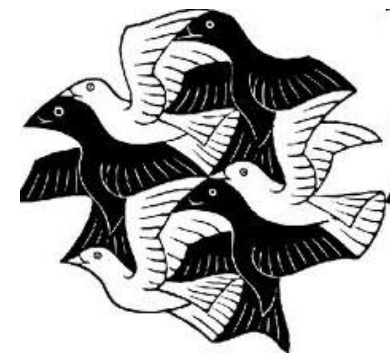


1. Въведение - преосмисляне на ООП

Софтуерни шаблони за проектиране

Боян Бончев,
кат. Софтуерни технологии,
ФМИ - СУ

© 2006-2016

Анотация на лекцията

- Критерии на Meyer за оценка на системната модулност
- ООП – абстракция, скриване на информацията, капсулация, полиморфизъм
- Евристики
- Метрики
- Свързаност
- Кохезия
- Основни принципи в ООП
- SOLID

Литературни източници

- *Object-Oriented Software Construction*, Bertrand Meyer, Prentice Hall, 1988
- *Object-Oriented Design Heuristics*, Arthur Riel, Addison-Wesley, 1996
- Object Coupling and Object Cohesion, chapter 7 of *Essays on Object-Oriented Software Engineering*, Vol 1, Berard, Prentice-Hall, 1993
- *Agile Software Development: Principles, Patterns, and Practices*, Robert C. Martin, Prentice Hall, 2002
- The Sun Java 1.6 Tutorial
- The Sun Java Certification Book
- Wikipedia

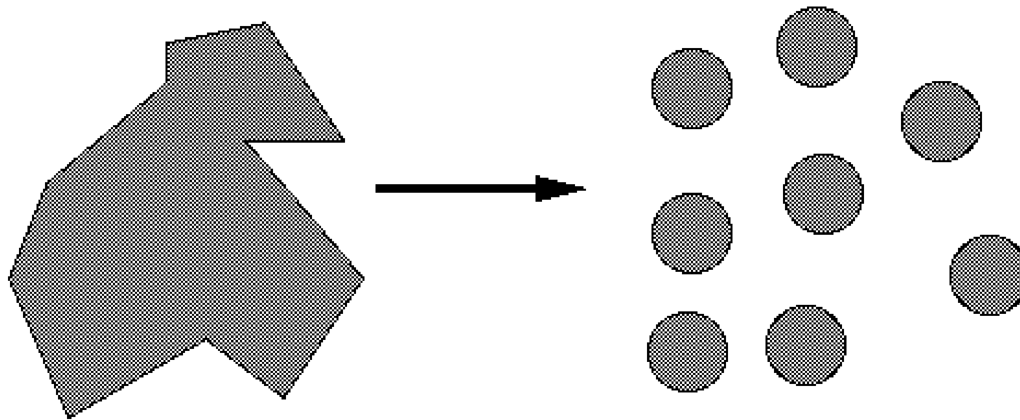
Критерии на Meyer за оценка на системната модулност

- Декомпозируемост
- Композируемост
- Разбираемост
- Непрекъснатост (континюитет)
- Протекция

Декомпозируемост (Decomposability)

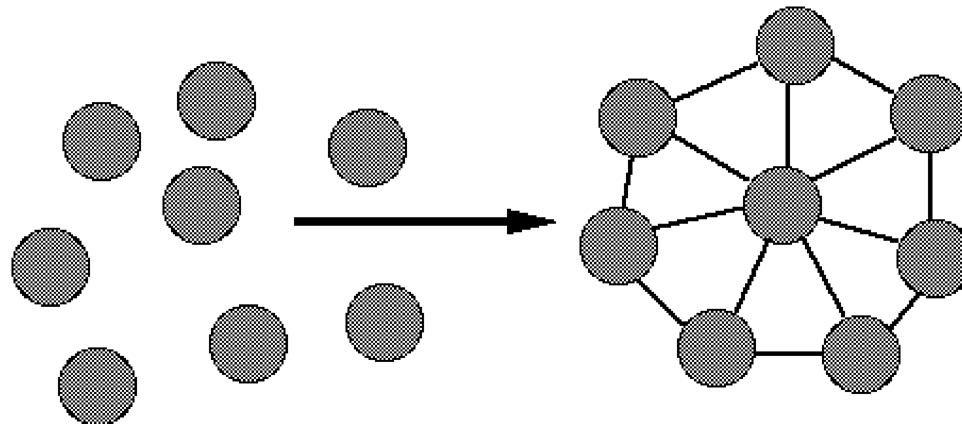
- Идея: проблемът да се разложи на по-малки под-проблеми, които могат да бъдат решени отделно

- Пример: Top-Down дизайн
- Контра-пример: Модул за инициализация



Композируемост

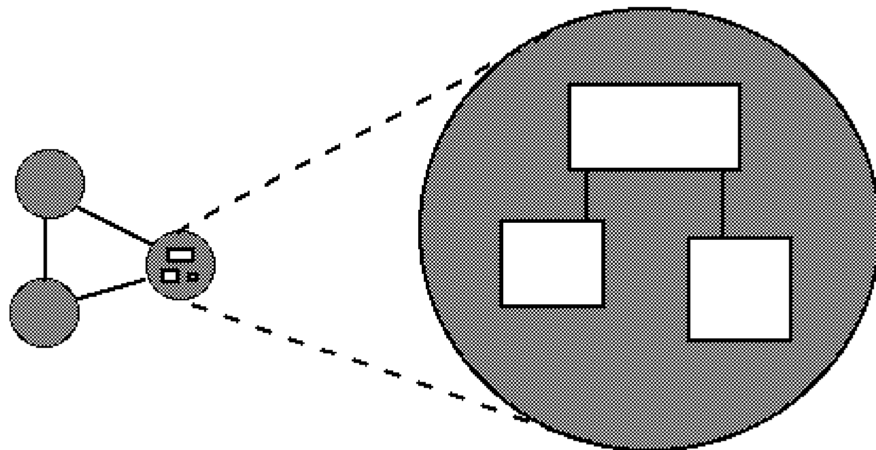
- Идея: да се комбинират свободно модули за създаване на нови системи
- Пример: Math библиотека, Unix command & pipes
- Контра-пример: ?



Разбираемост

- Идея: отделните модули да бъдат разбираеми за читателя

- Пример: ?
- Контра-пример: последователни зависимости

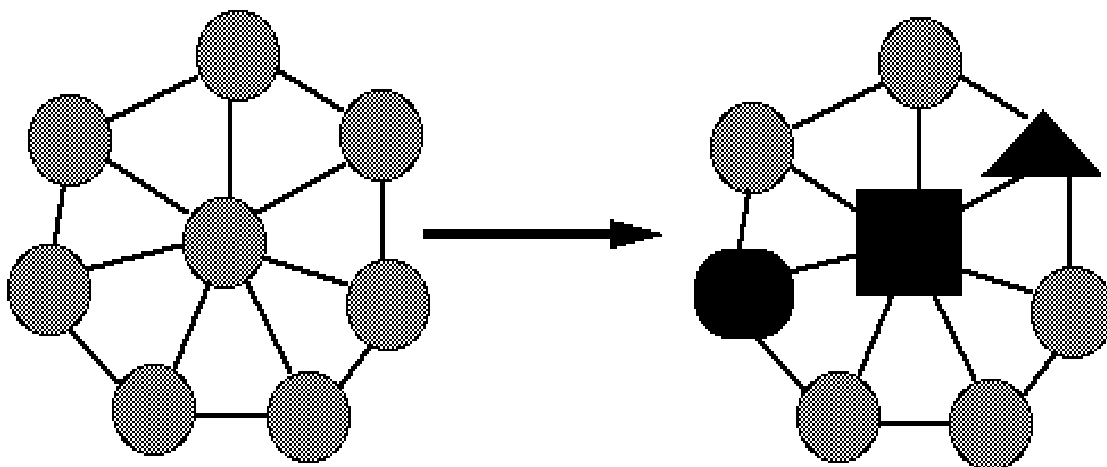


Непрекъснатост (континюитет)

- Идея: малката промяна в спецификацията да има за резултат:
  - - промени в само няколко модула
  - - да не засяга архитектурата

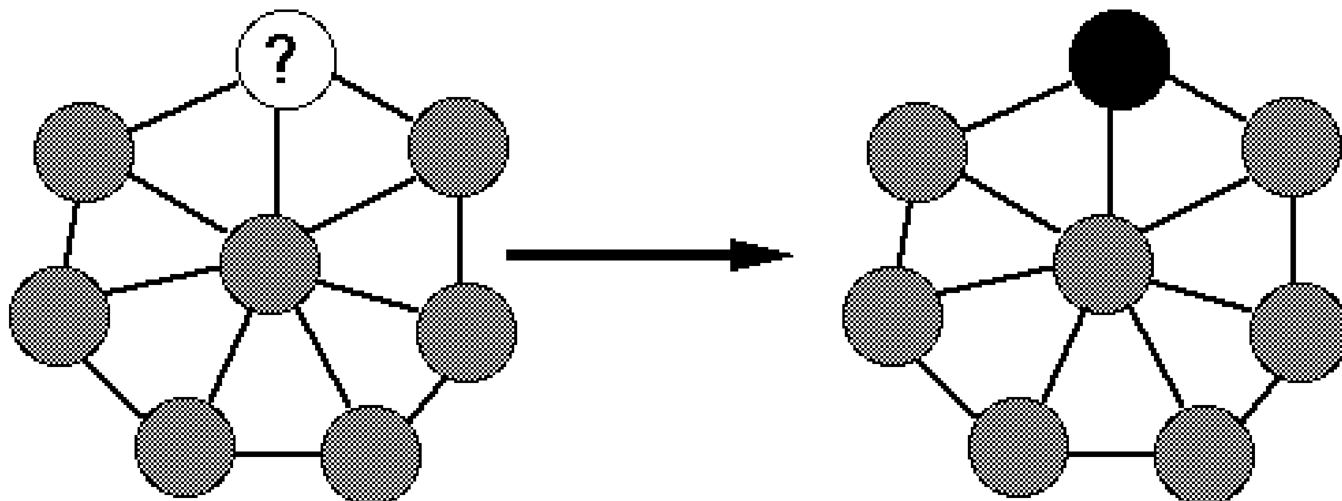
- Пример: константи -> const MaxSize = 100

- Контра-пример: ?



Протекция

- Идея: влиянието на ненормално състояние по време на изпълнение се ограничава до няколко модула
- Пример: валидиране на входа
- Контра-пример: ?



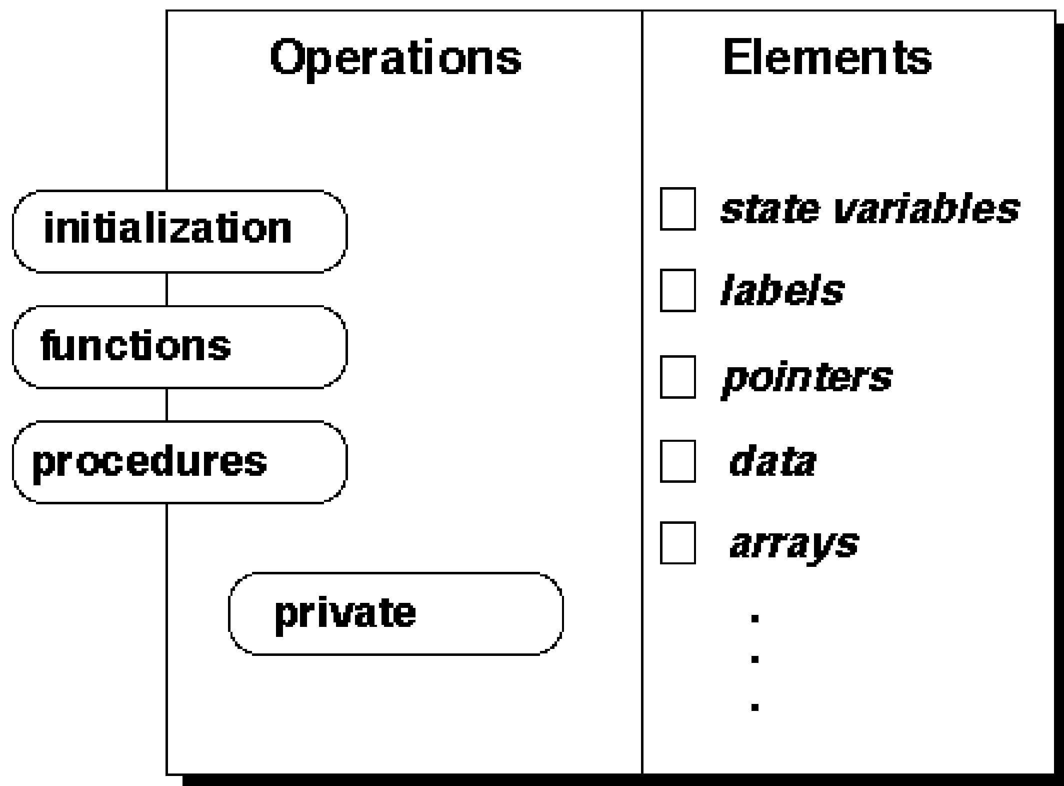
Два важни принципа за разработка на софтуер

KISS: Keep it *simple & stupid*	***Small* is Beautiful**
Поддържа: ■ Разбираемост ■ Композируемост ■ Декомпозируемост	Горна граница за средния размер (редовете код) на операция (метод) *[Lorenz'93]:* Smalltalk – 8 C++ - 24 Java, C# ? *Поддържа*: ■ Декомпозируемост ■ Композируемост ■ Разбираемост

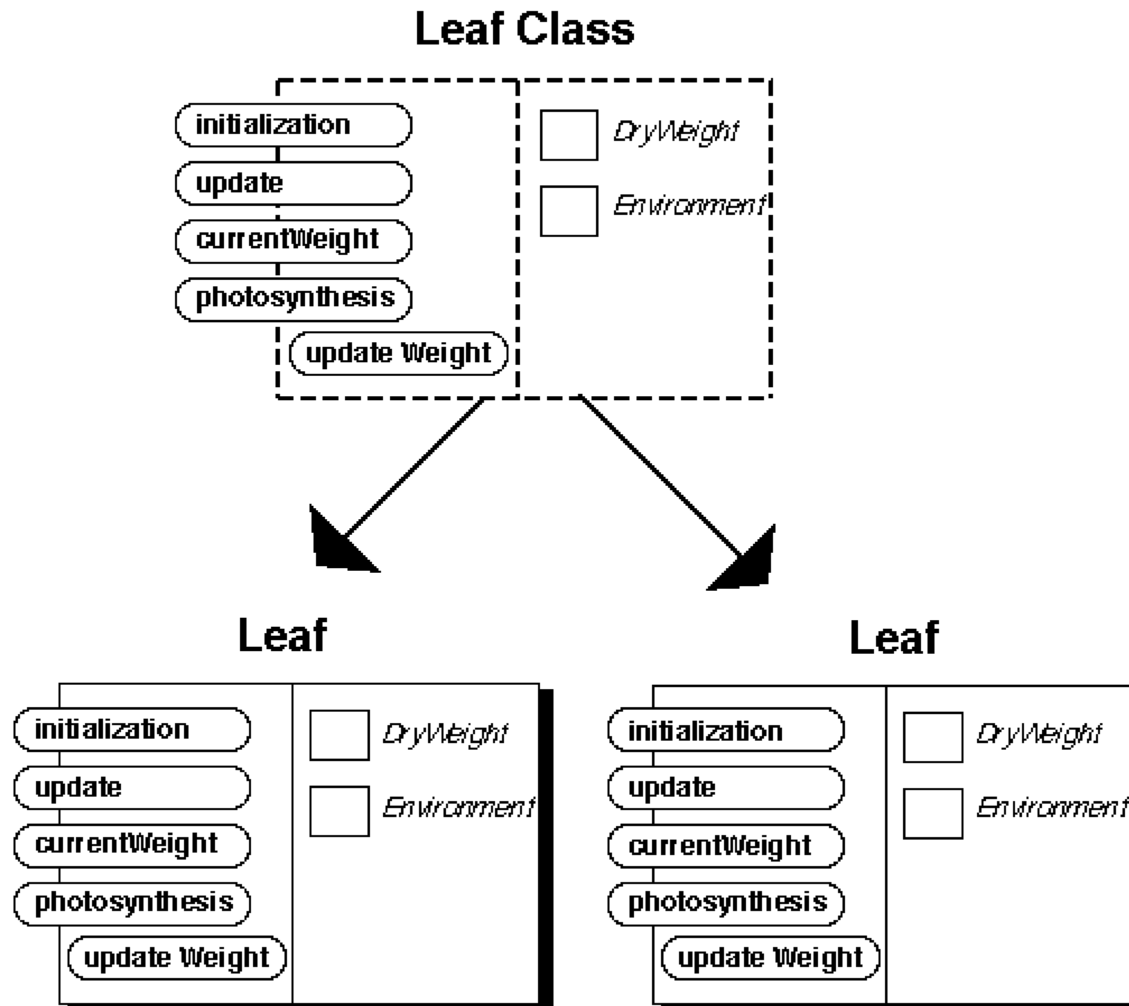
Обектно-ориентирано програмиране (ООП)

- **Дефиниция на езиково ниво**
  - клас
  - обект
  - наследяване
- **Дефиниция на концептуално ниво**
  - абстракция
  - делегация
  - капсулация
  - скриване на информация
  - йерархичност

Дефиниция на езиково ниво на ООП: обект



Дефиниция на езиково ниво на ООП: *клас*



Дефиниция на концептуално ниво на ООП: *абстракция 1/2*

- **абстракция** - практиката на пренебрегване на детайлите, така че можем да се съсредоточим върху ключовото понятие
- " Извличане на най-важните детайли за елемент или група елементи, докато несъществените детайли се игнорират."
  Edward Berard
- “Процесът на определяне на общи модели, които имат систематични промени; една абстракция представлява общ модел и предоставя средства за определяне кой вариант да се използва."
  Richard Gabriel

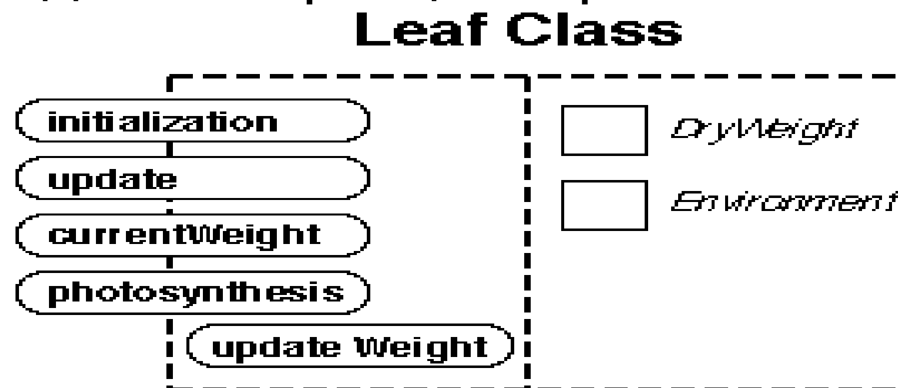
Дефиниция на концептуално ниво на ООП: *абстракция* 2/2

Пример за абстракция:

- Шаблон: опашка с приоритети
- Важни детайли:
  - дължина
  - елементи в опашката
  - операции за добавяне / премахване / намиране на елемент
- Вариации на имплементация: свързан списък спрямо масив; стек; опашка

Дефиниция на концептуално ниво на ООП: *капсулация, скриване на информация*

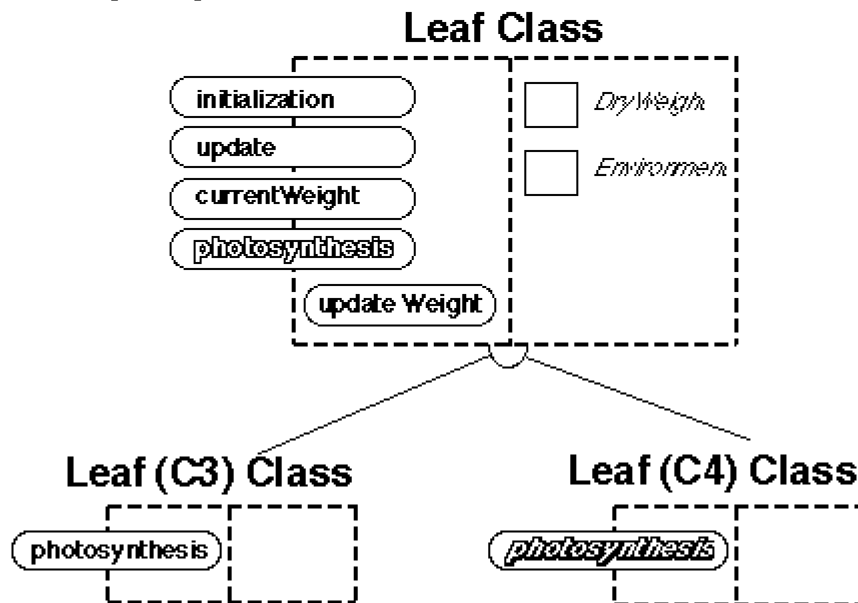
- ***скриване на информация*** – скриване на части на абстракцията в рамките на обекта
- ***капсулация*** - скриване на дизайнерски решения в програмата, които са най-вероятно могат да се променят, като по този начин защитава и други части на програмата, ако дизайнът впоследствие се промени. Включва *скриване на информация* и капсулация на детайли по представянето на данните. Капсуловане на всички части на една абстракция в рамките на един клас-контейнер.
- **Пример:**



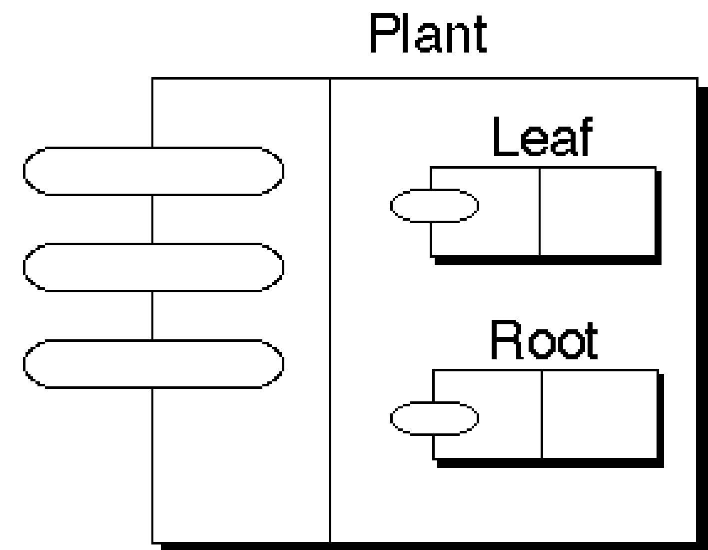
Дефиниция на концептуално ниво на ООП: *йерархии*

- *Абстракции, подредени по ред на рангове или нива*

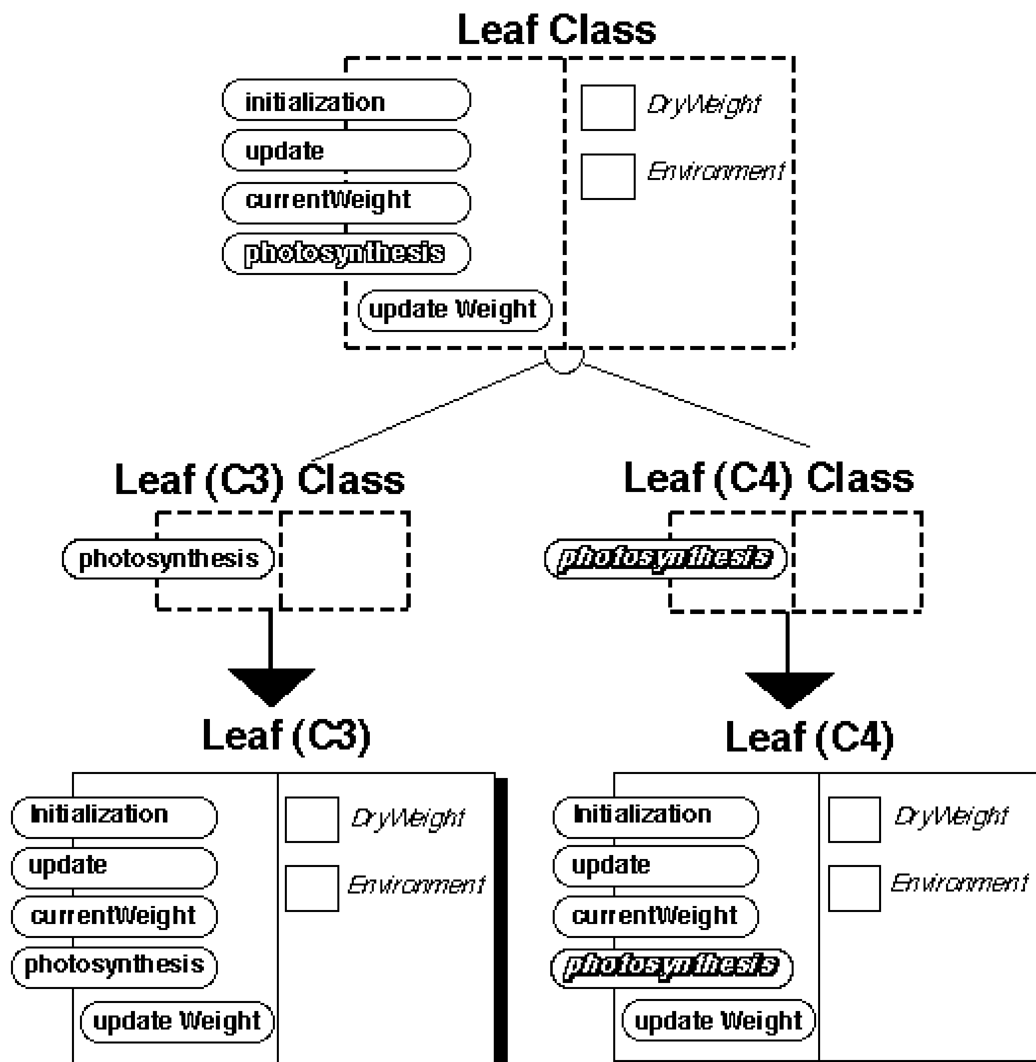
Йерархия на класове



Йерархия на обекти



Дефиниция на езиково ниво на ООП : *наследяване*



Създаване на под-типове спрямо имплементационно наследяване

Очертани са две големи групи наследяване:

- **Създаване на под-типове (Sub-typing)**
  - Логическа класификация на **B** като подтип на **A**. Нарича се също ***наследяване на интерфейси***. **A** обикновено е абстрактен клас (или интерфейс в Java). Всички (или почти всички) операции на **A** са приложими в **B**. Можем да използваме **B** вместо **A**. <u>*Ние често може да наследим и някои имплементации на **A**.*</u>
- **Имплементационно наследяване**
  - Използваме изпълнението (имплементацията) на **A** за удобство в **B**. **A** често е конкретен клас. Само някои операции от **A** са приложими в **B**. <u>*Най-вероятно няма да можем да използваме **B** на мястото на **A**.*</u>
- **Създаването на под-типове (Sub-typing)** е ОК, при условие че класификацията е валидна.
- Имплементационното наследяване е дискусионно за ООП.

Две главни причини да ползваме наследяване

- С цел ***многократно използване на код*** – по-специализирани класове могат да наследяват общата версия на класа и да добавят нова функционалност
- С цел ***полиморфизъм*** – напр. ако преглеждаме списък от различни типове Shape обекти и викаме display() за всеки от тях. При писането на класа не познаваме всички типове Shape подкласове, които друг може да напише.
- **Полиморфни извиквания на методи са възможни само за методите на екземпляра (не на класа)*!***

Дефиниция на концептуално ниво на ООП: *делегация*

Три начина да използваме многократно класове:

- Да генерализираме **A**, обикновено чрез *параметризация*, така че да прибавим нова функционалност към първоначалната за **A**
- *Да създадем нов клас **B** който **делегира** изпълнението на операции към **A**, чрез извикване на методи на обект на класа **A**, рефериран от променлива на екземпляра (instance variable). **B** тогава се нарича клиент на **A**.*
- *Да създадем нов клас **B** който наследява* **A**. ***B** тогава се нарича подклас на **A**.*

- Дискусия – *ЗА и ПРОТИВ?*

Релации от тип IS-A и HAS-A

- A Renault ***IS-A*** Car.
- A Renault ***HAS-A*** Garage.

```
public class Car { }
public class Renault extends Car {
   private Garage homeGarage;
   public void park(int howFast) {
     homeGarage.park(4);
     // Delegates parking to the Garage
   }
}
```

Когато реферираме даден обект: 1/2

- Променливата-референция може да бъде само от един тип, и веднъж декларирана, този тип никога не може да се променя (въпреки че реферираният обект може да се промени!)

- Референцията е променлива, така че може да бъдат пренасочена към други обекти (освен ако не е декларирана като final).

- <u>Типът на референцията определя методите, които могат да бъдат извикани в реферирания обект.</u>

Когато реферираме даден обект: 2/2

- Референцията може да сочи към произволен обект от същия тип като този на деклариране на референцията, или ***към негов под-тип***!

- Променливата-референция може да бъде декларирана като **class** или **interface** (ако е интерфейс, тя може да сочи към всеки обект на клас, който ***имплементира този интерфейс***)

Предефиниране на метод (method overriding) 1/3

- Метод на екземпляра в под-клас със същата сигнатура и тип на резултата както тези на метода на супер-класа предефинира (overrides) метода на супер-класа.
- Списъкът с аргументи трябва точно да съвпада с този на предефинирания (overridden) метод, иначе - overloaded method!
- Връщаният тип трябва да бъде същия като, *или под-тип на*, връщания тип на оригиналния предефиниран метод на супер-класа.

Предефиниране на метод (method overriding) 2/3

- Нивото на достъп не може да бъде по-рестриктивно от това на предефинирания метод, но МОЖЕ да бъде по-малко рестриктивно!

- Instance методите могат да се предефинират, само ако те са наследени от под-класа:
  - В същия пакет на супер-класа и не са маркирани като **private** или **final**.
  - В различен пакет и не са маркирани като **final**, но са **public** или **protected**

Предефиниране на метод (method overriding) 3/3

- Предефиниращият метод МОЖЕ да хвърля unchecked (runtime) exception.
- Предефиниращият метод НЕ МОЖЕ да хвърля checked exceptions, които са ***нови или по-широки*** от тези в предефинирания метод (напр. метод с FileNotFoundException от метод с SQLException).
- Предефиниращият метод МОЖЕ да хвърля ***по-тесни или по-малко*** изключения!
- Не можем да пренаписваме final методи.
- Не можем да пренаписваме static методи.
- Ако метод не може да се наследи => той не може да се предефинира

Извикване на метода от супер-класа вместо предефинирания

```
public class Car {
    public void park() { }
    public void showItself() {
        // Useful printing code goes here
    }
}
class BMW extends Car {
    public void showItself(){
        // Take advantage of Car code, then add some more
        super.showItself(); // Invoke the superclass code
        // Then do BMW-specific print work here
    }
}
```

Предефиниране с ковариантен тип

- Когато подклас предефинира метод, той трябва да спази точно сигнатурата на метода в супер-класа
- Covariant return - от Java 5 насам, можем да променяме връщания тип на предефинирания метод, но само ако той е под-тип на декларирания тип на връщане в супер-класа

```
class Alpha {
    Alpha doStuff(char c) {
        return new Alpha();
    }
}
class Beta extends Alpha {
    Beta doStuff(char c) { // legal override in Java 1.5+
        return new Beta();
    }
}
```

Използване на static

- Използваме static методи за реализиране на поведение, които не зависи от състоянието на екземплярите на класа
- Използваме static променливи за данни специфични за класа, а не за екземпляра – такава променлива има само едно копие; _*static променливите са споделени от обектите на класа*_.
- Всички static променливи пренадлежат на класа, а не на обект на този клас
- static метод не може да работи директно с instance variable – напр., main() се нуждае от референция към обект, за да ползва променливите му
- За достъп до статични променл.: имеКлас**.**статПроменл
- Static методите не могат да ползват ***this*** и ***super***.

Предефиниране и скриване на методи

- Ако под-клас дефинира метод със същата сигнатура като тази на метод в супер-класа, то методът в под-класа ***скрива*** този в супер-класа
- Версията на скрития метод зависи от това дали се обръщаме към него в супер-класа или в под-класа

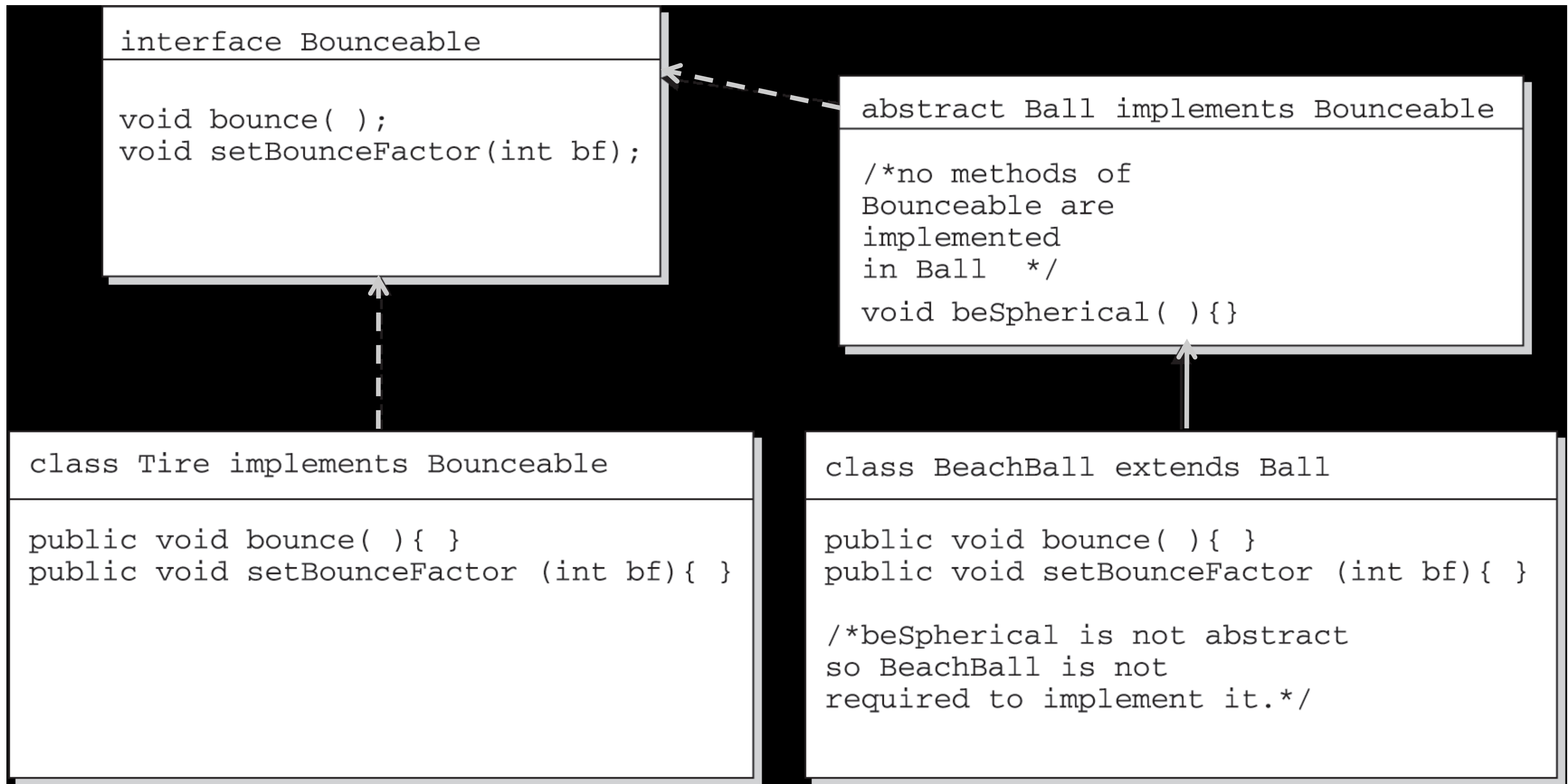
```
public class Animal {
  public static void testClassMethod() {
    System.out.println("Class method in Animal."); }

  public void testInstanceMethod() {
    System.out.println("Instance method in Animal."); }
}
```

```
public class Cat extends Animal {
  public static void testClassMethod() {
    System.out.println("Class method in Cat.");
}
  public void testInstanceMethod() {
    System.out.println("Instance method in Cat."); }
  public static void main(String[] args) {
    Cat myCat = new Cat();
    Animal myAnimal = myCat;
    Animal.testClassMethod();
    myAnimal.testInstanceMethod(); }
}
```

Overloaded (съвместно използвани) методи:

- *Overloading разрешава съвместното използване на методи с еднакво име, но с различни аргументи (опционално, с различен тип на връщане)*
- Overloaded методите ТРЯБВА да променят списъка от аргументи
- Overloaded методите МОГАТ да променят типа на резултата.
- Те МОГАТ да променят модификатора за достъп.
- Те МОГАТ да декларират нови или по-широки checked exceptions.
- Метод може да бъде overloaded *в същия клас или в под-клас*.

Конкретни и абстрактни примери на extends и implements



Примери на правилни и грешни декларации на клас и интерфейс

```
class Foo { } // OK
class Bar implements Foo { } // No! Can't implement a class
interface Baz { } // OK
interface Fi { } // OK
interface Fee implements Baz { } // No! Interface can't  implement an interface
interface Zee implements Foo { } // No! Interface can't implement a class
interface Zoo extends Foo { } // No! Interface can't extend a class
interface Boo extends Fi { } // OK. Interface can extend an interface
class Toon extends Foo, Button { } // No! Class can't extend multiple classes
class Zoom implements Fi, Fee { } // OK. class can implement multiple interfaces
interface Vroom extends Fi, Fee { } // OK. interface can extend multiple interfaces
class Yow extends Foo implements Fi { } // OK. Class can do both
```

Динамично извикване на метод

- Динамичното извикване на метод (dynamic method dispatch) е механизъм, при който решението за това кой предефиниран метод да се извика се взема по време на изпълнение
- Така Java имплементира run-time полиморфизъм
- Версията на предефинирания метод се определя от ***типа на реферирания обект, а не типа на референцията към обекта***

Пример

```
class A {void callme() {System.out.println("Inside A's callme method");}}
class B extends A {
  // override callme()
  void callme() {System.out.println("Inside B's callme method");}
}
class C extends A {void callme() {System.out.println("Inside C's callme
method");}
}
class Dispatch {
	public static void main(String args[]) {
		A a = new A(); // object of type A
		B b = new B(); // object of type B
		C c = new C(); // object of type C
		A r; // obtain a reference of type A
		r = a; // r refers to an A object
		r.callme(); // calls A's version of callme
		r = b; // r refers to a B object
		r.callme(); // calls B's version of callme
		r = c; // r refers to a C object
		r.callme(); // calls C's version of callme
	}
}
```

Конвертиране надолу (Downcasting)

```
class Animal {
        void makeNoise() {System.out.println("generic noise"); }
}
class Dog extends Animal {
        void makeNoise() {System.out.println("bark"); }
        void playDead() { System.out.println(" roll over"); }
}
class CastTest2 {
    public static void main(String [] args) {
        Animal [] a = {new Animal(), new Dog(), new Animal() };
        for(Animal animal : a) {
            animal.makeNoise();
            if(animal instanceof Dog) {
                Dog d = (Dog) animal; //need casting to the ref. var.!
                d.playDead(); // try to do a Dog behavior
            }
        }
    }
}
```

ОК, защото тук
animal instanceof Dog

Конвертираме надолу по дървото на наследяване до по-специфичен клас

Не разчитайте само на компилатора!

```
class Animal { }
class Dog extends Animal { }
class DogTest {
    public static void main(String [] args) {
        Animal animal = new Animal();
        Dog d = (Dog) animal; // compiles but fails later
                                  // with run-time java.lang.ClassCastException
        String s = (String) animal; // cannot compile!
                                  // animal can't EVER be a String
    }
}
```

Downcasting *предполага, че по-късно е възможно да се извика по-специфичен метод*

Конвертиране нагоре (Upcasting)

- За разлика от downcasting, upcasting (конвертиране нагоре по дървото на *наследяване) работи неявно*
- При upcast ние неявно ограничаваме броя на възможните методи, докато при *downcast е възможно по-късно да извикаме по-специфичен метод:*

```
class Animal { }
class Dog extends Animal { }
    class DogTest {
    public static void main(String [] args) {
        Dog d = new Dog();
        Animal a1 = d; // upcast ok with no explicit cast
        Animal a2 = (Animal) d; // upcast ok with an explicit cast
    }
}
```

Конструктори на клас

- Всеки клас, дори и абстрактният*, ТРЯБВА да има конструктор.*
- Конструкторите в Java нямат тип на връщане
- Имената им повтарят името на класа
- Верига на изпълнение на конструктори:
  - Всеки конструктор вика конструктора на своя супер-клас с (неявно) обръщение към super()
  - Накрая се извиква конструктора на Object (Object е последният супер-клас на всички класове)
  - На променливите на екземпляра (обекта) се дават явните им стойности (а не по подразбиране)

Други правила за конструктори

- Конструкторите могат да имат всякакъв достъп, вкл. private.
- Ако в кода на класа не опишем конструктор, то компилаторът ще генерира автоматично *default конструктор БЕЗ АРГУМЕНТИ*
- При обръщение към super() или this() могат да бъдат достъпвани само статични променливи и методи
- Абстрактните класове имат конструктори
- Единственият начин да извикаме конструктор е да се обърнем към него от друг конструктор
- ***Конструкторите не могат да се наследяват – те не са методи и не могат да се предефинират***

Евристики [Roger Whitney]

- Един клас трябва да обхваща една и само една абстракция ->
  клас = *абстракция*

- Дръжте свързаните помежду си данни и поведение на едно място - абстракцията е данни и поведение (методи) = *капсулация*

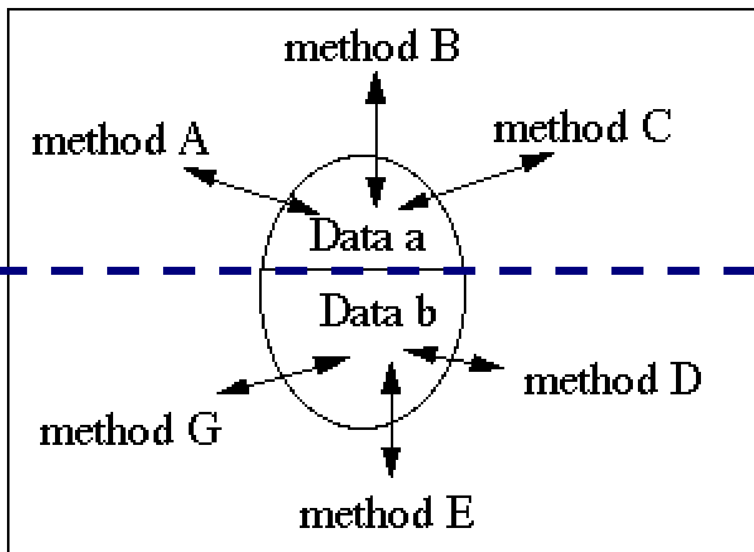
- Всички данни трябва да са скрити вътре в обекта = скриване на информация

- Пазете се от класове, които имат много методи за достъп, определени чрез публичния си интерфейс. Многото методи предполагат, че свързаните с тях данни и поведение не се съхраняват на едно място.

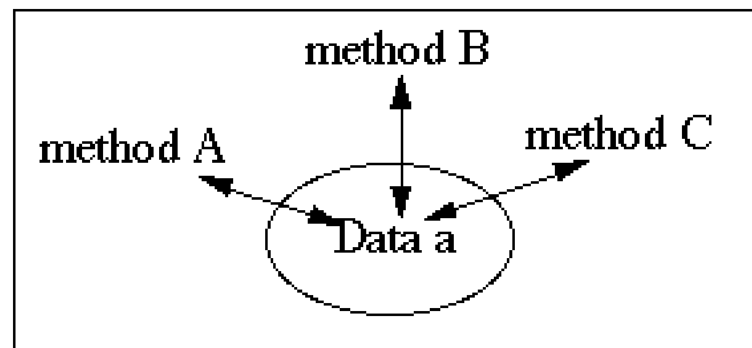
Проблемът на „Божествения клас“ [Roger Whitney] (1/2)

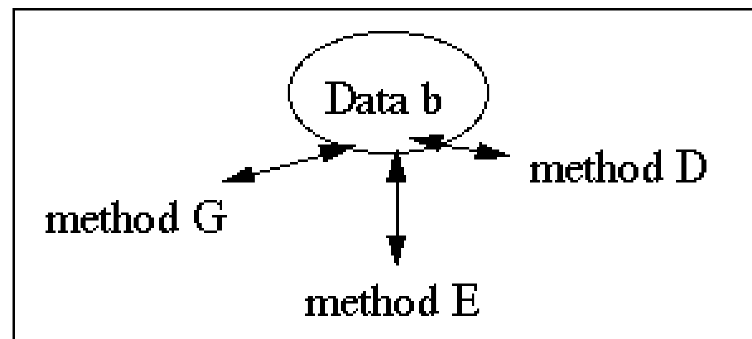
- Разпределете системната функционалност хоризонтално, колкото е възможно по-равномерно така, че класовете от най-високо ниво да поделят работата си по еднакъв начин.
- Не създавайте божествени (всемогъщи) класове и обекти във вашата система. Бъдете много подозрителни към клас, чието име е **Driver**, **Manager**, **System**, или **Subsystem**
- Пазете се от класове, които имат твърде много не-комуникативно поведение, което означава методи, които работят на подмножество от полета за данни на класа. Такива класове често се проявяват чрез не-комуникативно поведение.

Проблемът на „Божествения клас“ [Roger Whitney] (2/2)



solution ->





Метрики [Roger Whitney] **(1/2)**

- Средна горна граница на брой редове в метод: Smalltalk - 5, C++/Java - 15
- Среден брой на методи за клас – да е по-малък от 20
- Среден брой instance променливи на екземпляра (полета, членове-данни) за клас – да е по-малък от 6
- Ниво на вложеност на йерархиите за класа (ниво на йерархичност) – да е по-малко от 6

Метрики [Roger Whitney**] (2/2)**

- Среден брой на линии-коментар за метод – да е по-голям от 1
- Броят рапортувани проблеми за класа да е нисък
- C++ кодът има от 2 до 3 пъти повече редове от този на Smalltalk
- Обемът код ще се разшири през първата половина на проекта и спадне през втората половина, тъй като при ревютата грешките и проблемите ще бъдат отстранени

Свързаност (coupling)

- **Свързаност –** метрика, показваща степента на зависимост на класовете един от друг. Два класа са свързани, когато методи декларирани в единия използват методи или член-променливи, декларирани в другия клас.
- Отношението може да се разглежда и от двете страни – **зависим и влияещ**, но при всички случаи се брои само едната посока. Многократните позовавания към един и същ клас се считат за едно позоваване.
- С цел повишаване модуларността и индуциране на капсулация, свързаността между класове трябва да бъде **свеждана до минимум**.

Свързаността като мярка за взаимна зависимост между модулите

- Излишната и прекомерна свързаност между класовете:
  - води до предразположен към **грешки** код
  - **вредна е за модуларността на дизайна**
  - възпрепятства повторната употреба.

Типове модулна свързаност по ред на нежеланост

Решение: Decompose the operation into multiple primitive operations

- **Свързаност по данни** (най-слаба и най-желана) - изход от един модул е вход за друг
- **Свързаност по управление** - предаване на контролни маркери между модули, така че един модул контролира определянето на последователността на стъпките за обработка в друг модул
- **Свързаност по глобални данни** - две или повече модули споделят същите глобални структури от данни
- **Свързаност по вътрешни данни** (най-силна и най-малко желана) - един модул директно променя локалните данни на друг модул (като C++ Friends)
- **Свързаност по съдържание** (извън класацията) - някои или всички от съдържанието на един модул са включени в съдържанието на друг (като C/C++ header файлове)

Източник: *1) Object свързаност and Object кохезия, chapter 7 of Essays on Object-Oriented Software Engineering, Vol 1, Berard, Prentice-Hall, 1993;*
2) SDSU & Roger Whitney;

Кохезия

- “Cohesion is the *degree to which the tasks* performed by a single module *are functionally related*.“ *IEEE, 1983*
- "A software component is said to exhibit a high degree of cohesion if the elements in that unit exhibit a high degree of *functional relatedness*. This means that each element in the program unit should be essential for that unit to achieve its purpose.“ *Sommerville, 1989*

- Типове на кохезия
  - случайна (най-лоша)
  - логическа
  - темпорална
  - процедурна
  - комуникационна
  - модулна, явяваща се като резултат
  - функционална (най-добра)

Случайна модулна кохезия 1/2

- Малко или никакви конструктивни отношения между елементите на модула
- Общо срещане в обекти:
  - Обектът не представлява една ОО концепция
  - Събиране на често използван сорс код като клас, наследен чрез множествено наследяване
- Пример:

```
class Eclectic{
    public static int findPattern( String text, String pattern) {
        // blah
    }
    public static int average( Vector numbers ) {
        // blah
    }
    public static OutputStream openFile( String fileName ) {
    // blah
    }
}
```

Случайна модулна кохезия 2/2

Орел *и* ***рак и щука****,*
не зная по каква сполука,
такваз им работа дошла
да теглят наедно кола.

Логическа кохезия 1/2

- Модулът изпълнява набор от свързани функции; при викане на модула избираме една чрез подаване на параметър
- Подобна на свързаността по управление
- **Решение:** изолирайте всяка функция в отделни операции

```
public void sample( int flag ) {
	switch ( flag ) {
		case ON:
				// bunch of on stuff
				break;
		case OFF:
				// bunch of off stuff
				break;
		case CLOSE:
				// bunch of close stuff
				break;
	}
}
```

Логическа кохезия 2/2

Модулът изпълнява набор от
свързани функции;
при викане на модула избираме
една чрез подаване на параметър...

До кога?

Източник: 1) Object свързаност and Object кохезия, chapter 7 of Essays on Object-Oriented Software Engineering, Vol 1, Berard, Prentice-Hall, 1993;
2) SDSU & Roger Whitney;

Темпорална кохезия 1/2

- Елементите са групирани в един общ модул, тъй като те се обработват в рамките на същия ограничен период от време
- Популярен пример:
  - "Initialization" модули, които осигуряват стойности по подразбиране за много обекти
  - "End of Job" модули – за много обекти

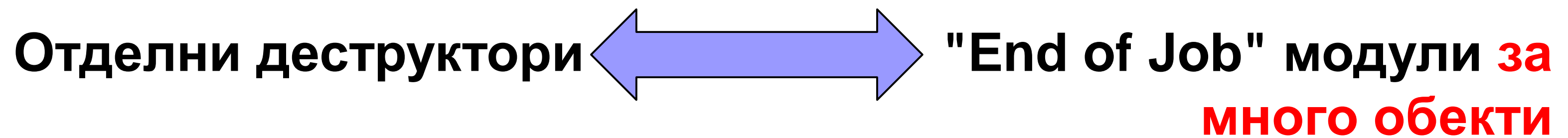
```
procedure initializeData() {
    font = "times"; windowSize = "200,400";
    foo.name = "Not Set"; foo.size = 12;
    foo.location = "/usr/local/lib/java";
}
```

- **Решение:** всеки обект да има конструктор и деструктор

```
class foo {
    public foo() {
        foo.name = "Not Set";
        foo.size = 12;
        foo.location = "/usr/local/lib/java";
} }
```

Call these constructors/ destructors from a non-object oriented routine that performs a single, cohesive task

Темпорална кохезия 2/2



Отделни деструктори ⟺ **"End of Job" модули за много обекти**

Процедурна кохезия 1/2

- Асоциира обработващи елементи въз основа на техни процедурни или алгоритмични отношения
- Процедурните модули са специфични за отделните приложения
- В контекста на модула изглежда разумна
- Извадени от контекста, тези модули изглеждат странно и е много трудно да се разберат
- *Обаче*:
  - Модулът не може да се разбере без разбиране на програмата и съществуващите условия, когато се извиква този модул
  - Прави модула труден за промени и за разбиране
- **Решение:** направете редизайн на системата
- **Class Builder** срещу **Program writer**

Процедурна кохезия 2/2

- Модулът не може да се разбере без разбиране на програмата и съществуващите условията, когато се извиква този модул
- Прави модула труден за промени и за разбиране

- **Решение:** направете ТОТАЛЕН редизайн на системата!

Комуникационна кохезия 1/2

- Операциите на модул работят върху един и същ вход набор от данни
и / или
- произвеждат едни и същи изходни данни

Решение:

- Изолирайте всеки елемент в отделен модул

- Рядко се среща в обектно-ориентирани системи, заради полиморфизма

Комуникационна кохезия 2/2

- Операциите на модул работят върху един и същ вход набор от данни

Решение:

- Изолирайте всеки елемент в отделен модули

Модулна кохезия, явяваща се като резултат (Sequential, Pipeline) 1/2

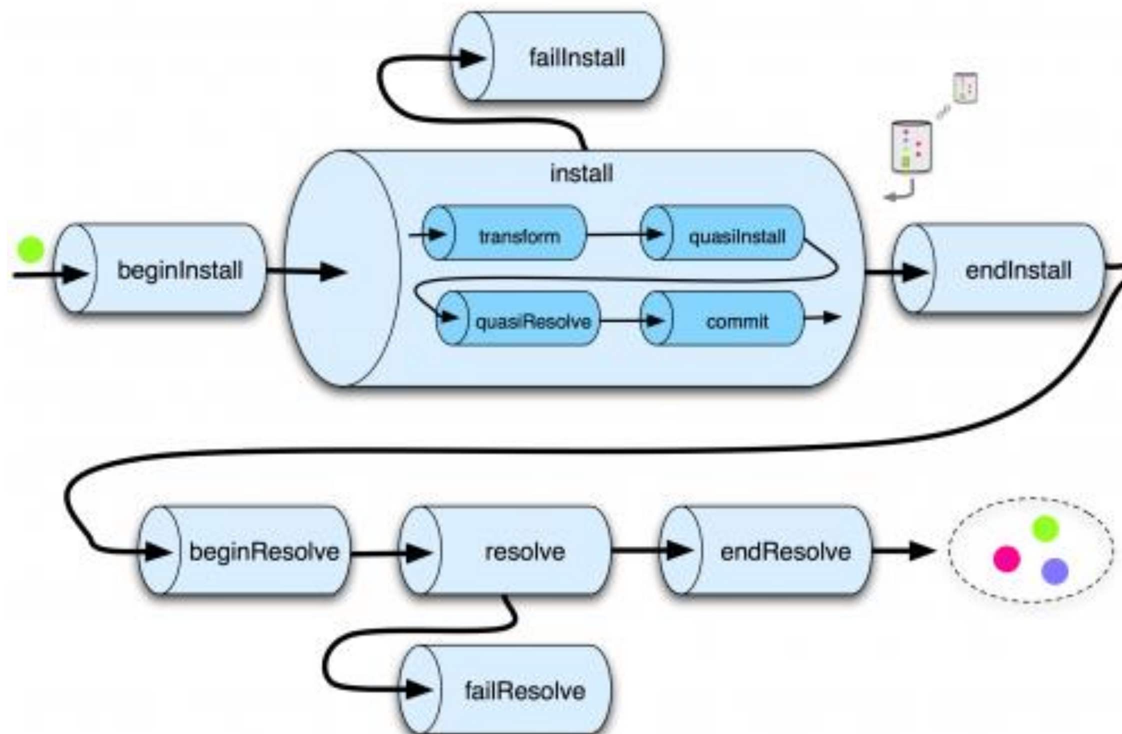
- Тип кохезия, в който изходните данни от една обработка служат като входни данни за следващия елемент
- Модул, който изпълнява множество последователни функции, където последователните отношения между всички функции са предизвикани от вътрешни проблеми или прилагането баланс между всички функции

Решение:

- Декомпозирайте на по-малки модули

Модулна кохезия, явяваща се като резултат (Sequential, Pipeline) 1/2

Решение:
Декомпозирайте на по-малки модули

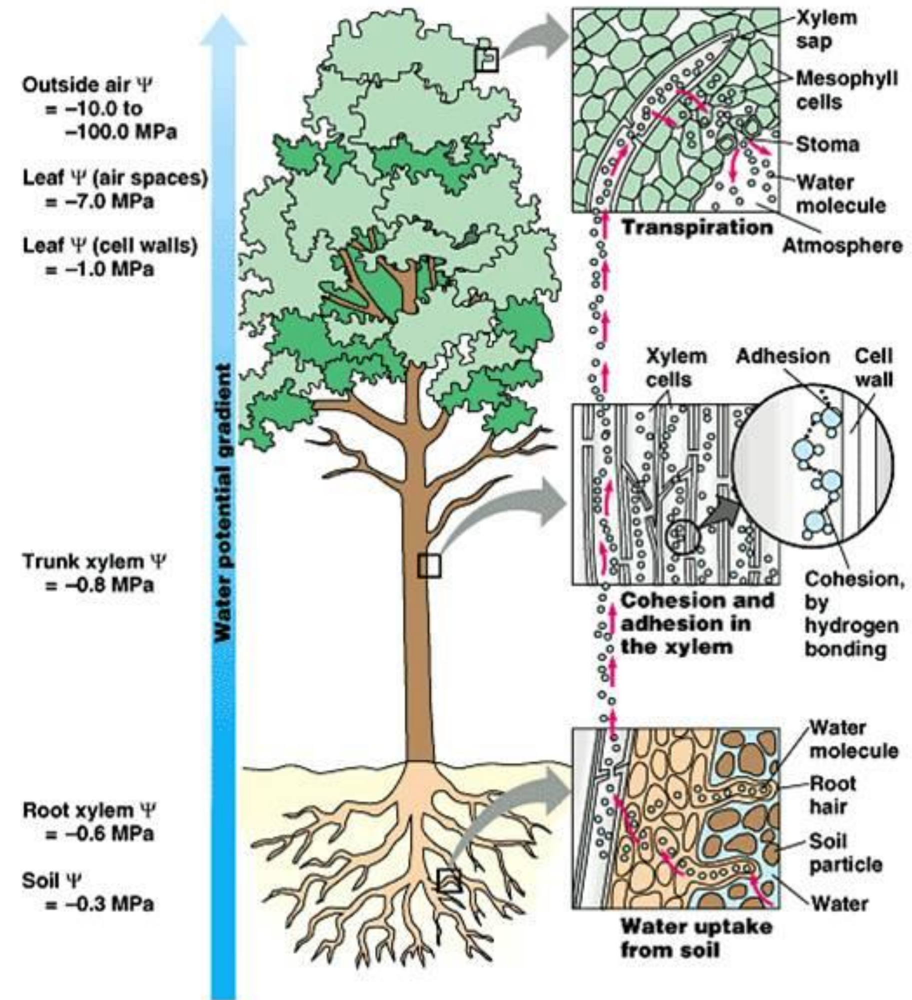


Функционална кохезия 1/2

- Ако операциите на модул могат да се опишат като една специфична функция по съгласуван начин, модулът има функционален кохезия
- Ако не, модулът има по-нисък тип кохезия
- В една ОО система:
  - Всяка операция в публичния интерфейс на обект трябва да бъде функционално сплотена, кохезивна
  - Всеки обект трябва да представлява една кохезивна концепция

Функционална кохезия 2/2

- Ако операциите на модул могат да се опишат като една специфична функция по съгласуван начин...
- В една ОО система:
  - Всяка операция в публичния интерфейс на обект трябва да бъде функционално сплотена, кохезивна
  - Всеки обект трябва да представлява една кохезивна концепция



SOLID принципи за проетиране *(за самостоятелна работа)*

- **S**RP: The Single Responsibility Principle
- **O**CP: The Open/Closed Principle
- **L**SP: The Liskov Substitution Principle
- **I**SP: The Interface Segregation Principle
- **D**IP: The Dependency Inversion Principle

Източник: *Agile Software Development: Principles, Patterns, and Practices.* Robert C. Martin, Prentice Hall, 2002.

SRP: The Single Responsibility Principle 1/2

- In object-oriented programming, the single responsibility principle states that *every object should have a single responsibility*, and that responsibility should be entirely encapsulated by the class. All its services should be narrowly aligned with that responsibility.
- The term was introduced by Robert C. Martin in his Principles of Object Oriented Design, and his book Agile Software Development, Principles, Patterns, and Practices - described it as being based on the principle of кохезия.
- The single responsibility principle is used in responsibility driven design methodologies like the Responsibility Driven Design (RDD) and the Use Case / Responsibility Driven Analysis and Design (URDAD).

Източник: http://en.wikipedia.org/

SRP: The Single Responsibility Principle 2/2

- Example: consider a module that compiles and prints a report. Such a module can be changed for two reasons:
  - First, the content of the report can change.
  - Second, the format of the report can change.
- These two things change for very different causes; one substantive, and one cosmetic. The single responsibility principle says that these two aspects of the problem are really two separate responsibilities, and should therefore be in separate classes or modules. It would be a bad design to couple two things that change for different reasons at different times.
- The reason it is important to keep a class focused on a single concern is that it makes the class more robust. Continuing with the foregoing Пример, if there is a change to the report compilation process, there is greater danger that the printing code will break if it is part of the same class.

OCP: The Open/Closed Principle 1/2

- The open/closed principle states: "software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification";
- An entity can allow its behavior to be modified without altering its Източник code.
- This is especially valuable in a production environment, where changes to Източник code may necessitate code reviews, unit tests, and other such procedures to qualify it for use in a product: code obeying the principle doesn't change when it is extended, and therefore needs no such effort.

Източник: http://en.wikipedia.org/

OCP: The Open/Closed Principle 2/2

- Meyer's Open/Closed Principle - in his 1988 book Object Oriented Software Construction: once completed, the implementation of a class could only be modified to correct errors; **new or changed features would require that a different class be created**. That class could reuse coding from the original class through наследяване. The derived subclass might or might not have the same interface as the original class - *advocates implementation наследяване* – the existing implementation is closed to modifications, and new implementations need not implement the existing interface.

- Polymorphic Open/Closed Principle - Robert C. Martin's 1996 article "The Open-Closed Principle" redefines it to refer to the use of abstracted interfaces, where the implementations can be changed and *multiple implementations could be created and polymorphically substituted for each other* - advocates наследяване from abstract base classes. The existing interface is closed to modifications.

Източник: http://en.wikipedia.org/

LSP: The Liskov Substitution Principle

- Barbara Liskov in a 1987: **Let q(x) be a property provable about objects x of type T. Then q(y) should be true for objects y of type S where S is a subtype of T**.
- Liskov's notion of a behavioral subtype defines a notion of substitutability for mutable objects; that is, if S is a subtype of T, then objects of type T in a program may be replaced with objects of type S without altering any of the desirable properties of that program (e.g., correctness).
- Behavioral subtyping is a stronger notion than typical subtyping of functions defined in type theory, which relies only on the contravariance of argument types and covariance of the return type:
  - **Contravariance** (converting from narrower to wider) of method arguments in the subtype.
  - **Covariance** (converting from wider to narrower) of return types in the subtype.
  - **No new exceptions** should be thrown by methods of the subtype, except where those exceptions are themselves subtypes of exceptions thrown by the methods of the supertype.

ISP: The Interface Segregation Principle

- The ISP was formulated by Robert C. Martin for Xerox. Xerox had created a new printer system which could perform a variety of tasks such as stapling a set of printed papers, faxing, and so forth. The software grew it became harder and harder to change - there was one main Job class that was used by almost all of the tasks. Anytime a print job or a stapling had to be done, a call was made to some method in the Job class. This meant that the Job class was getting huge or 'fat', full of tons of different methods which were specific to a variety of different clients.
- Martin suggested that they add a layer of interfaces to sit between the Job class and all of its clients. Using the properties of the Dependency Inversion Principle, all of the dependencies could be reversed. Instead of having just one 'fat' Job class that all the tasks used, there would be a Staple Job interface or a Print Job interface that would be used by the Staple class or Print class, respectively, and would call methods of the Job class.

Источник: http://en.wikipedia.org/

DIP: The Dependency Inversion Principle

- Coined by Robert C. Martin in 1996
- DIP refers to a specific form of decoupling where conventional dependency relationships established from high-level, policy-setting modules to low-level, dependency modules are inverted (e.g. reversed) for the purpose of *rendering high-level modules independent of the low-level module implementation details*.
- DIP states:
  - *A. High-level modules should not depend on low-level modules. Both should depend on abstractions.*
  - *B. abstractions should not depend upon details. Details should depend upon abstractions.*

Източник: http://en.wikipedia.org/

SOLID Class Design Principles

- **S**RP: The Single Responsibility Principle
- **O**CP: The Open/Closed Principle
- **L**SP: The Liskov Substitution Principle
- **I**SP: The Interface Segregation Principle
- **D**IP: The Dependency Inversion Principle

Homework:

http://butunclebob.com/files/SDWest2006/AdvancedPrinciplesOfClassDesign.ppt