



Проектирование и эксплуатация информационных систем в медиаиндустрии

Выломова Екатерина Алексеевна
e-mail: evylotova@gmail.com

0. Лекция 3

Парадигмы программирования:

- Декларативное vs императивное
- Процедурное, Модульное, Объектно-ориентированное, Компонентно-ориентированное, Аспектно-ориентированное

Объектно-ориентированный подход:

- Абстракция
- Инкапсуляция
- Наследование
- Полиморфизм

Стили проектирования ИС:

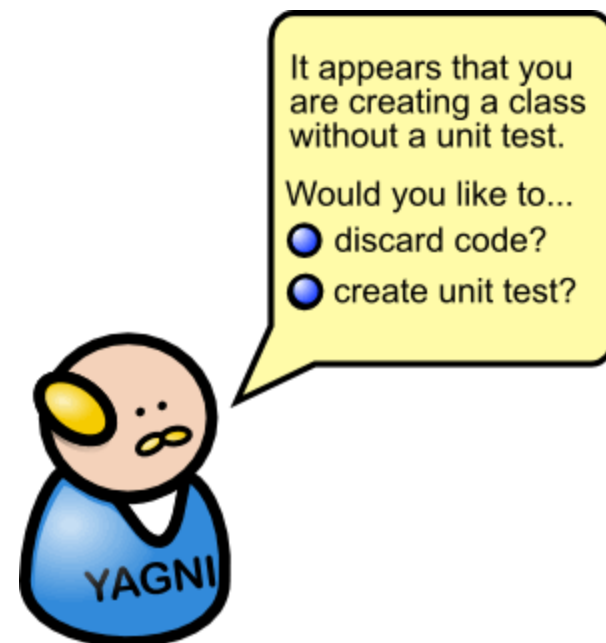
- Нисходящее, восходящее, эволюционное

Этапы проектирования ИС

0. YAGNI

You ain't gonna need it

“Вам это не понадобится”



Отказ от добавления функциональности, в которой нет
непосредственной нужды

Используется в экстремальном программировании

I. Лекция 4. Модели жизненного цикла, проектирование как конструирование

- Модели проектирования систем
- Проектирование как конструирование
- Шаблоны проектирования
- Дополнительно: сложность алгоритмов

I. Стандарты и методики

Жизненный цикл ПО – это непрерывный процесс, который начинается с момента принятия решения о необходимости его создания и заканчивается в момент его полного изъятия из эксплуатации.

- **ГОСТ 34.601-90** – стадии и этапы создания автоматизированных систем.
- **ISO/IEC 12207:1995** – организация и процессы жизненного цикла.
- **Custom Development Method** (методика Oracle) – детализация до уровня заготовок проектных документов.
- **Rational Unified Process (RUP)**- итеративная модель разработки из 4-х фаз (начало, исследование, проектирование, внедрение).
- **Microsoft Solution Framework (MSF)** – аналогична RUP.
- **Extreme Programming (XP)** –методика на основе командной работы, коммуникаций с заказчиком и быстрого прототипирования.

I. ISO / IEC 12207

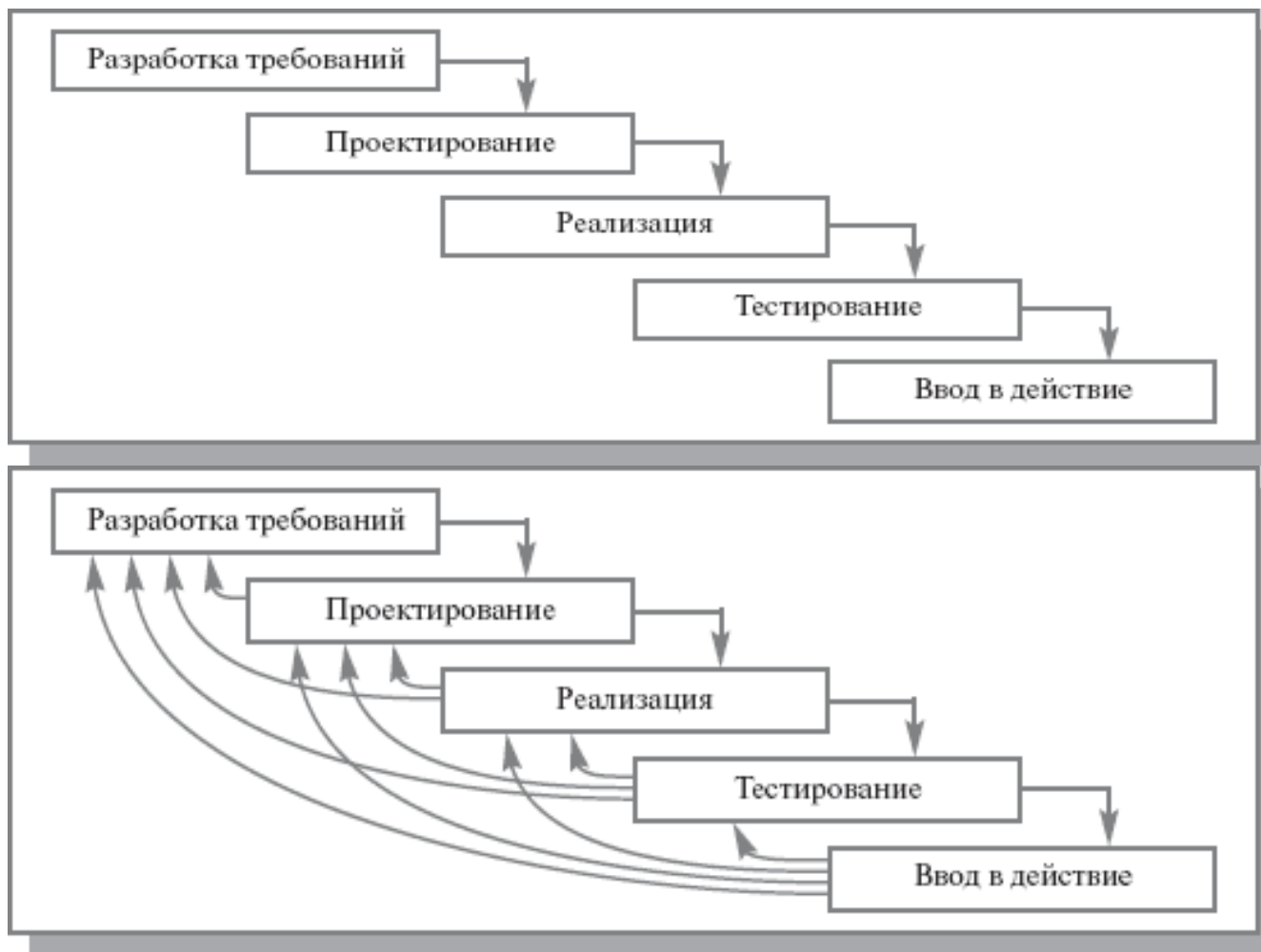
ISO/IEC 12207 определяет структуру ЖЦ, содержащую процессы, действия и задачи, которые должны быть выполнены во время создания ПО.

ISO – International Organization of Standardization – Международная организация по стандартизации

IEC – International Electrotechnical Commission – Международная комиссия по электротехнике

- **Основные процессы:** приобретение, поставка, разработка, эксплуатация, сопровождение.
- **Вспомогательные процессы:** документирование, управление конфигурацией, обеспечение качества, разрешение проблем, аудит, совместная оценка, верификация.
- **Организационные процессы:** создание инфраструктуры, управление, обучение, усовершенствование.

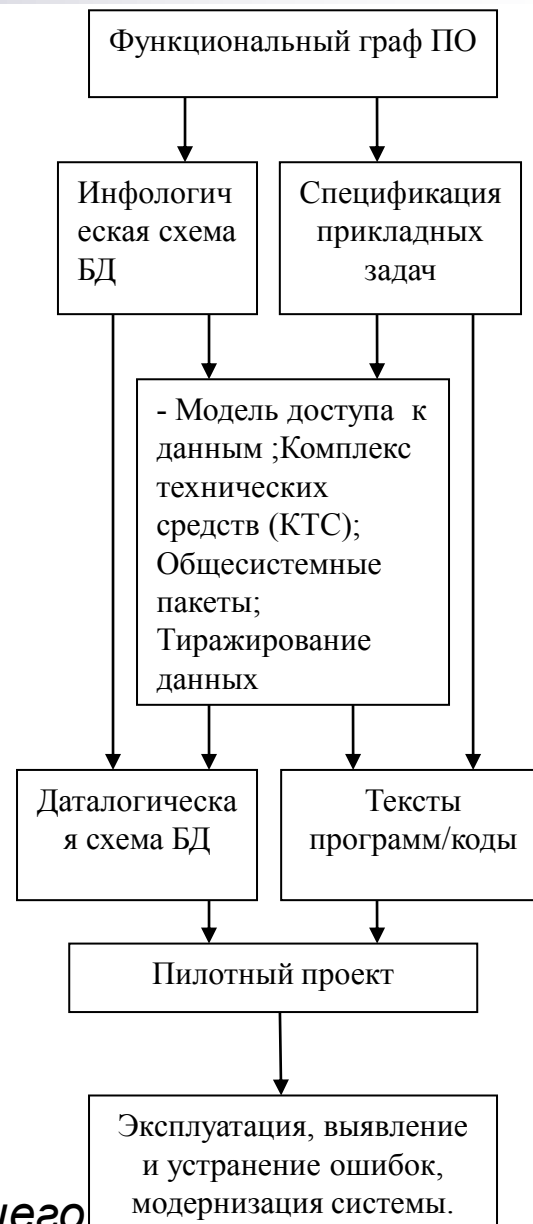
I. Каскадная модель



I. Каскадная модель

1. Выявление информационных потребностей конечных пользователей (предпроектное обследование, разработка ТЗ, частных ТЗ)
2. Концептуальный проект (эскизное проектирование)
3. Архитектура системы (технический проект)
4. Логическое проектирование (техническое проектирование)
5. Комплексная отладка (разработка рабочей документации)
6. Сопровождение

Каждый этап – после предыдущего



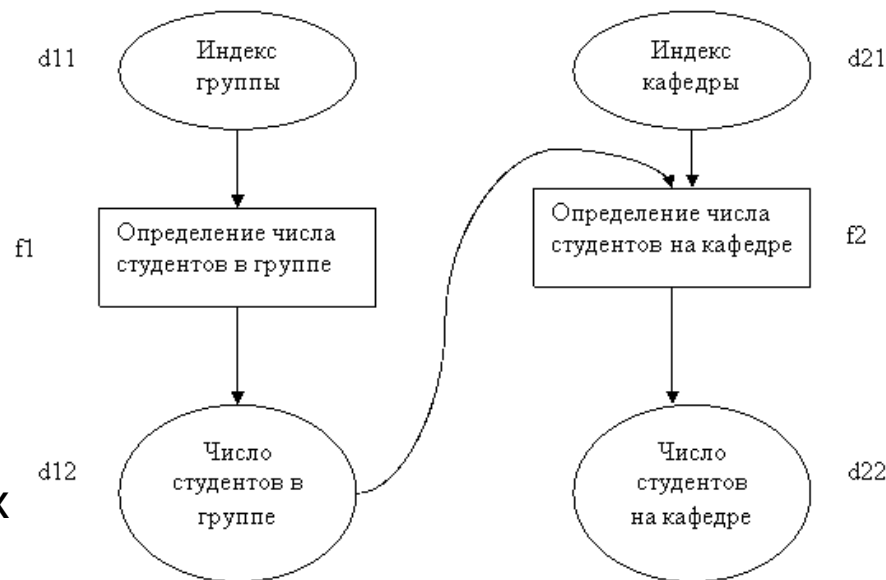
I. Функциональный граф ПО

Функциональный граф ПО - граф, узлы которого обозначают данные и процессы будущей системы. Дуги используются для обозначения входных/выходных данных для процесса.

$$d22 = f2(d12, d21) = f2(f1(d11), d21)$$

Процессы и данные
объединены!

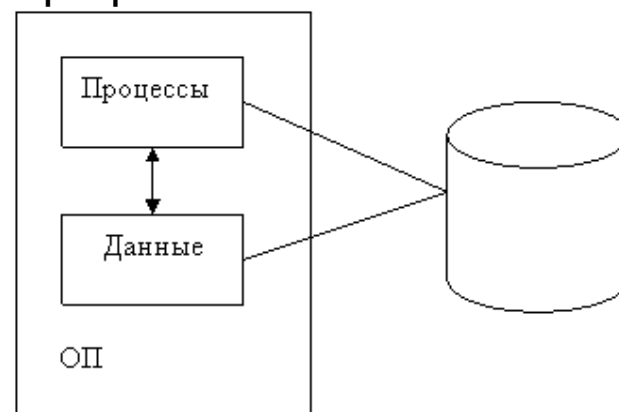
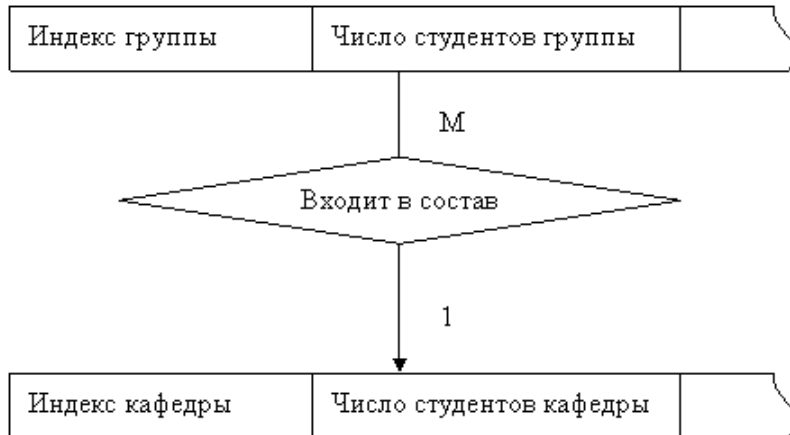
Машина Фон-Неймана:
Разделение процессов и данных



I. Концептуальный проект

- Выделение данных из функционального графа.
- Проектирование схемы БД

*Пример: инфологическая схема БД в
Нотации Чена*



*Спецификация процессов -
входные и выходные данные
процессов, а также
алгоритмическая связь между
ними.*

Концептуальный проект не зависит от архитектуры!

Модели жизненного цикла ПО

I. Выбор архитектуры

- Выбор модели доступа к данным (файл-сервер, сервер-БД, сервер-приложение, доступ к данным по Internet/Intarnet)
- Выбор комплекса технических средств (выбор «железа»)
- Выбор общесистемных пакетов
- Выбор способа тиражирования данных

I. Логическое проектирование

Отражение концептуального проекта в СУБД-ориентированную среду с помощью выбранных оболочек программирования



- Сущности --> таблицы
- Спецификации --> тексты программ



I. Отладка и сопровождение

Отладка:

Результаты проектирования БД и приложений объединяются. В итоге разрабатывается пилотный проект системы.

Сопровождение:

Выявление ошибок и их устранение, модернизация.

I. Достоинства и недостатки каскадной модели

Достоинства:

- Проста, естественна, имеет некоторую привязку к ГОСТу

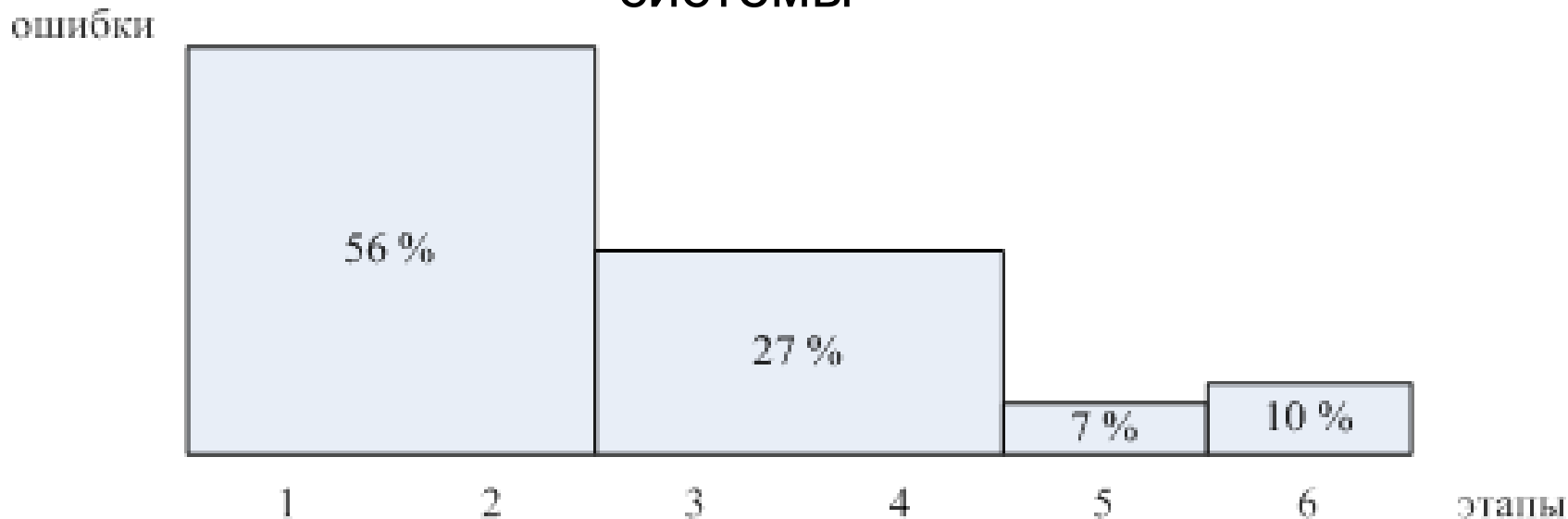
Недостатки:

- достаточно продолжительный цикл разработки по времени (система морально устаревает)
- доработка системы связана с большим объемом перепрограммирования (из-за слабого использования CASE-средств)
- Устаревание модели

I. Результаты исследований

Д.Мартина (80-е)

Распределение ошибок и просчетов по этапам проектирования, выявленных при сопровождении системы

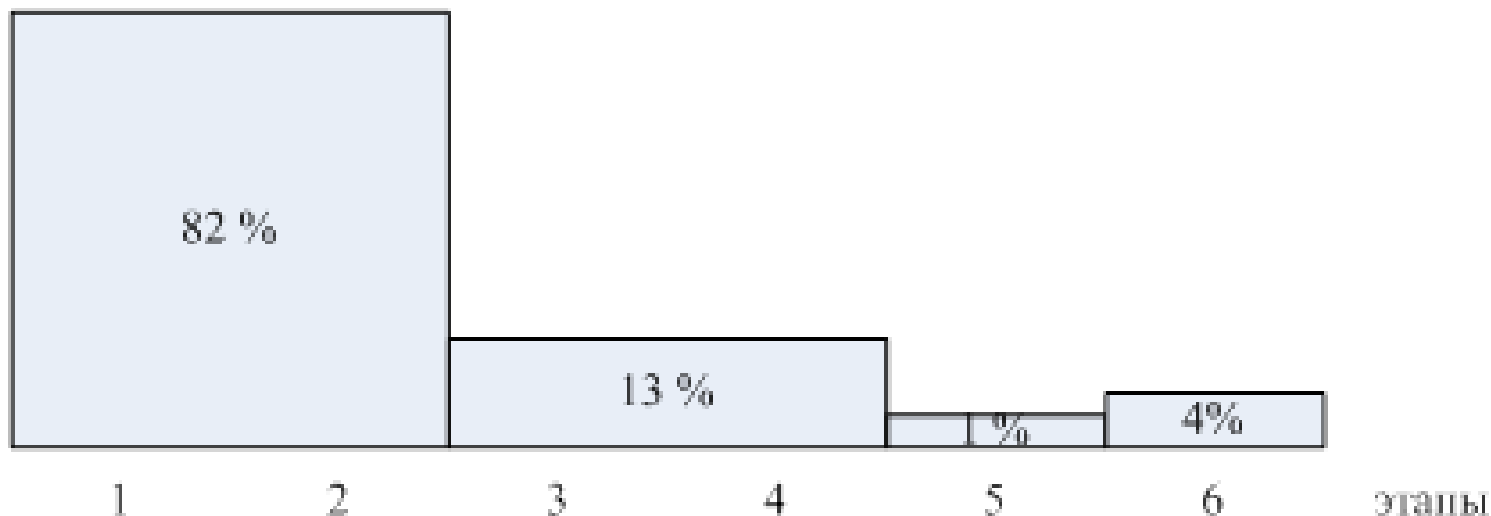


I. Результаты исследований

Д.Мартина (80-е)

Распределение затрат на исправление ошибок и просчетов, выявленных при сопровождении

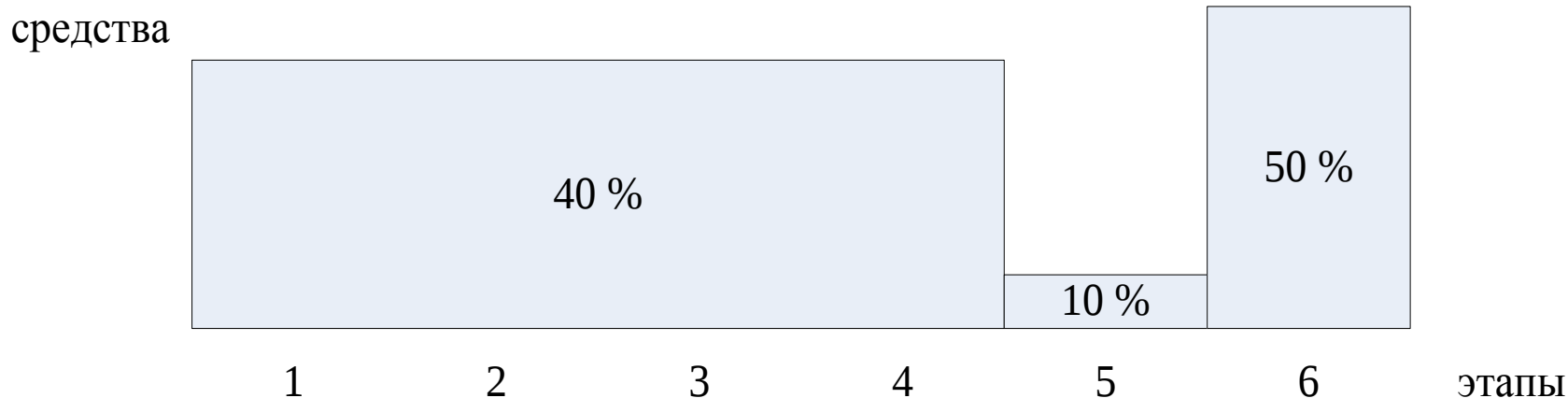
средства



I. Результаты исследований

Д.Мартина (80-е)

Распределение трудозатрат по этапам проектирования



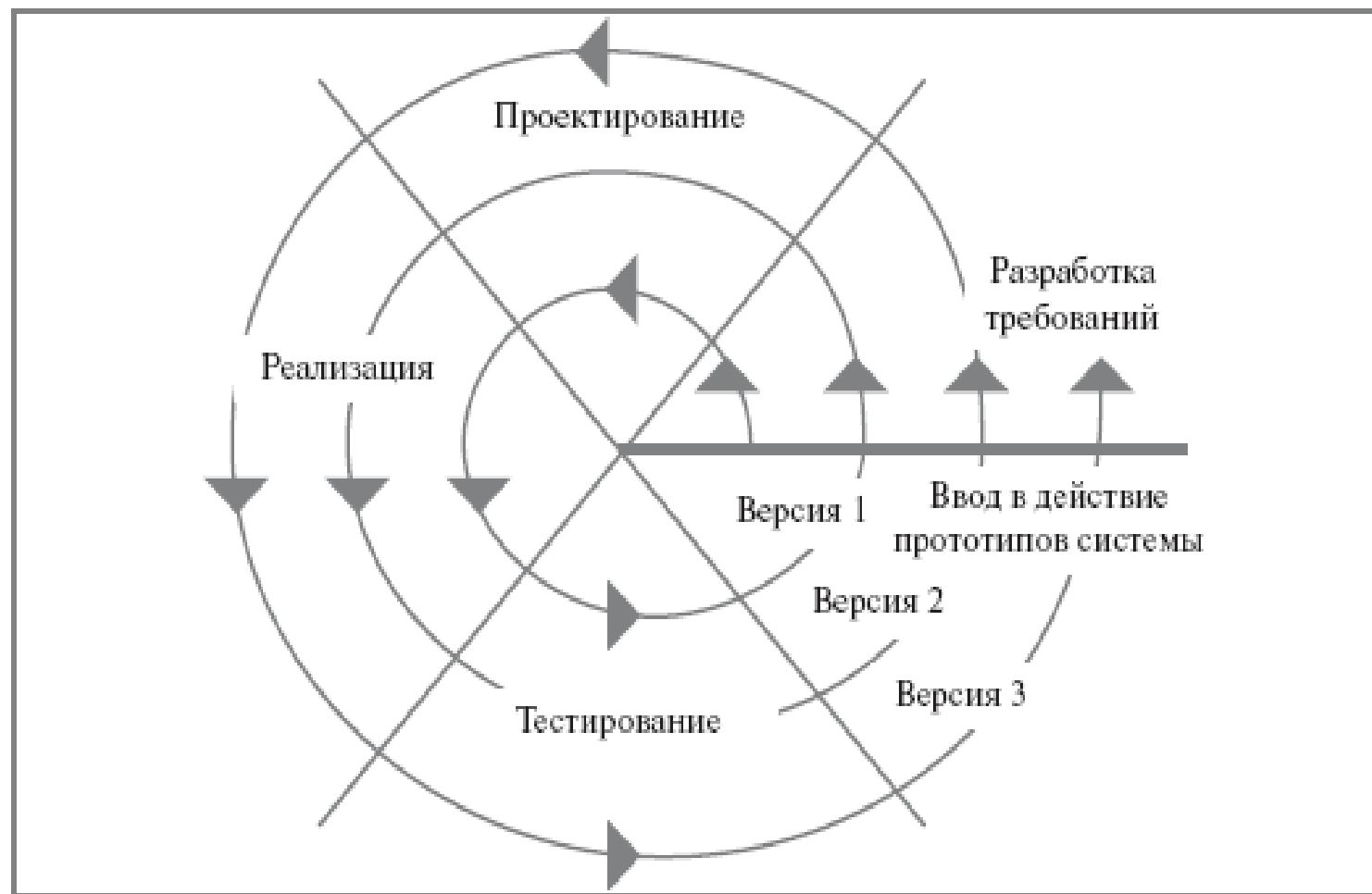
Почти половина трудозатрат приходится на устранение ошибок, допущенных на первых 2-х этапах.

Модели жизненного цикла ПО

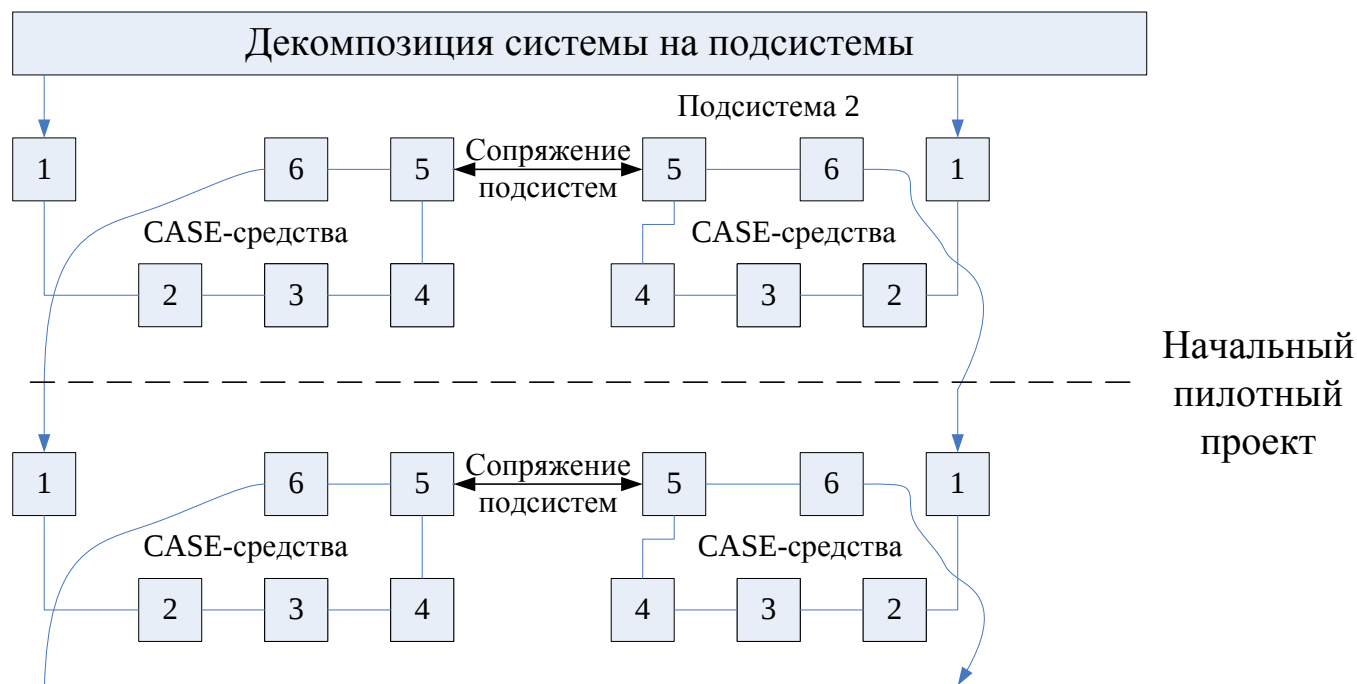
I. Законы Мартина

1. **Закон неопределенности в информатике:**
процесс автоматизации задачи меняет представление пользователя об этой задаче, т.е. пользователь решает задачу с использованием средств автоматизации иначе, чем без них.
2. Чем больше времени прошло с момента совершения ошибки до момента ее обнаружения, тем больше средств необходимо для ее устранения (диагр. 2).
3. Программисты и проектировщики не учатся на чужих ошибках, а только на своих.

I. Спиральная модель



I. Спиральная модель



1. выбор архитектуры подсистем
2. выявление информационных потребностей конечных пользователей системы
3. концептуальное проектирование
4. логическое проектирование
5. отладка подсистем
6. сопровождение подсистемы

II. Компонентно-ориентированный ПОДХОД

Компонентно-ориентированный подход: разработка компонентов, предназначенных для многократного использования.

Компонент – независимый программный модуль, предназначенный для повторного использования и развертывания.

Интероперабельность программного обеспечения (функциональность программного обеспечения) - способность программного продукта выполнять набор функций, определенных в его внешнем описании и удовлетворяющих заданным или подразумеваемым потребностям пользователей.

II. Аспекты описания ИС

Аспект описания (страта) — описание системы или ее части с некоторой оговоренной точки зрения, определяемой функциональными, физическими или иного типа отношениями между свойствами и элементами.

Основные аспекты описания:

- Функциональный (функции ИС)
- Информационный (описание ПО, существ. объектов и их возможных значений)
- Структурный (морфология ИС)
- Поведенческий (алгоритмы, технологические процессы)

Дополнительные (возможные) аспекты:

- Конструкторский (форма и расположение компонентов ИС)
- Алгоритмический (алгоритмы и ПО)
- Технологический (технологические процессы)

III. Шаблоны проектирования

Преимущества
стандартизации
проектных
решений

Каталог решений способствует обучению

Формализованное описание решений

Общая терминология

Конец 70-х: Кристофер Александер – шаблоны проектных решений в архитектуре (не ПО).
1995: GoF: Design Patterns: Elements of Reusable Object-Oriented Software.

Типы

- Производящие шаблоны
- Поведенческие шаблоны
- Структурные шаблоны
- Системные шаблоны

Уровень

- Отдельный класс
- Компонент
- Архитектурный

<i>Метод</i>	<i>Повторное использование</i>	<i>Абстракция</i>	<i>Универсальность подхода</i>
Копирование и вставка	Очень плохо	Отсутствует	Очень плохо
Структуры данных	Хорошо	Тип данных	Средне — хорошо
Функциональность	Хорошо	Метод	Средне — хорошо
Типовые блоки кода	Хорошо	Типизируемая операция	Хорошо
Алгоритмы	Хорошо	Формула	Хорошо
Классы	Хорошо	Данные + метод	Хорошо
Библиотеки	Хорошо	Функции	Хорошо — очень хорошо
API	Хорошо	Классы утилит	Хорошо — очень хорошо
Компоненты	Хорошо	Группы классов	Хорошо — очень хорошо
Шаблоны проектирования	Отлично	Решения проблем	Очень хорошо

Шаблоны проектирования

III. Основные типы шаблонов

Причины использования производящих шаблонов

- Единый способ создания объектов
- Упрощение
- Учет ограничений

Основные представители производящих шаблонов

- Abstract Factory
- Builder
- Factory Method
- Prototype
- Singleton

Основные представители структурных шаблонов

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Half-Object Plus Protocol (HOPP)
- Proxy

Основные представители поведенческих шаблонов

- Chain of Responsibility
- Command
- Interpreter
- **Iterator**
- Mediator
- Memento
- **Observer**
- State
- Strategy
- Visitor
- Template Method

III. Пример: абстрактная фабрика

Порождающий шаблон проектирования, позволяющий изменять поведение системы, варьируя создаваемые объекты, при этом сохраняя интерфейсы.

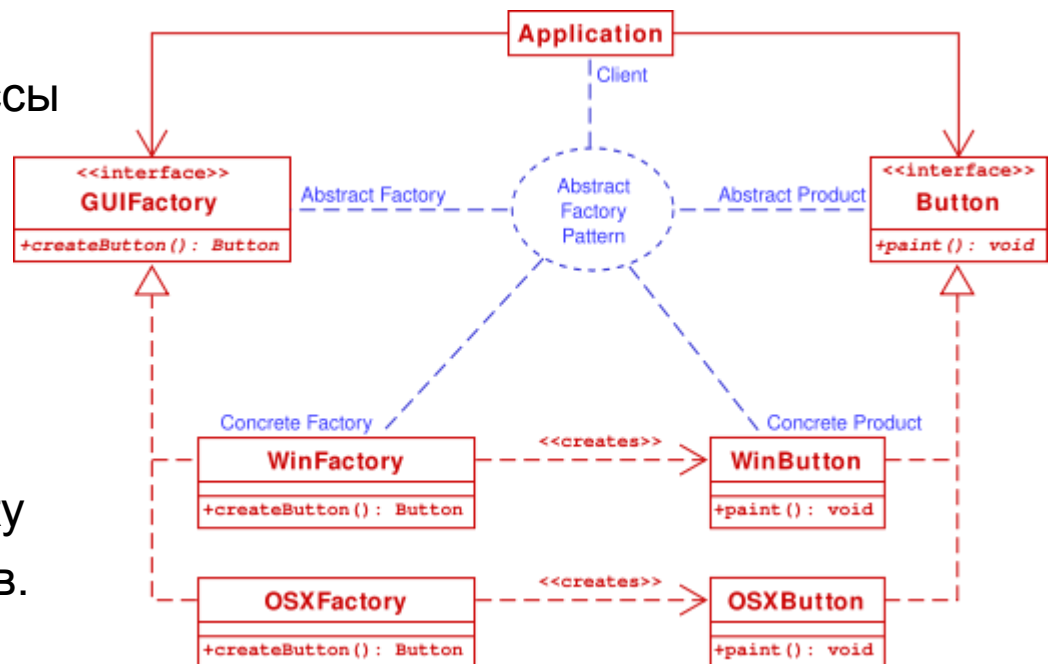
Предоставляет интерфейс для создания семейств взаимосвязанных или взаимозависимых объектов, не специфицируя их конкретных классов.

Плюсы:

- изолирует конкретные классы
- упрощает замену семейств продуктов
- гарантирует сочетаемость продуктов

Минусы:

- сложно добавить поддержку нового вида продуктов.



III. Пример: абстрактная фабрика

```
/* GUIFactory example -- */
#include<iostream>
using namespace std;
class Button { public: virtual void paint() = 0; };
class WinButton: public Button {
public: void paint() { cout << "I'm a WinButton"; }
};
class OSXButton: public Button {
public: void paint() { cout << "I'm an OSXButton"; }
};
class GUIFactory {
public: virtual Button * createButton() = 0; };
class WinFactory: public GUIFactory {
public: Button * createButton()
{ return new WinButton(); }
};
```

```
class OSXFactory: public GUIFactory {
public: Button * createButton()
{ return new OSXButton(); } };
class Application {
public: Application(GUIFactory * factory)
{ Button * button = factory->createButton();
button->paint(); } };
GUIFactory * createOsSpecificFactory()
{ int sys; cout << endl << "Enter OS Type(0 - Win,
1 - OSX): ";
cin >> sys;
if (sys == 0) { return new WinFactory(); }
else { return new OSXFactory(); } }
int main(int argc, char **argv) {
Application * newApplication = new
Application(createOsSpecificFactory()); return 0;
}
```

III. Пример: синглтон

Объект-одиночка

Гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.

Плюсы:

- контролируемый доступ к единственному экземпляру
- уменьшение числа имён
- допускает уточнение операций и представления
- допускает переменное число экземпляров
- бóльшая гибкость, чем у операций класса

Минусы:

- Глобальные объекты могут быть вредны для объектного программирования, в некоторых случаях приводя к созданию немасштабируемого проекта.
- Усложняет написание модульных тестов

Singleton	
-	<u>singleton : Singleton</u>
-	Singleton()
+	<u>getInstance() : Singleton</u>

III. Пример: синглтон

/// Class implements singleton pattern.

public class Singleton {

*// Protected constructor is sufficient to avoid other
///instantiation*

*// This must be present otherwise the compiler
///provides a default public constructor*

protected Singleton() { }

/// Return an instance of Singleton

public static Singleton Instance { get {

*/// An instance of Singleton wont be created until
///the very first call to the sealed class. This is a
///CLR optimization that provides a properly lazy-
///loading singleton.*

return SingletonCreator.CreatorInstance; } }

*/// Sealed class to avoid any heritage from this
///helper class*

private sealed class SingletonCreator {

// Retrieve a single instance of a Singleton

private static readonly Singleton _instance =
new Singleton();

*//explicit static constructor to disable before field
///init*

static SingletonCreator() { }

/// Return an instance of the class Singleton

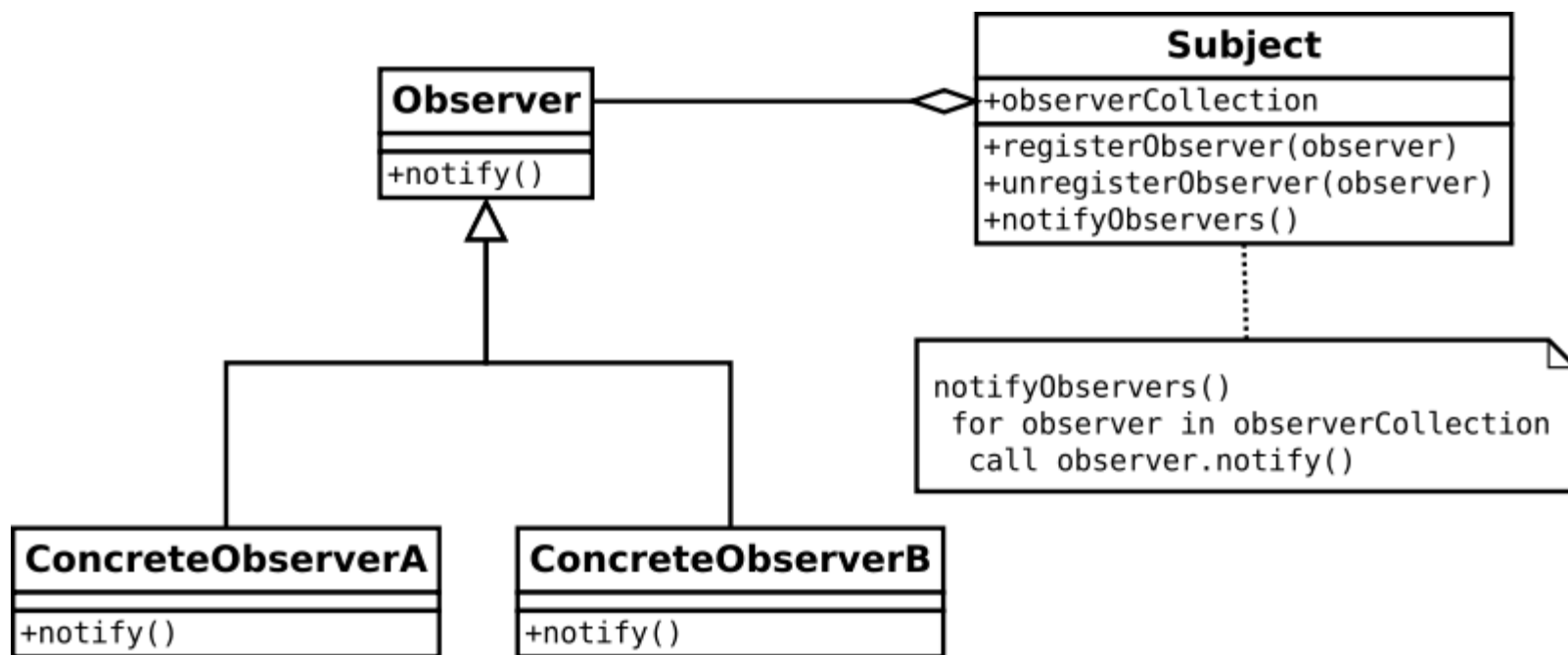
public static Singleton CreatorInstance { get {
return _instance; } }

}

}

III. Пример: наблюдатель

Поведенческий шаблон. Определяет зависимость типа «один ко многим» между объектами таким образом, что при изменении состояния одного объекта все зависящие от него оповещаются об этом событии.



III. Пример: наблюдатель

```
class AbstractSubject:
def register(self, listener):
raise NotImplementedError("Must subclass me")
def unregister(self, listener):
raise NotImplementedError("Must subclass me")
def notify_listeners(self, event):
raise NotImplementedError("Must subclass me")
class Listener:
def __init__(self, name, subject): self.name = name
    subject.register(self)
def notify(self, event):
print self.name, "received event", event
class Subject(AbstractSubject):
def __init__(self):
    self.listeners = [] self.data = None
def getUserAction(self):
    self.data = raw_input('Enter something to do:') return
    self.data
# Implement abstract Class AbstractSubject
def register(self, listener):
    self.listeners.append(listener)
```

```
def unregister(self, listener):
    self.listeners.remove(listener)
def notify_listeners(self, event):
for listener in self.listeners:
    listener.notify(event)
if __name__=="__main__":
# make a subject object to spy on
    subject = Subject()
# register two listeners to monitor it.
    listenerA = Listener("<listener A>", subject)
    listenerB = Listener("<listener B>", subject)
# simulated event
    subject.notify_listeners ("<event 1>")
# outputs:
# <listener A> received event <event 1>
# <listener B> received event <event 1>
    action = subject.getUserAction()
    subject.notify_listeners(action)
#Enter something to do:hello
# outputs: # <listener A> received event hello
# <listener B> received event hello
```

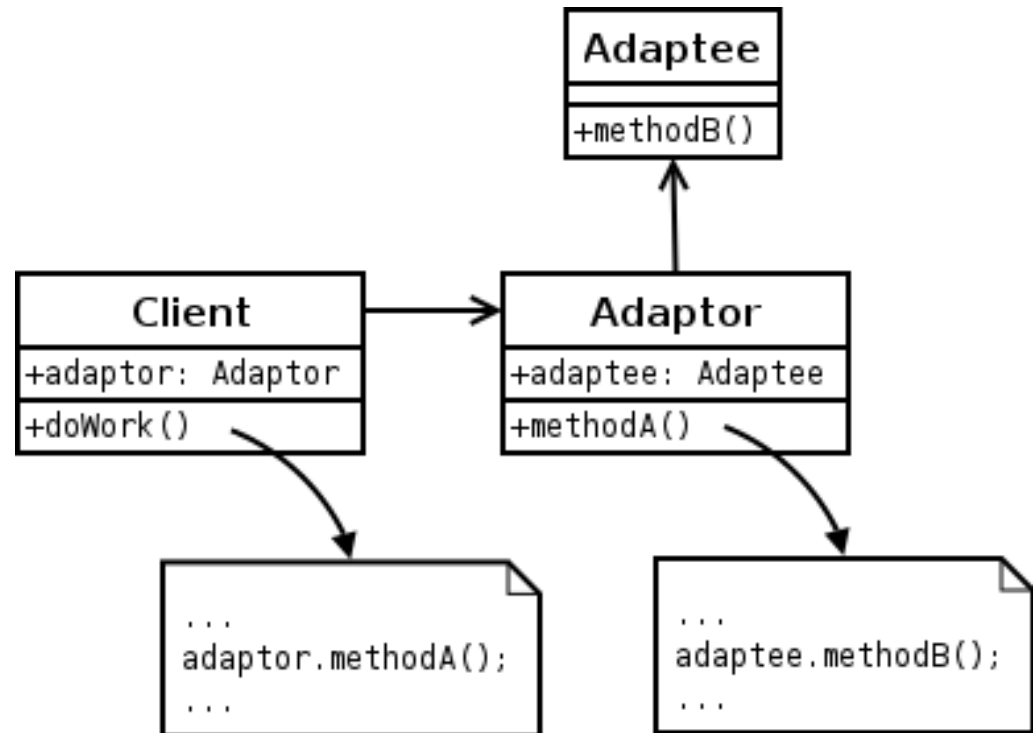
III. Пример: адаптер

Структурный шаблон, предназначенный для организации использования функций объекта, недоступного для модификации, через специально созданный интерфейс.

Задача: Система поддерживает требуемые данные и поведение, но имеет неподходящий интерфейс.

Решение:

Адаптер предусматривает создание класса-оболочки с требуемым интерфейсом



III. Пример: адаптер

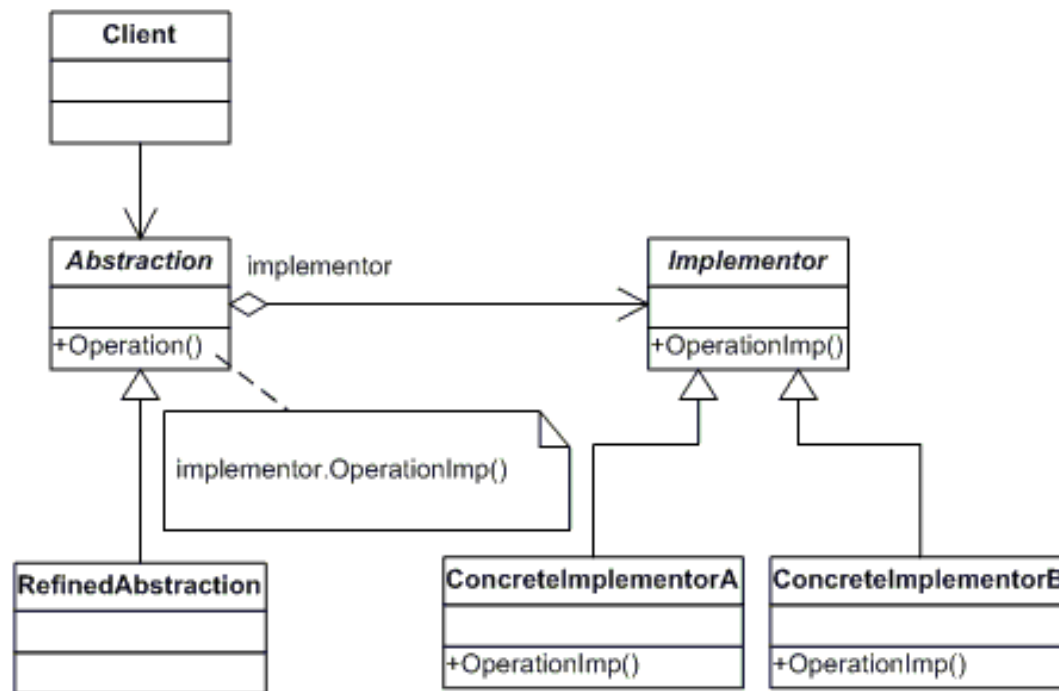
```
class Adaptee:
    def specific_request(self):
        return 'Adaptee'

class Adapter:
    def __init__(self, adaptee):
        self.adaptee = adaptee
    def request(self):
        return self.adaptee.specific_request()

client = Adapter(Adaptee())
print client.request()
```

III. Пример: мост

Структурный шаблон, предназначенный для того, чтобы «разделять абстракцию и реализацию так, чтобы они могли изменяться независимо».



III. Пример: мост

Implementor

class DrawingAPI:

def drawCircle(x, y, radius): **pass**

ConcreteImplementor 1/2

class DrawingAPI1(DrawingAPI):

def drawCircle(self, x, y, radius):

print "API1.circle at %f:%f radius %f" % (x, y, radius)

ConcreteImplementor 2/2

class DrawingAPI2(DrawingAPI):

def drawCircle(self, x, y, radius):

print "API2.circle at %f:%f radius %f" % (x, y, radius)

Abstraction

class Shape:

low-level

def draw(self):

pass

high-level

def resizeByPercentage(self, pct):

pass

Refined Abstraction

class CircleShape(Shape):

def __init__(self, x, y, radius, drawingAPI):

self.__x = x self.__y = y self.__radius = radius

self.__drawingAPI = drawingAPI

low-level i.e. Implementation specific

def draw(self):

self.__drawingAPI.drawCircle(self.__x, self.__y,
self.__radius)

high-level i.e. Abstraction specific

def resizeByPercentage(self, pct):

self.__radius *= pct

def main(): shapes =

[CircleShape(1, 2, 3, DrawingAPI1()), CircleShape(5,
7, 11, DrawingAPI2())]

for shape **in** shapes:

shape.resizeByPercentage(2.5)

shape.draw()

if __name__ == "__main__": main()

IV. Сложность алгоритмов

Теория алгоритмов:

- Асимптотический анализ сложности алгоритмов
- Доказательство асимптотической неразрешимости алгоритмов
- Классификация алгоритмов в соответствии с классами сложности

Теория сложности вычислений:

- оценка стоимости работы, требуемой для решения вычислительной проблемы

IV. Сложность алгоритмов

Основные абстракции:

- **Время (вычислительная сложность)** – количество элементарных шагов, необходимых для решения задачи.

Оценка худшего, среднего и лучшего времени.

Пример: Поиск информации по БД-> наихудший случай будет тогда, когда информация в БД отсутствует.

- **Пространство** - объем памяти или места на носителе данных.

Не учитывается место, которое занимает исходный массив и независимые от входной последовательности затраты, например, на хранение кода программы.

Дополнительно

IV. Сложность алгоритмов

Обозначение	Описание	Определение
$f(n) \in O(g(n))$	асимптотическая верхняя граница	$\exists(C > 0), n_0 : \forall(n > n_0) f(n) \leq Cg(n)$
$f(n) \in \Omega(g(n))$	асимптотическая нижняя граница	$\exists(C > 0), n_0 : \forall(n > n_0) Cg(n) \leq f(n) $
$f(n) \in \Theta(g(n))$	Асимптотические верхняя и нижняя границы	$\exists(C, C' > 0), n_0 : \forall(n > n_0) Cg(n) < f(n) < C'g(n) $
$f(n) \in o(g(n))$	g доминирует над f асимптотически	$\forall(C > 0), \exists n_0 : \forall(n > n_0) f(n) < Cg(n) $
$f(n) \in \omega(g(n))$	f доминирует над g асимптотически	$\forall(C > 0), \exists n_0 : \forall(n > n_0) Cg(n) < f(n) $
$f(n) \sim g(n)$	f эквивалентна g асимптотически	$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 1$

Дополнительно

IV. Сложность алгоритмов

Примеры:

«Пропылесосить ковер» - линейная зависимость от площади ковра $\Theta(A)$

«найти имя в телефонной книге» - $O(\log_2(n))$ (если фамилии отсортированы по алфавиту)

Дополнительно

IV. Сложность алгоритмов

Название	Сложность	Пример	Пример алгоритмов
Константное время	$O(1)$	10	Определение четности числа Взятие элемента массива/хэш
Логарифмическое время	$O(\log n)$	$\log n$, $\log n^2$	Бинарный поиск
Линейное время	$O(n)$	n	Поиск наименьшего значения в неотсортированном массиве
Linearithmic time	$O(n \log n)$		Наиболее быстрая сортировка сравнением (quicksort и др)
Квадратичное время	$O(n^2)$	n^2	Сортировка пузырьком, сортировка вставками
Кубическое время	$O(n^3)$	n^3	Перемножение матриц $n \times n$ (без спец. алгоритмов)
Экспоненциальное время	$2^{O(n)}$	1.1^n , 10^n	Решение задачи о коммивояжере, динамическое программирование

Дополнительно

IV. Сортировка пузырьком

Временная сложность $O(n^2)$:

```
def swap(arr, i, j):  
    arr[i], arr[j] = arr[j], arr[i]  
  
def bubble_sort(arr):  
    i = len(arr)  
    while i > 1:  
        for j in xrange(i - 1):  
            if arr[j] > arr[j + 1]:  
                swap(arr, j, j + 1)  
        i -= 1
```

Дополнительно

IV. Сортировка слиянием

Временная сложность $O(n \log n)$:

```
def mergesort(w):
```

```
    """Sort list w and return it."""
```

```
        if len(w)<2:
```

```
            return w
```

```
        else: mid=len(w)//2
```

```
    # sort the two halves of list w recursively with mergesort and merge them
```

```
    return merge(mergesort(w[:mid]), mergesort(w[mid:]))
```

```
def merge(u, v):
```

```
    """Merge two sorted lists u and v together. Return the merged list r."""
```

```
        r=[]
```

```
        while u and v:
```

```
            # pop the smaller element from the front of u or v and append it to list r
```

```
                r.append( u.pop(0) if u[0] < v[0] else v.pop(0) )
```

```
            # extend result with the remaining end
```

```
                r.extend(u or v)
```

```
        return r
```

Дополнительно