1) && (inStrob_mem ==0)){
res = 1;
}
inStrob_mem = hal_GetStrob();
return res;
}
	Отслеживание строба

void HandleShow (unsigned char LR){
hal_HideAll();
switch (LR){
case 0: hal_Show_GR();
break;
case 1: hal_Show_YL();
break;
case 2: hal_Show_RD();
break;
}
}
	Управление индикацией

Граф вызовов функций



Устойчивость к изменениям в требованиях

Что изменится если произойдут изменения в требованиях…

- Изменения в входных / выходных интерфейсах, например A,B,C необходимо принимать по UART.

  Изменится фунция Handle_ABC_Contr (), в ней будут вызывать другие функции вместо **hal_getA(), hal_getB(), hal_getC(), hal_GetStrob().** Остальное останется неизменным.

- Изменения в алгоритме обработки входных значений, например A+B-C.

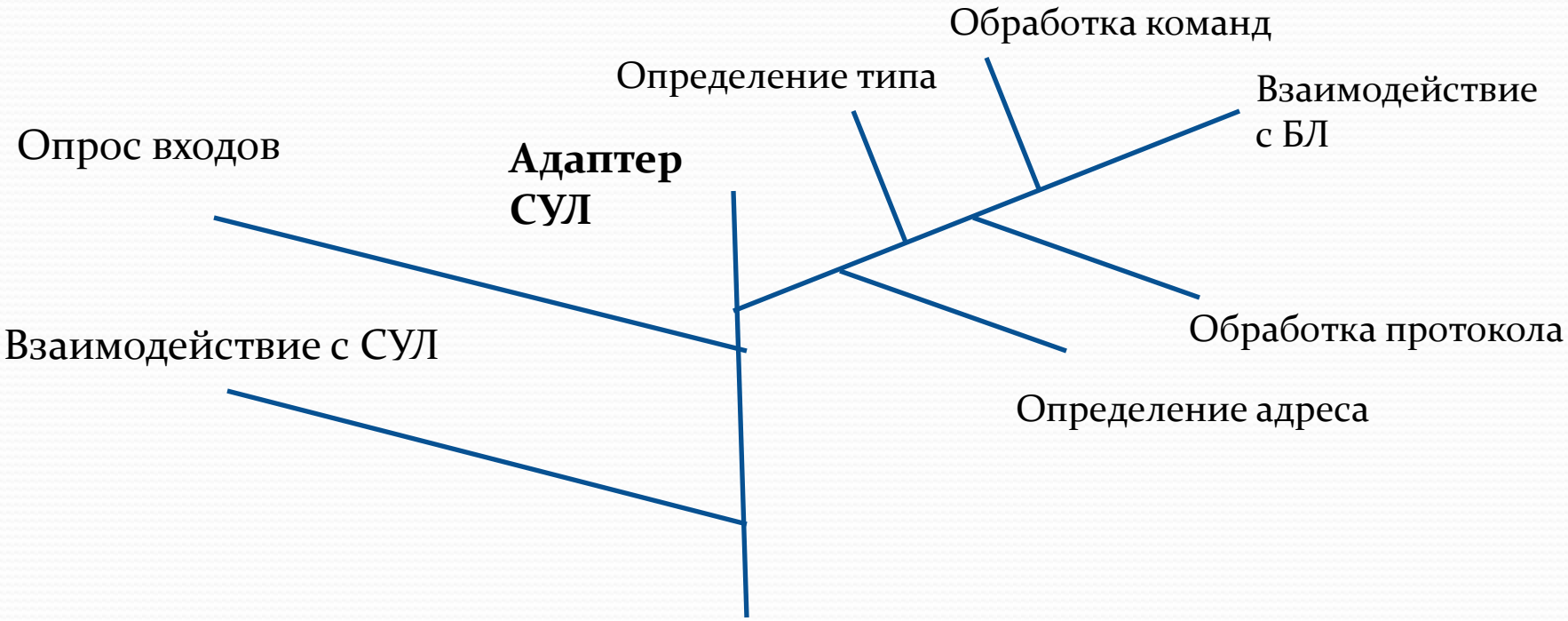
  Изменится только функция **LogicCalc()**, остальное останется неизменным.

- Изменятся пороги или правила индикации светодиодами, то изменится функция **LogicCheckInterval()**, хотя может поменяться и **HandleShow()**

Вывод: при изменении требований, меняется только те части, в отношении которых поменялись требования.

Разработка архитектуры ПО шаг 1

1. Построение дерева функционала



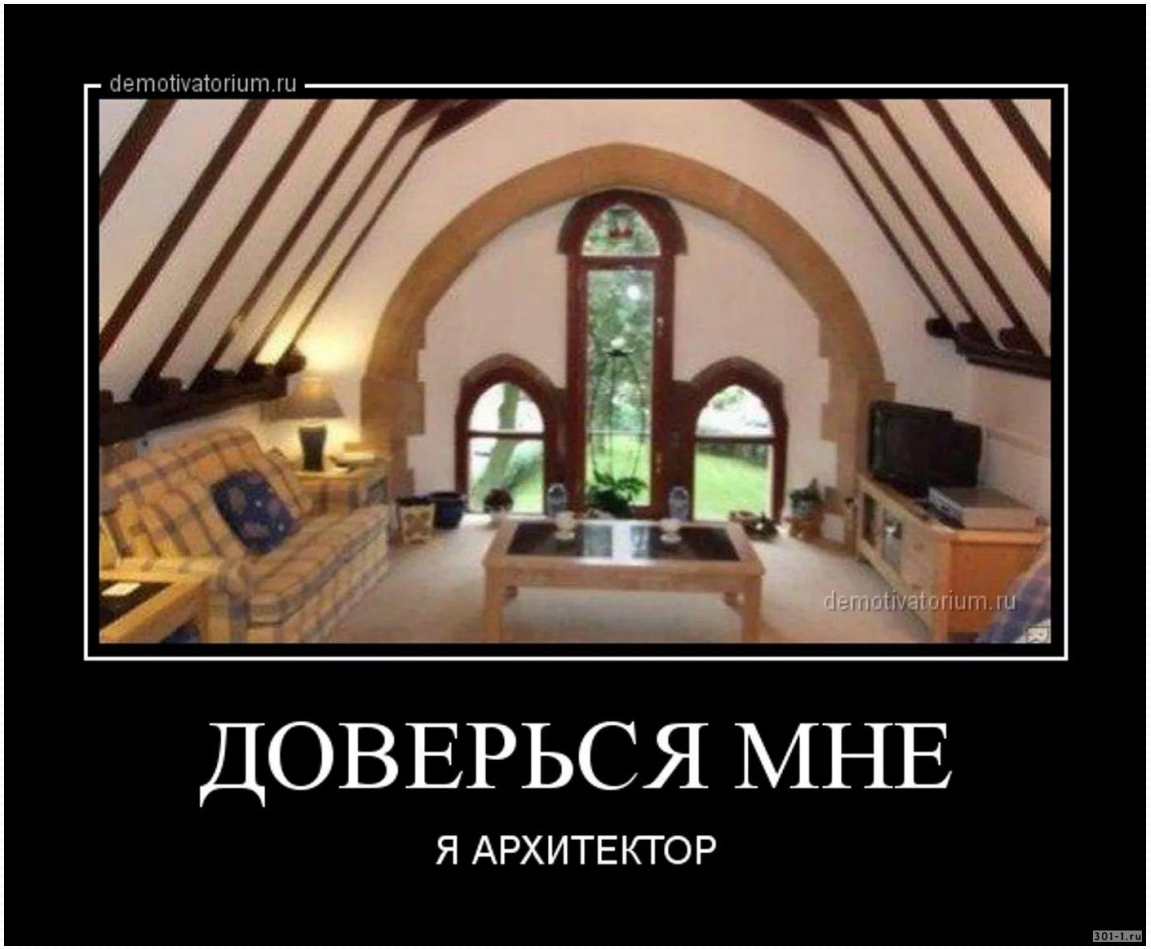
Разработка архитектуры ПО шаги 2..n

2. Укрупненное разбиение на классы (по функционалу)
3. Вводятся дополнительные классы для обеспечения реализации функционала (класс таймеров, HAL и тд)
3. Определение связей между классами (лучше всего UML модель диаграмма классов)
4. Для каждого полученного класса выделяются (лучше рисуются UML модели диаграмма классов):
    1. Интерфейсы взаимодействия
    2. Бизнес логика.
    3. Контроллер (поток управления)
5. При необходимости каждый класс разбивается на более мелкие классы (функции) в соответствии с принципом единой ответственности.

Выводы о применении гексагональной архитектуры

1. Минусы
    1. Разработка архитектуры требует затрат времени
    2. Реализация ПО в соответствии с архитектурой требует затрат времени, увеличивает объем кода, снижает быстродействие.
2. Плюсы
    1. Если придерживаться архитектуры, то ПО получается гибким, устойчивым к изменениям, появляется возможность повторного использования кода в других проектах.
    2. Архитектура окупается при дальнейшем развитии проекта
    3. Архитектура позволяет производить автоматизированное тестирование ПО, а следовательно позволяет чаще производить рефакторинг!!!!

Все!


demotivatorium.ru
demotivatorium.ru
ДОВЕРЬСЯ МНЕ
Я АРХИТЕКТОР
301-1.ru