Σχολή Ηλεκτρολόγων Μηχανικών & Μηχανικών Η/Υ

Εθνικό Μετσόβιο Πολυτεχνείο

Τεχνολογία Λογισμικού

7ο Εξάμηνο 2019 - 20

Ν.Παπασπύρου, Καθ. ΣΗΜΜΥ, nickie@softlab.ntua.gr

Β.Βεσκούκης, Αν.Καθ. ΣΑΤΜ, v.vescoukis@cs.ntua.gr

Κ.Σαΐδης, ΠΔ 407, saiko@softlab.ntua.gr

Περιεχόμενα

1. Βασικές σχεδιαστικές αρχές λογισμικού
2. Η έννοια του τεχνικού χρέους (technical/design/code debt)
3. Αντικειμενοστραφής προγραμματισμός (με παραδείγματα σε Java, Javascript)

1. Σχεδιαστικές αρχές λογισμικού

Ποιο είναι το ζητούμενο

Να σχεδιάσουμε το λογισμικό με τέτοιο τρόπο που να είναι:

- ευκολονόητο (comprehensible)
- επεκτάσιμο (extensible)
- εξελίξιμο (evolvable)
- συντηρήσιμο (maintainable).

Αφαίρεση (Abstraction)

Θεμελιώδης έννοια

- "Η εννοιολογική διαδικασία κατά την οποία προκύπτουν γενικοί κανόνες από την εξέταση επιμέρους παραδειγμάτων." (Wikipedia)
- Χρησιμοποιείται σε πολλές επιστήμες.
- Αποτελεί το κύριο εργαλείο σχεδιασμού.

Αφαίρεση

Αναπαράσταση ενός ουσιώδους χαρακτηριστικού του λογισμικού χωρίς τις δευτερεύουσες λεπτομέρειες.

Παράδειγμα

```
interface Item {
  String getId()
  void setId(String id)
  Map<String, Object> toMap()
  void fromMap(Map<String, Object> map)
}

interface Datastore {
  Item load(String id)
  void save(Item item)
}
```

Βελτίωση (Refinement)

- Συμπληρωματική διαδικασία της αφαίρεσης
- Τμηματική προσαρμογή των αφαιρέσεων σε νέες απαιτήσεις, περιορισμούς ή αποφάσεις

Παράδειγμα

```
interface Datastore {
  boolean exists(String id)
  void load(String id, Function<Item> callback)
  void save(Item item, Function<Item> callback)
}
```

Refactoring

- Η διαδικασία επαναπροσαρμογής του κώδικα ως τμήμα κάποιας βελτιωτικής απόφασης
- Αυτοματοποιείται από πολλά IDEs
- Θα δούμε λεπτομερές παράδειγμα σε επόμενη διάλεξη

Τμηματοποίηση (modularity)

- Αναλύουμε ένα πολύπλοκο σύστημα σε επιμέρους απλούστερα τμήματα
- Σχεδιάζουμε ξεχωριστά το ένα τμήμα από το άλλο
- Συνθέτουμε ξανά τα τμήματα και τις αλληλεπιδράσεις τους σε ένα ενιαίο σύνολο
- Με σκοπό τη διευκόλυνση της υλοποίησης, της συντήρησης και της εξέλιξης του λογισμικού

> The most difficult design task is to find the most appropriate decomposition of the whole into a module hierarchy, minimizing function and code duplications.
>
> N. Wirth

Απόκρυψη πληροφορίας (information hiding)

- Η απόκρυψη των χαρακτηριστικών που μπορεί να αλλάξουν στο λογισμικό (σχεδιαστικών αποφάσεων, λεπτομερειών υλοποίησης) από τα άλλα τμήματα, ώστε να ελαχιστοποιηθούν οι αλλαγές που απαιτούνται αν/όταν προκύψει η αλλαγή
- Παροχή σταθερών interfaces (αφαιρέσεων) για την επικοινωνία μεταξύ των τμημάτων

Συναφείς έννοιες / αρχές

- Ενθυλάκωση (encapsulation)
- Διαχωρισμός ενδιαφερόντων (separation of concerns)
- Σύζευξη & συνεκτικότητα (coupling & cohesion)

Ενθυλάκωση (encapsulation)

- Διαχωρισμός της δομής από τη συμπεριφορά ενός συστατικού
- Διαχωρισμός της αφαίρεσης από την υλοποίησή της
- Προστασία ενός συστατικού από τη μετάβασή του σε μη έγκυρη κατάσταση
- Μπορεί να θεωρηθεί ως τεχνική υλοποίησης της αρχής της απόκρυψης πληροφορίας

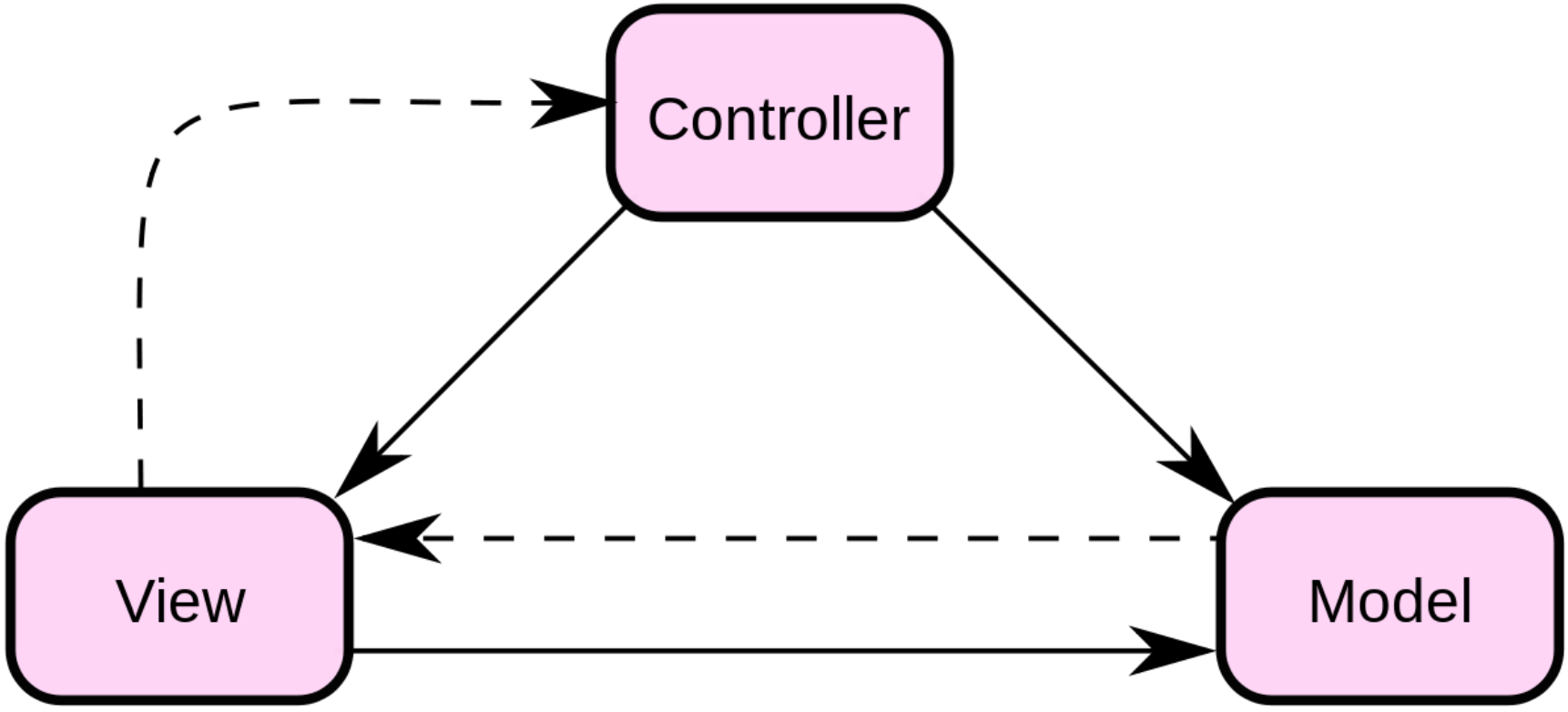
Παράδειγμα

```
class MySQLDatastore implements Datastore {
  private Connection con  //encapsulation
  public boolean exists() {
    String query = "select 1 from items where item = $"
    ResultSet rs = con.execute(query, getId())
    return !rs.isEmpty()
  }
  ...
}
```

Διαχωρισμός ενδιαφερόντων (separation of concerns)

- Κάθε τμήμα του λογισμικού επικεντρώνεται στην επίλυση ενός ξεχωριστού ζητήματος (concern)
- Αρχιτεκτονικά επίπεδα (επίπεδο παρουσίασης, επίπεδο επιχειρησιακής λογικής, επίπεδο πρόσβασης δεδομένων)

Παράδειγμα



Σύζευξη & Συνεκτικότητα

Σύζευξη (coupling)

Ο βαθμός της αλληλεξάρτησης μεταξύ δύο συστατικών (κλάσεων, modules, κλπ)

Συνεκτικότητα (cohesion)

Ο βαθμός με τον οποίο τα στοιχεία που ανήκουν σε ένα συστατικό σχετίζονται λειτουργικά (ορθώς συνανήκουν στο ίδιο συστατικό)

Ζητούμενο

Χαλαρή σύζευξη (loose coupling) μεταξύ των συστατικών

Υψηλή συνεκτικότητα (high cohesion) "μέσα" σε κάθε συστατικό

Επαναχρησιμοποίηση (reuse)

- Το λογισμικό δεν πρέπει να ανακαλύπτει κάθε φορά εκ νέου τον τροχό
- Οι καλές αφαιρέσεις και τα καλώς διαχωρισμένα συστατικά είναι εύκολο να επαναχρησιμοποιηθούν

Παράδειγμα

```
var animals = ["dog", "cat", "fish"]
var len = function(s) { return s.length; }
var sum = function(a, b) { return a+b; }
animals.map(len).reduce(sum, 0); //10
```

(reuse) οι συναρτήσεις len, sum μπορεί να προϋπάρχουν

Μήπως ο ίδιος ο κώδικας είναι το design;

> TEX would have been a complete failure if I had merely specified it and not participated fully in its initial implementation. The process of implementation constantly led me to unanticipated questions and to new insights about how the original specifications could be improved.
>
> Donald Knuth

Πρόσθετες σχεδιαστικές αρχές (design principles)

Πώς να δομήσουμε και να υλοποιήσουμε τις αφαιρέσεις (abstractions)

Μην επαναλαμβάνεσαι (Don't Repeat Yourself)

> "Every piece of knowledge must have a single, unambiguous, authoritative representation within a system."
>
> A. Thomas, D. Hunt

Μόνο μια φορά (Once and Only Once)

- Αρχή του Extreme Programming (XP)
- Each and every declaration of behavior should appear Once and Only Once.

DRY vs WET

- WET: We Enjoy Typing
- WET: Waste Everybody's Time

Εφαρμογή της αρχής DRY στην πράξη

- Η επανάληψη (duplication) αυξάνει την τυχαία/τεχνητή πολυπλοκότητα του συστήματος (accidental complexity).
- Οπουδήποτε στον κύκλο ζωής του λογισμικού υπάρχουν "χειροκίνητες" διαδικασίες, θα πρέπει να αυτοματοποιούνται (π.χ. μέσω build system, scripting, κ.ά.).
- Αν προκύπτουν επαναλήψεις στη λογική, κάπου απαιτείται η προσθήκη μιας νέας αφαίρεσης.

Σύνολο αρχών S.O.L.I.D.

- Single responsibility (S)
- Open/closed (O)
- Liskov substitution (L)
- Interface segregation (I)
- Dependency inversion (D)

Μοναδική ευθύνη (Single responsibility)

- Κάθε κλάση/συστατικό πρέπει να έχει μία και μοναδική ευθύνη.
- A class should have only one reason to change.

Ανοικτό/κλειστό (Open/closed)

- Κάθε κλάση/συστατικό πρέπει να είναι ανοικτή σε επεκτάσεις (π.χ. προσθήκη νέων πεδίων ή μεθόδων).
- Κάθε κλάση/συστατικό πρέπει να είναι κλειστή και οριοθετημένη (έτοιμη προς χρήση από τρίτα συστατικά).

Δυνατότητα αντικατάστασης (Liskov substitution)

- Αν το `S` είναι υποτύπος του `T` τότε όλα τα αντικείμενα του δεύτερου θα πρέπει να μπορούν να αντικατασταθούν με αντικείμενα του πρώτου χωρίς να αλλοιωθεί κανένα από τα επιθυμητά χαρακτηριστικά του συστήματος.
- Strong behavioral subtyping.

Επιμερισμός διεπαφών (Interface segregation)

- Κανένα συστατικό δεν πρέπει να εξαρτάται από μεθόδους που δε χρησιμοποιεί.
- Χρήση μικρών και διαφορετικών interfaces για τη θέσπιση των διεπαφών μεταξύ συστατικών.

Ανιστροφή εξαρτήσεων (Dependency inversion)

- Τα υψηλού επιπέδου συστατικά δεν πρέπει να εξαρτώνται από τα χαμηλού επιπέδου συστατικά. Και τα δύο θα πρέπει να εξαρτώνται από κοινές αφαιρέσεις.
- Οι αφαιρέσεις δεν πρέπει να εξαρτώνται από λεπτομέρειες. Οι λεπτομέρειες θα πρέπει να εξαρτώνται από τις αφαιρέσεις.

Παράδειγμα

```
class DataAccessLayer {
  private Function<Item> updateIndex = ...
  //private MySQLDatastore store  //bad - strong coupling
  private Datastore store         //good
  DataAccessLayer() {
   //store = new MySQLDatastore() //bad - strong coupling
   store = InjectConfig.get('Datastore') //good - "injection"
  }
  void save(Item item) {
   store.save(Item item, updateIndex)
  }
}
```

Dependency inversion through injection (απλουστευμένο παράδειγμα)

2. Η έννοια του τεχνικού χρέους (technical/design/code debt)

Ποιοτικά ζητούμενα σχεδιασμού

- Performance - Scalability
- Maintainability - Extensibility
- Security - Safety
- Robustness - Fault-tolerance
- Usability - Reliability

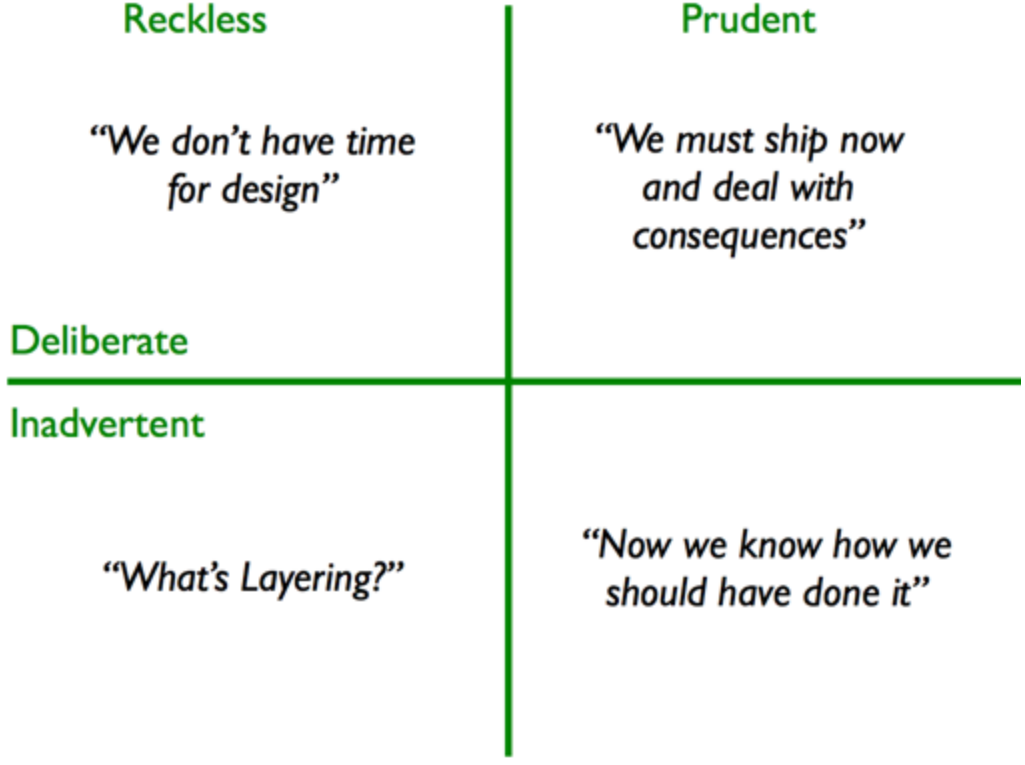
Σχεδιασμός = Συμβιβασμός

- Συνήθως δεν είναι εφικτό να γίνουν όλα καλά με την πρώτη (τη δεύτερη, την τρίτη, … 😄)
- Επιλογή των επιθυμητών trade-offs

Τεχνικό χρέος

- Το κόστος της πρόσθετης δουλειάς που θα απαιτηθεί από την επιλογή μιας εύκολης και γρήγορης υλοποίησης αντί για την εφαρμογή της συνολικά καλύτερης λύσης.

Τέσσερα είδη



By Martin Fowler

Το κλασικότερο trade-off

Efficiency vs Abstraction

> Programmers have spent far too much time worrying about efficiency in the wrong places at the wrong times; premature optimization is the root of all evil.
>
> Donald Knuth

3. Αντικειμενοστραφής προγραμματισμός

Αντικείμενα

- Ενθυλακώνουν (encapsulate)
  - Κατάσταση (state)
    - δεδομένα που τηρούνται σε πεδία
  - Συμπεριφορά (behavior)
    - λειτουργίες που τηρούνται σε μεθόδους (behavior)
- Στιγμιοτύπιση (instantiation)
  - Μέσω κατασκευαστών (constructors)
- Αυτο-αναφορά (this, self)
- Ανταλλαγή μηνυμάτων (message passing)

Βασικές έννοιες

- Συνάθροιση (aggregation)
- Σύνθεση (composition)
- Κληρονομικότητα (inheritance)
  - Με βάση κλάσεις / διεπαφές (classes / interfaces)
  - Με βάση πρωτότυπα (prototypes)
- Δυναμική αποστολή μηνυμάτων (dynamic method dispatch)
- Αργή δέσμευση (late binding)
- Πολυμορφισμός

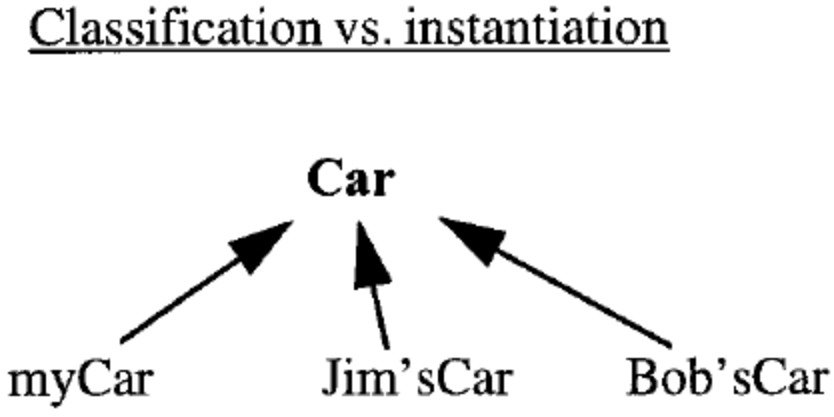
Βασικές αντικειμενοστραφείς αρχές αφαίρεσης (OO abstraction principles)

Το σήμειο που το **αντικειμενοστραφές μοντέλο** συνδέεται με την εννοιολογική μοντελοποίηση (conceptual modeling) και την αναπαράσταση γνώσης (knowledge represenation)

Λίστα αναγνωσμάτων

Antero Taivalsaari, "On the notion of inheritance", ACM Computing Surveys, Vol. 28, No 3, September 1996.

I. Classification - Instantiation



A. Taivalsaari

Classification - Instantiation

- Σχέση του αντικειμένου με την κλάση του και αντίστροφα.
- Όλα τα αντικείμενα / στιγμιότυπα μιας κλάσης μοιράζονται κοινά και ομοιόμορφα χαρακτηριστικά.
- Η κλάση είναι το intensional abstraction όλων των δυνατών της αντικειμένων.

Παράδειγμα (Java)

```
class Point {
  private int x;
  private int y;

  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }
  public int getX() { return x; }

  public int getY() { return y; }

  public void setX(int x) { this.x = x; }

  public void setY(int y) { this.y = y; }

  public void addToX(int num) { this.x += num; }

  public void addToY(int num) { this.y += num; }
}
```

Χρήση

```
Point p1 = new Point(0, 0); //στιγμιοτύπιση
p1.addToX(10); //αποστολή μηνύματος (επίκληση μεθόδου)
Point p2 = new Point(10, 0);
assert(p1.getX() == p2.getX());
assert(p1 instanceof Point);
assert(p2 instanceof Point);
```

Παράδειγμα (Javascript < ES6)

```
var Point = function(x, y) { //constuctor
  this.x = x;
  this.y = y;
  this.getX = function() { return this.x; }
  this.getY = function() { return this.y; }
  this.setX = function(x) { this.x = x; }
  this.setY = function(y) { this.y = y; }
  this.addToX = function(num) { this.x += num; }
  this.addToY = function(num) { this.y += num; }
}
```

Καλύτερο Παράδειγμα (Javascript < ES6)

```
var Point = function(x, y) { //constuctor
  this.x = x;
  this.y = y;
}

Point.prototype.getX = function() { return this.x; }

Point.prototype.getY = function() { return this.y; }

Point.prototype.setX = function(x) { this.x = x; }

Point.prototype.setY = function(y) { this.y = y; }

Point.prototype.addToX = function(num) { this.x += num; }

Point.prototype.addToY = function(num) { this.y += num; }
```

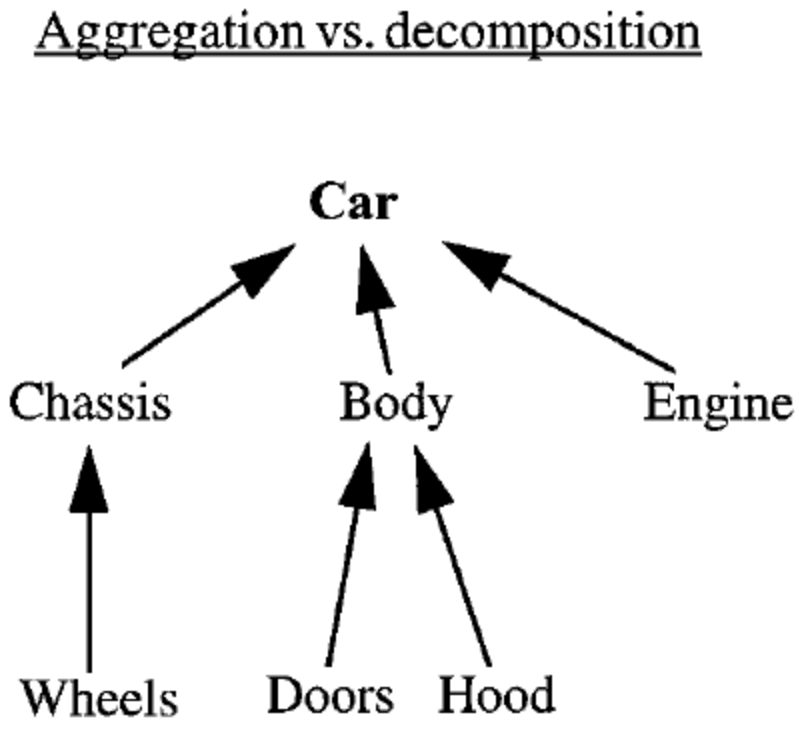
Παράδειγμα (Javascript >= ES6)

```
class Point {

  constructor(x, y) {
    this.x = x;
    this.y = y;
  }

  get x() { return this.x; }

  get y() { return this.y; }

  set x(x) { this.x = x; }

  set y(y) { this.y = y; }

  addToX(num) { this.x += num; }

  addToY(num) { this.y += num; }
}
```

Χρήση

```
var p1 = new Point(0, 0);
p1.addToX(10);
var p2 = new Point(10, 0);
assert p1.getX() == p2.getX();
assert (p1 instanceof Point);
assert (p2 instanceof Point);
```

2. Aggregation - Decomposition



A. Taivalsaari

Aggregation - Decomposition

- Συνάθροιση (ή σύνθεση) επιμέρους εννοιών για τη σύσταση μιας νέας ξεχωριστής έννοιας.
- Σχέσεις μέρους-όλου (part-of).

Παράδειγμα (Java)

Ένα αντικείμενο περιέχει (συντίθεται) από άλλο αντικείμενα

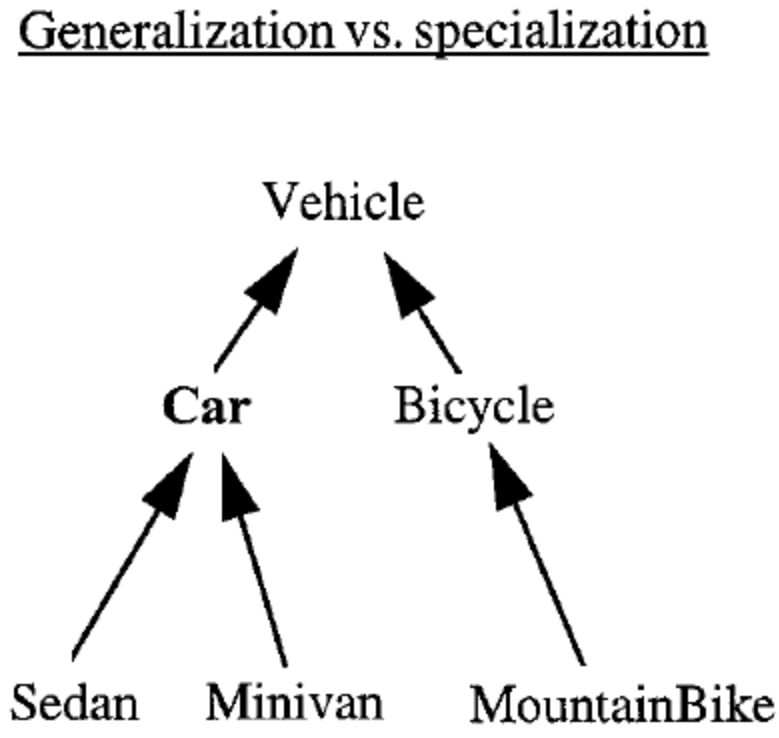
```
class Line {
  private Point first;
  private Point second;

  public Line(Point first, Point second) {
    this.first = first;
    this.second = second;
  }

  Point getFirstPoint() { return first; }

  Point getSecondPoint() { return second; }
}
```

Παράδειγμα (ES6)

Ένα αντικείμενο περιέχει (συντίθεται) από άλλο αντικείμενα

```
class Line {

  constructor(first, second) { //Points
    this.first = first;
    this.second = second;
  }

  getFirstPoint() { return this.first; }

  getSecondPoint() { return this.second; }
}
```

3. Generalization - Specialization



A. Taivalsaari

Generalization - Specialization

- Σχέση μεταξύ κλάσεων.
- Η γενική κλάση συγκεντρώνει τα κοινά στοιχεία όλων των εξειδικεύσεών της.
- Η κληρονομικότητα είναι ο κατεξοχήν μηχανισμός υλοποίησης της εξειδίκευσης.

Κληρονομικότητα

- Ένα αντικείμενο κληρονομεί τα πεδία ή/και τις μεθόδους των "προγόνων" του
- Στην πράξη έχουμε:
  - Κληρονομικότητα για εξειδίκευση (specialization)
  - Κληρονομικότητα για επαναχρησιμοποίηση (reuse)

Παράδειγμα (Java)

```
class Arrow extends Line {

  enum Direction{FIRST_TO_SECOND, SECOND_TO_FIRST};

  private Direction d;

  public Arrow(Point p1, Point p2, Direction d) {
    super(p1, p2);
    this.d = d;
  }

  public void toggleDirection() {
    if (d == Direction.FIRST_TO_SECOND)
      d = Direction.SECOND_TO_FIRST;
    else
      d = Direction.FIRST_TO_SECOND;
  }
}
```

Χρήση

```
Point p1 = new Point(0, 0);
Point p2 = new Point(10, 10);
Arrow a = new Arrow(p1, p2, Arrow.Direction.FIRST_TO_SECOND);
assert(a instanceof Arrow); //Προφανώς
assert(a instanceof Line);  //Επίσης -- πολυμορφισμός
assert(a.getFirstPoint().getX() == 0); //Κληρονομικότητα
```

Παράδειγμα (Javascript < ES6)

```
var Line = function() { //Line constructor
  ...
}

var Direction = {
  FIRST_TO_SECOND: 1,
  SECOND_TO_FIRST: 2
}

var Arrow = function(p1, p2, d) {
  Line.call(this, p1, p2); //call the Line constructor
  this.d = d;
}

Arrow.prototype = new Line(); //"Inherit" from Line
Arrow.prototype.constructor = Arrow; //Just to make sure
Arrow.prototype.toggleDirection = function() {
  if (this.d == Direction.FIRST_TO_SECOND)
    this.d = Direction.SECOND_TO_FIRST;
  else
    this.d = Direction.FIRST_TO_SECOND;
}
```

Παράδειγμα (Javascript >= ES6)

```
class Line {
  ...
}

var Direction = {
  FIRST_TO_SECOND: 1,
  SECOND_TO_FIRST: 2
}

class Arrow extends Line {

  constructor(p1, p2, d) {
    super(p1, p2);
    this.d = d;
  }

  toggleDirection() {
    if (this.d == Direction.FIRST_TO_SECOND)
      this.d = Direction.SECOND_TO_FIRST;
    else
      this.d = Direction.FIRST_TO_SECOND;
  }
```

Χρήση

```
var p1 = new Point(0, 0);
var p2 = new Point(10, 10);
var a = new Arrow(p1, p2, Direction.FIRST_TO_SECOND);
assert(a instanceof Arrow);
assert(a instanceof Line);
assert(a.getFirstPoint().getX() == 0);
```

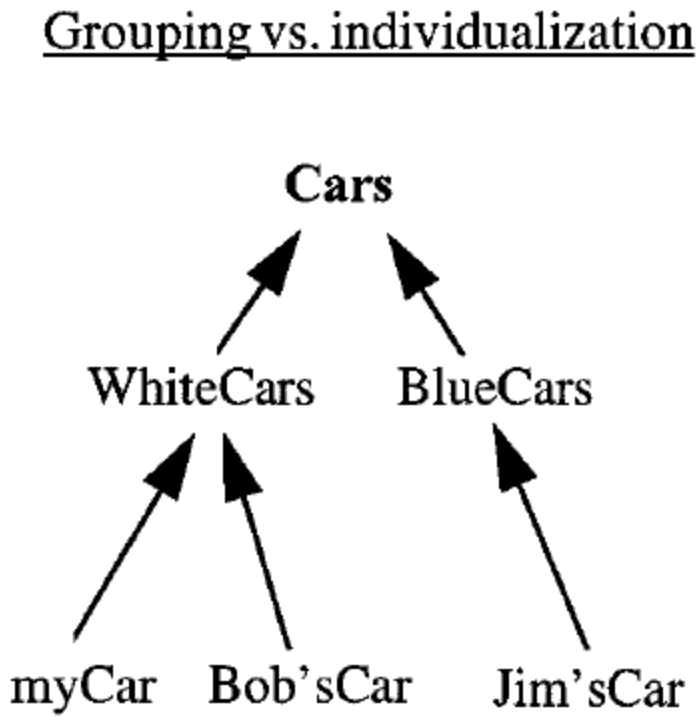
Private state/behavior

- Στο (απλό) παράδειγμά μας:
  - Η κλάση Point ενθυλακώνει δύο ακεραίους
  - Η κλάση Line ενθυλακώνει δύο Point αντικείμενα
  - Η κλάση Arrow ενθυλακώνει δύο Point αντικείμενα και μια διεύθυνση
- Μηχανισμός απόκρυψης πληροφορίας (information hiding)

Private state σε Javascript

```
var Line = function(x, y) {
  var state = {
    x:x,
    y:y
  };

  this.getX = function() {
    return state.x
  }

  ...
};
```

4. Grouping - Individualization



A. Taivalsaari

Ομαδοποίηση - Διαχωρισμός

- Ομαδοποίηση αντικειμένων με βάση κάποιο extensional και όχι intensional χαρακτηριστικό.
- Παραδείγματα:
  - Αγαπημένα του χρήστη (user favorites)
  - Πιο πρόσφατα / πιο δημοφιλή
  - Αποτελέσματα μιας αναζήτησης

Πολυμορφισμός

Παροχή μιας κοινής διεπαφής για αντικείμενα διαφορετικών τύπων

ή

Ένα αντικείμενο μπορεί να έχει πολλούς τύπους (πολλές συμπεριφορές)

Είδη πολυμορφισμού

Universal polymorphism

- Parametric
- Inclusion

Ad-hoc

- Overloading
- Coercion

Λίστα αναγνωσμάτων

Luca Cardelli, Peter Wegner, "On Understanding Types, Data Abstraction, and Polymorphism", ACM Computing Surveys, Vol 17 n. 4, pp 471-522, December 1985.

Ad-hoc πολυμορφισμός (Overloading)

Operator overloading

```
int x = 3 + 5;
String s = name + " " + surname;
```

Method overloading

```
class Foo {
  void doSomething(A a) { ... }
  void doSomething(A a, B b) { ... }
}

class Bar extends Foo {
  void doSomething(C c, D d) { ... }
}
```

Ad-hoc πολυμορφισμός (Coercion)

Type coercion

```
double x = 1;
//The int constant is converted to double automatically

double avg, sum;
int count;
...
avg = sum / count;
//The int count is coverted to double automatically
//before applying the division
```

Παραμετρικός πολυμορφισμός

Generics

```
class Cache<K, V> {
  private final Map<K, V> cache = new HashMap<>();
  public synchronized void put(K key, V value) {
    cache.put(key, value);
  }

  public synchronized V get(K key) {
    return cache.get(key);
  }
}
```

```
Cache<String, Line> lineLabels = new Cache<>();
cache.put("First line", someLine);
cache.put("Second line", someOtherLine);
```

Πολυμορφισμός υπο-τύπων (inclusion polymorphism)

```
interface Shape {
  double getArea();
}

class Rectangle implements Shape {
  private double width;
  private double height;
  public double getArea() {
    return width * height;
  }
}

class Circle implements Shape {
  private double radius;
  public double getArea() {
    return Math.PI * radius * radius;
  }
}
```

```
class AreaPrinter {
  public static void print(Shape s) {
    System.out.println(s.getArea());
  }
}

Rectangle r = new Rectangle(2.0, 3.0);
Circle c = new Circle(1.0);

AreaPrinter.print(r); // 6.0
AreaPrinter.print(c); // 3.14
```

Δυναμική αποστολή μηνυμάτων και αργή δέσμευση

```
interface Computation {
  String getName();
  void compute();
}

abstract class ComputationBase implements Computation {
  public String getName() {
    return getClass().getName();
  }
}
```

```
class SimpleComputation extends ComputationBase {
  public void compute() {
    System.out.println("Done computing");
  }
}

class SimpleComputation2 extends SimpleComputation {
  public String getName() {
    return "Simple2"
  }
}
```

```
class ListOfComputations extends ComputationBase {
  private final List<Computation> computations;

  public ListOfComputations(Computation... computations) {
    this.computations = Arrays.asList(computations);
  }

  public void compute() {
    for(Computation c: computations) {
      System.out.println("Computing: " + c.getName());
      c.compute();
    }
  }
}
```

```
Computation c1 = new SimpleComputation();
Computation c2 = new SimpleComputation2();
Computation list = new ListOfComputations(c1, c2);
System.out.println(list.getName());
list.compute();

//Output
ListOfComputations
Computing: SimpleComputation
Done computing
Computing: Simple2
Done computing
```