Παν. Ιωαννίνων
Τμήμα Μηχ. Η/Υ & Πληροφορικής

Ανάπτυξη Λογισμικού

(διαφάνειες του μαθήματος)

Πάνος Βασιλειάδης
pvassil@cs.uoi.gr

www.cs.uoi.gr/~pvassil/courses/sw_dev/

ΠΛΥ 308 – ΑΝΑΠΤΥΞΗ ΛΟΓΙΣΜΙΚΟΥ

ΠΑΝΑΓΙΩΤΗΣ ΒΑΣΙΛΕΙΑΔΗΣ

ΠΕΡΙΕΧΟΜΕΝΟ ΤΟΥ ΜΑΘΗΜΑΤΟΣ

Ως σχεδιαστές λογισμικού και προγραμματιστές σκοπό έχουμε να υλοποιούμε λογισμικό για την επίλυση προβλημάτων με βάση κάποιες δοσμένες απαιτήσεις. Η ανάπτυξη του λογισμικού αποτελείται από μια ακολουθία από διακριτές φάσεις.

- **Ανάλυση απαιτήσεων:** Καταγραφή με συγκροτημένο τρόπο της λειτουργικότητας την οποία θα πρέπει το σύστημα να προσφέρει
- **Σχεδίαση:** Από ποια βασικά δομικά στοιχεία θα αποτελείται το λογισμικό – π.χ., δομές δεδομένων, συναρτήσεις, κλάσεις
- **Υλοποίηση / Κατασκευή:** Αποτύπωση της σχεδίασης σε κώδικα και υλοποίηση σε κάποια γλώσσα προγραμματισμού (Java, C, C++, Python, … to name a few)
- **Έλεγχος:** Αποσφαλμάτωση και διακρίβωση της ορθότητας του προγράμματος
- **Εγκατάσταση**
- **Συντήρηση**

Στην πραγματική ζωή, το μείζον πρόβλημα δεν είναι η αποσπασματική υλοποίηση ενός μόνο αλγορίθμου ή η μετατροπή ενός αλγορίθμου σε κώδικα αλλά η κατασκευή, **με συγκροτημένο τρόπο,** ενός **ολοκληρωμένου** συστήματος που προσφέρει διάφορες λειτουργίες.

Ο σκοπός του μαθήματος της Ανάπτυξης Λογισμικού είναι διττός και αφορά στην:

- παρουσίαση θεμελιωδών θεμάτων σχεδίασης και ανάπτυξης εφαρμογών λογισμικού
- πρακτική τριβή των φοιτητών, μέσω προγραμματιστικής εργασίας (project) με πραγματικά προβλήματα που ανακύπτουν στα πλαίσια της ανάπτυξης μιας ευμεγέθους εφαρμογής οργανωμένης σε επί μέρους στάδια: ανάλυση απαιτήσεων, σχεδίαση, υλοποίηση και έλεγχος.

Οι λεπτομέρειες της διεξαγωγής του μαθήματος μπορούν να αναζητηθούν στο δικτυακό τόπο του μαθήματος: **`http://www.cs.uoi.gr/~pvassil/courses/sw_dev/`**

ΔΙΔΑΚΤΕΑ ΎΛΗ

Το αντικείμενο του μαθήματος οργανώνεται στις παρακάτω ενότητες:

1. Βασικές αρχές αντικειμενοστρεφούς προγραμματισμού
2. Διαχείριση απαιτήσεων και use cases
3. Διαγράμματα UML
4. Ανάλυση και σχεδίαση αντικειμενοστρεφούς λογισμικού
5. Βασικές αρχές αντικειμενοστρεφούς σχεδίασης
6. Έλεγχος λογισμικού

ΒΙΒΛΙΟΓΡΑΦΙΑ

- Αντικειμενοστρεφής Σχεδίαση: UML, Αρχές, Πρότυπα Και Ευρετικοί Κανόνες, Α. Χατζηγεωργίου, Κλειδάριθμος, ISBN 960-209-882-1. Κωδικός Βιβλίου στον Εύδοξο: 13600
- Ανάπτυξη Προγραμμάτων σε Java: αφαιρέσεις, προδιαγραφές, και αντικειμενοστρεφής σχεδιασμός, B. Liskov and J. Guttag, Κλειδάριθμος, ISBN 978-960-461-063-1. Κωδικός Βιβλίου στον Εύδοξο: 13596

Επανάληψη βασικών αρχών του αντικειμενοστρεφούς προγραμματισμού

Ανάπτυξη Λογισμικού (Software Development)

www.cs.uoi.gr/~pvassil/courses/sw_dev/

ΜΥΥ301/ΠΛΥ 308

Αντικείμενο της ενότητας

- Η ενότητα στόχο έχει να συνοψίσει και να σας υπενθυμίσει τις βασικές έννοιες του αντικειμενοστρεφούς παραδείγματος που θα αποτελέσει το βασικό εργαλείο μας στο μάθημα αυτό.

Εκπαιδευτικοί στόχοι

- Τελειώνοντας αυτή την ενότητα, θα πρέπει οι βασικές έννοιες να είναι ξεκάθαρες στο μυαλό σας και ως προς την ουσία τους και ως προς το πώς τις αξιοποιούμε στην γλώσσα Java
- Οι εν λόγω βασικές έννοιες είναι απολύτως προαπαιτούμενες για να μπορείτε να ανταποκριθείτε στην ανάπτυξη λογισμικού και εδώ και στην καριέρα σας
- Αν έχετε δυσκολίες: η στιγμή να εξασκηθείτε πρακτικά σε κώδικα είναι τώρα!

Οι πυλώνες του αντικειμενοστρεφούς προγραμματισμού…

- Κλάσεις, αντικείμενα και ενθυλάκωση
- Πολυμορφισμός
- Σύνθετα αντικείμενα και συλλογές αντικειμένων
- Ιεραρχίες κλάσεων

Για μια ιστορική αναφορά, βλ. το σχόλιο του Alan Kay https://qr.ae/pNP89E

Η ουσία του αντικειμενοστρεφούς προγραμματισμού

- Η αρχική ιδέα ήταν «υπολογιστικές οντότητες που μιλάνε μεταξύ τους με κάποιο πρωτόκολλο»
- Ο αντικειμενοστρεφής προγραμματισμός έγινε το κεντρικό εργαλείο ανάπτυξης λογισμικού σε ομάδες – για να καταλάβετε ό,τι λέμε, βοηθά να σκέφτεστε ότι κάθε κλάση υλοποιείται κι από διαφορετικό άνθρωπο…
- **Δουλεύεις μαζί με άλλους για να φτιάξετε ένα puzzle**
  - **Ο καθένας φτιάχνει το κομμάτι του μόνος του (άρα πρέπει να μπορείς να το φτιάξεις μόνος σου)**
  - **Τα κομμάτια μετά πρέπει να ταιριάζουν (άρα πρέπει να έχουμε μια συμφωνία στο πώς θα ταιριάξουν μετά)**

ΚΛΑΣΕΙΣ ΚΑΙ ΕΝΘΥΛΑΚΩΣΗ

Κλάσεις και αντικείμενα

- Το βασικό στοιχείο σχεδίασης και δόμησης του αντικειμενοστρεφούς κώδικα είναι η κλάση (class)
- Μία κλάση είναι ένα καλούπι από το οποίο παράγονται αντικείμενα (objects), τα οποία:
  - Αναπαριστούν ομοειδείς οντότητες του πραγματικού κόσμου
  - Έχουν ίδια δομή

7

Κλάσεις και αντικείμενα

- Κάθε κλάση είναι μια ενότητα λογισμικού, η οποία σχεδιάζεται με σκοπό
  - Να διευκολύνει την συνεργασία των προγραμματιστών
  - Να διευκολύνει την επαναχρησιμοποίηση έτοιμου κώδικα από άλλους
  - Να συμπτύξει στο ίδιο σημείο ενός project (στο ίδιο αρχείο κατά βάση) κώδικα και δεδομένα
- Για το σκοπό αυτό:
  - Αντί για να διαχωρίζουμε δεδομένα και κώδικα, ο σχεδιαστής μιας κλάσης εξάγει προς τους υπόλοιπους προγραμματιστές μία δημόσια διαπροσωπεία (public interface), η οποία λέει στους υπόλοιπους προγραμματιστές, τι λειτουργίες μπορεί να προσφέρουν τα αντικείμενα της κλάσης αυτής
  - Έτσι, εκτός από τα δεδομένα που κουβαλά ένα αντικείμενο (τα οποία αξιοποιούν τα πεδία της κλάσης) εκτελεί και συναρτήσεις (οι οποίες εκτελούν τις μεθόδους της κλάσης)

8

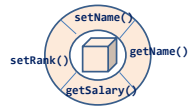
Προσοχή: αντικειμενοστρεφής σχεδίαση

- Η κλάση είναι το καλούπι, αλλά τα αντικείμενα είναι αυτά που εκτελούν τις λειτουργίες
- Όταν σχεδιάζουμε τον κώδικα, σκεπτόμαστε πώς κάποια αντικείμενα που θα δημιουργηθούν κατά την εκτέλεση του προγράμματος θα επιτελούν λειτουργίες!
  - Σχεδιάζουμε, λοιπόν, έχοντας υπόψη λειτουργίες που θα επιτελούνται και όχι δομές δεδομένων!

9

Ενθυλάκωση

- Η αρχή της **ενθυλάκωσης** (encapsulation):
  - σε κάθε κλάση, ο προγραμματιστής που την υλοποίησε, διαχωρίζει:
    - ποια πεδία και μέθοδοι (**public**) μπορούν να χρησιμοποιηθούν από όλο τον υπόλοιπο κώδικα
    - ποια πεδία και μέθοδοι (**private**) μπορούν να χρησιμοποιηθούν μόνο μέσω κλήσης των public μεθόδων της κλάσης…
  - μια συνήθης πρακτική την οποία πρέπει να ακολουθείτε και εσείς είναι ότι **όλα τα πεδία τα δηλώνουμε private**



Κατασκευαστές αντικειμένων

- Αν και η ενθυλάκωση επιτρέπει στους προγραμματιστές που αξιοποιούν τις λειτουργίες που προσφέρει μια κλάση να το κάνουν με συγκροτημένο τρόπο μέσω των δημόσιων μεθόδων, η ενθυλάκωση από μόνη της δεν εξασφαλίζει τη σωστή δημιουργία των αντικειμένων
- … έτσι μπορεί να υπάρχουν σφάλματα κακής αρχικοποίησης …

- Δήλωση **constructors** στις κλάσεις μας
  - μέθοδοι που **καλούνται αυτόματα** όταν δημιουργείται ένα αντικείμενο
  - Πιθανώς περισσότεροι του ενός ανάλογα με τις παραμέτρους αρχικοποίησης
  - Default constructor (άνευ ορισμάτων)

Στατικά πεδία και μέθοδοι

- **Στατικά πεδία** είναι πεδία των οποίων η τιμή χαρακτηρίζει μια ολόκληρη κλάση αντικειμένων και όχι μόνο κάποιο συγκεκριμένο αντικείμενο της κλάσης. Δηλώνοντας το πεδίο ως **`static`** μπορούμε
  - να υπάρχει ίδια τιμή για όλα τα αντικείμενα της κλάσης.
  - και αν μεταβληθεί αυτή η τιμή για ένα αντικείμενο να μεταβάλλεται αυτόματα και για όλα τα υπόλοιπα….

- Οι **στατικές μέθοδοι** είναι μέθοδοι μιας κλάσης που μπορούν να κληθούν αυτόνομα – δεν χρειάζεται να κληθούν πάνω σε ένα αντικείμενο της κλάσης και χρησιμοποιούνται για την πρόσβαση σε static πεδία της κλάσης.

- Μην το παρακάνετε με τα static πεδία! Το ότι υπάρχει μια δυνατότητα δε σημαίνει ότι πρέπει υποχρεωτικά να τη χρησιμοποιούμε!

ΙΕΡΑΡΧΙΕΣ ΚΛΑΣΕΩΝ

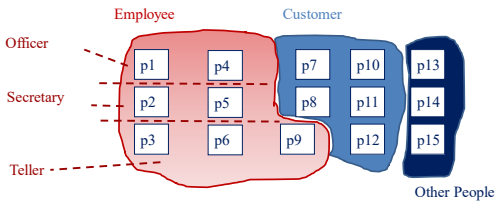
Κληρονομικότητα

- Το πρόβλημα:
  - Σε πολλές εφαρμογές έχουμε διάφορες κλάσεις αντικειμένων οι οποίες έχουν κοινά χαρακτηριστικά/πεδία και μεθόδους
    - πχ. τόσο οι εργαζόμενοι όσο και οι πελάτες μιας εταιρίας χαρακτηρίζονται από ένα όνομα, επίθετο, κλπ….
  - ένα σφάλμα μπορεί να εισαχθεί σε πολλά διαφορετικά σημεία
  - μια αλλαγή θα πρέπει να γίνει σε πολλά διαφορετικά σημεία
  - μια διόρθωση, ένας έλεγχος θα πρέπει να γίνει σε πολλά διαφορετικά σημεία

Κληρονομικότητα

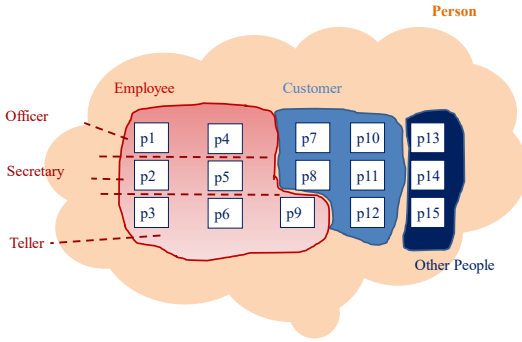
- Γιατί υπάρχει το παραπάνω χαρακτηριστικό?
- Υποψιαζόμαστε ότι οι υπάλληλοι και οι πελάτες μοιράζονται χαρακτηριστικά γιατί όλοι ανήκουν σε ένα ευρύτερο υπερσύνολο, αυτό των ανθρώπων
- **Η σχέση του συνόλου των ανθρώπων με τα σύνολα των πελατών και των υπαλλήλων είναι σχέση υπερσυνόλου!!**
- Σημαντική παρατήρηση: οι πελάτες/υπάλληλοι ΕΊΝΑΙ ΚΑΙ μέλη του συνόλου των ανθρώπων

Εξειδίκευση



16

Εξειδίκευση



17

Κληρονομικότητα

- Για καλύτερη οργάνωση και συντήρηση του κώδικα (για να μην επαναλαμβάνεται ο ίδιος κώδικας πολλές φορές)
  - φτιάχνουμε μια **γενική/βασική κλάση** που να περιλαμβάνει τα κοινά χαρακτηριστικά/πεδία και μεθόδους δύο ή περισσοτέρων κλάσεων και
  - εν συνεχεία **επεκτείνουμε** τη βασική φτιάχνοντας κλάσεις που **κληρονομούν** από αυτή.

Κληρονομικότητα

- Τι σημαίνει επέκταση…
  - Στις **παραγόμενες κλάσεις** δηλώνουμε επιπλέον χαρακτηριστικά και λειτουργίες.
  - Τα αντικείμενα έχουν όλα τα χαρακτηριστικά και λειτουργίες που δηλώνονται στην βασική κλάση,
  - καθώς και τα επιπλέον χαρακτηριστικά και λειτουργίες που δηλώνονται στην παραγόμενη κλάση

Κληρονομικότητα

- Τα αντικείμενα των παραγόμενων κλάσεων συμπεριφέρονται και ως αντικείμενα της βασικής κλάσης (π.χ., στο πέρασμα παραμέτρων)
- Η επέκταση ονομάζεται και σχέση **IS_A** («είναι»).

Κληρονομικότητα – Πεδία και Μέθοδοι

- πεδία και μέθοδοι **public**: ορατά από όλους
  - αλλαγές / διορθώσεις μπορεί να επηρεάσουν τον υπόλοιπο κώδικα
- πεδία και μέθοδοι **private**: ορατά από την υλοποίηση των μεθόδων της κλάσης
  - αλλαγές / διορθώσεις / έλεγχοι συγκεντρώνονται στον κώδικα της κλάσης και δεν επηρεάζουν τον υπόλοιπο κώδικα
- πεδία και μέθοδοι **protected**: ορατά από
  - την υλοποίηση των μεθόδων της κλάσης, και επιπλέον από
  - την υλοποίηση των μεθόδων των κλάσεων που τα κληρονομούν
    - αλλαγές / διορθώσεις / έλεγχοι συγκεντρώνονται στον κώδικα της κλάσης
    - επηρεάζουν τον κώδικα των παραγόμενων κλάσεων
    - δεν επηρεάζουν τον υπόλοιπο κώδικα

Κληρονομικότητα – Πεδία και Μέθοδοι

- πεδία και μέθοδοι **package-private**: ορατά από όλους μέσα στο package της κλάσης
  - Δεν χρειάζεται modifier (αν δεν πείτε τπτ για την μέθοδο ή το πεδίο, είναι package-private)
  - Αν οι υποκλάσεις είναι σε άλλο package δεν βλέπουν την εν λόγω μέθοδο ή το πεδίο)
  - Δημιουργούν αρκετά προβλήματα αν αρχίσουν να υπεισέρχονται υποπακέτα στο πακέτο
- Συνήθως, η χρήση αυτών των μεθόδων είναι για utility methods to be invoked by the classes of a thematically focused package. Ουσιαστικά, περιορίζει τους clients μιας μεθόδου, αυστηρά μέσα σε ένα package. Αλλά είναι πολύ εύκολο να κακοποιηθεί η σκοπιμότητα μιας τέτοιας επιλογής.
- Στα πλαίσια του μαθήματος:
  - **ΑΠΑΓΟΡΕΥΕΤΑΙ Η ΧΡΗΣΗ package-private μεθόδων ή πεδίων παντοιοτρόπως** (ώστε να μάθετε να ορίζετε ορατότητες εσκεμμένα και να μην υπάρχει άλλοθι για την έλλειψη ορισμού ορατότητας) =>
  - Οπουδήποτε συναντήσω package-private visibility μετρά για λάθος

22

ΠΟΛΥΜΟΡΦΙΣΜΟΣ

23

Η συντήρηση κάνει τον κόσμο να γυρνάει …

- Η συντήρηση είναι το 60%-80% της προσπάθειας σε ένα software project και πολύ συχνά αφορά την επέκταση του υπάρχοντος κώδικα με επιπλέον λειτουργικότητα
- Γενικά είναι επιθυμητό είναι να έχουμε τη δυνατότητα να προσθέτουμε νέες δυνατότητες σε ένα πρόγραμμα χωρίς να χρειαστεί να αλλάξουμε δραστικά τον υπάρχοντα κώδικα …
- Πώς γίνεται αυτό ??
  - συνδυάζουμε **κληρονομικότητα** & **πολυμορφισμό**

24

Βασικοί πυλώνες του πολυμορφισμού

- **method overloading (υπερφόρτωση μεθόδων)**
  - σε μια κλάση ορίζονται 2 ή περισσότερες μέθοδοι με το ίδιο όνομα και διαφορετικές παραμέτρους – διαφορετικό πρωτότυπο (το είδαμε στην περίπτωση πολλαπλών constructors)
- **method overriding (επανακαθορισμός μεθόδων)**
  - **η μέθοδος μιας βασική κλάσης ορίζεται ξανά στην παραγόμενη κλάση με νέα υλοποίηση και το ίδιο πρωτότυπο**
  - **με αυτή τη τεχνική μπορούμε να πετύχουμε τον επεκτασιμότητα του κώδικα!!**

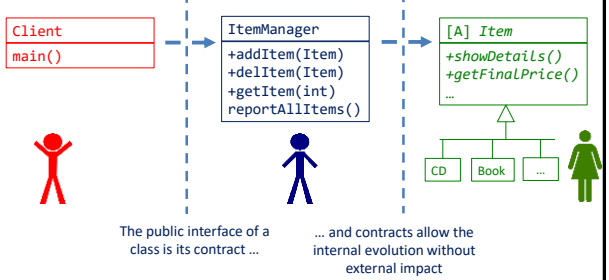
25

Αφηρημένες κλάσεις

- Δηλώνοντας μια κλάση ως αφηρημένη (abstract)
  - μια κλάση ονομάζεται αφηρημένη αν περιέχει τουλάχιστον μια αφηρημένη μέθοδο που δεν περιλαμβάνει υλοποίηση
    - Η κλάση δηλώνεται public abstract class MyAbstractClass
    - με τη μέθοδο public abstract returnType methodName();
  - η δημιουργία αντικειμένων αφηρημένης κλάσης δεν επιτρέπεται από τον compiler => υποχρεωτικά, θα κατασκευαστούν αντικείμενα των παραγόμενων
  - **Contract: η μη υλοποίηση αφηρημένων μεθόδων δεν επιτρέπεται από τον compiler**
  - η αφηρημένη κλάση λειτουργεί σαν καλούπι για την κατασκευή παραγόμενων που προσφέρουν **εναλλακτικές υλοποιήσεις στις αφηρημένες μεθόδους**
  - **Ουσιαστικά, ο πολυμορφισμός επιτυγχάνεται διότι ο υπόλοιπος κώδικας (πλην των new()) γράφεται σε σχέση με την βασική, abstract class, για την οποία ο μεταφραστής εγγυάται ότι οι υποκλάσεις της θα παρέχουν υλοποιήσεις για τις αφηρημένες μεθόδους => αναλόγως τι θα έχει γίνει new() θα εκτελεστεί και ο αντίστοιχος κώδικας**

26

Puzzles are made from contracts



The public interface of a class is its contract …

… and contracts allow the internal evolution without external impact

27

Interfaces

- Πρακτικά, **ένα interface ορίζει ένα σύνολο δημόσιων αφηρημένων μεθόδων τις οποίες, οι κλάσεις που υλοποιούν το interface υποχρεούνται να υλοποιήσουν και να παρέχουν**
  - μπορείτε να το σκέφτεστε ως **συμβόλαιο**, με τον interpreter/compiler στο ρόλο του δικαστή/συμβολαιογράφου που εντοπίζει/απαγορεύει παραβιάσεις του συμβολαίου
- **Η ομοιότητα αφηρημένων κλάσεων και interfaces αφορά στην υποστήριξη του πολυμορφισμού: αμφότερα λειτουργούν ως συμβόλαια μεθόδων που θα υλοποιήσουν άλλες κλάσεις και επιτρέπουν τη χρήση πολυμορφισμού από τον κώδικα που χρησιμοποιεί το συμβόλαιο, χωρίς γνώση του ποια συγκεκριμένη κλάση υλοποιεί τις αφηρημένες μεθόδους**
- Η διαφορά των interfaces από τις abstract classes είναι
  - Κατά βάση μεθοδολογική: τα interfaces χρησιμοποιούνται ως συμβόλαια για τη σύνδεση υποσυστημάτων, ενώ οι αφηρημένες κλάσεις ως καλούπια για την παραγωγή εναλλακτικών υλοποιήσεων του ίδιου κώδικα
  - Τυπική: οι αφηρημένες κλάσεις δημιουργούν ιεραρχία κλάσεων, ενώ τα interfaces απλά εγγυώνται την τήρηση των συμβολαίων, χωρίς καμία σχέση μεταξύ των κλάσεων που τα υλοποιούν

28

Κληρονομικότητα & Πολυμορφισμός

Πώς μπορούμε να ελαχιστοποιήσουμε τη συντήρηση του κώδικα μέσω του πολυμορφισμού?

1. για κάθε αρμοδιότητα του προγράμματος για την οποία υπάρχει δυνατότητα διαφορετικών εναλλακτικών υλοποιήσεων (πχ διαφορετικές μορφές αποθήκευσης των δεδομένων σε ένα αρχείο) και κατά συνέπεια η δυνατότητα μελλοντικών επεκτάσεων στο πρόγραμμά μας με μια νέα επιτλέον υλοποίηση ορίζουμε μια βασική αφηρημένη κλάση / interface
2. εν συνεχεία για κάθε διαφορετική εναλλακτική κατασκευάζουμε μια παραγόμενη κλάση η οποία υλοποιεί τις public abstract μεθόδους της βασικής κλάσης / interface
3. υλοποιούμε / παραμετροποιούμε τον υπόλοιπο κώδικα χρησιμοποιώντας αναφορές στη βασική αφηρημένη κλάση / interface
   - οι αναφορές αυτές μπορούν να δείχνουν σε αντικείμενα οποιασδήποτε κλάσης προκύπτει από τη βασική / υλοποιεί το interface
   - επομένως τα μόνα σημεία τα οποία πρέπει να αλλαχθούν σε μια μελλοντική επέκταση είναι τα σημεία στο οποία αρχικοποιούνται οι δείκτες αυτοί
   - ο υπόλοιπος κώδικας στον οποίο καλούνται μέθοδοι σε αντικείμενα στα οποία δείχνουν οι δείκτες δεν χρειάζεται αλλαγές

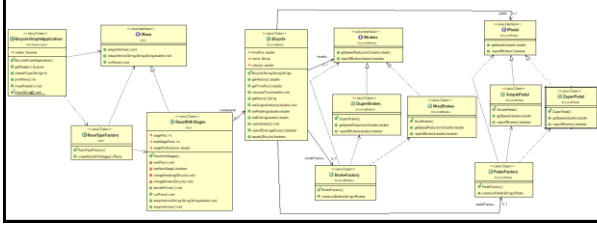
(πολλές ευχαριστίες στον Π. Τσαπάρα για τις διαφάνειες των συλλογών)

COMPOSITE OBJECTS

30

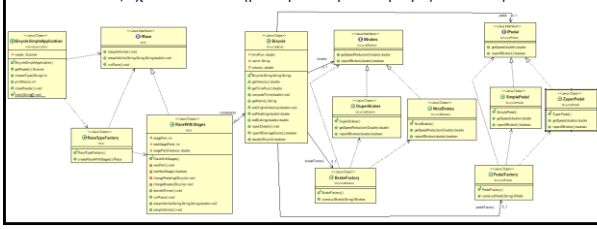
Σύνθετα αντικείμενα

- Γενικώς, η ιδέα ότι ένα αντικείμενο «έχει ως πεδίο» ένα άλλο αντικείμενο είναι ίσως λίγο παραπλανητική
- Η βασική ιδέα είναι ότι εν γένει **τα αντικείμενα συνεργάζονται**



Σύνθετα αντικείμενα

- Ένα ποδήλατο «έχει» ένα σύστημα φρένων και ένα σύστημα πεταλιών
  - Δλδ. Έχει ένα πεδίο τύπου IBreaks και ένα πεδίο τύπο IPedal
- Μια κούρσα «έχει» ποδήλατα που συμμετέχουν
  - Δλδ., έχει ένα σύνολο (με απροσδιόριστο αριθμό) από ποδήλατα



Collaborate = know = “have” other objects as attributes

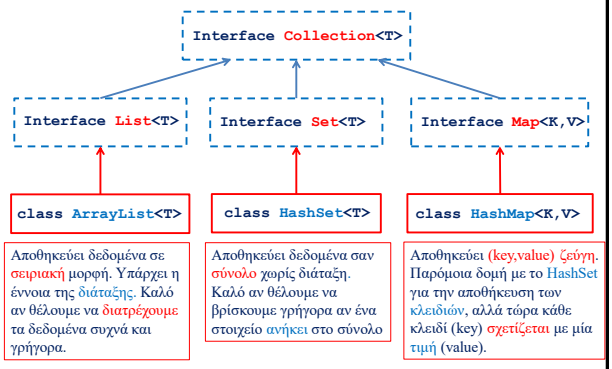
Just one collaborator

```
public class Bicycle {
        private double timeRun;
        private String name;
        private double velocity;
        private IBrakes breaks;
        private IPedal pedal;
...
}
```

More than one

```
public class RaceWithStages
        implements IRace{
  private int stagePart = 0;
  private int totalStageParts = 4;
  private ArrayList<Bicycle> contestants;
...
}
```

Java Collections



Αποθηκεύει δεδομένα σε σειριακή μορφή. Υπάρχει η έννοια της διάταξης. Καλό αν θέλουμε να διατρέχουμε τα δεδομένα συχνά και γρήγορα.

Αποθηκεύει δεδομένα σαν σύνολο χωρίς διάταξη. Καλό αν θέλουμε να βρίσκουμε γρήγορα αν ένα στοιχείο ανήκει στο σύνολο

Αποθηκεύει (key,value) ζεύγη. Παρόμοια δομή με το HashSet για την αποθήκευση των κλειδιών, αλλά τώρα κάθε κλειδί (key) σχετίζεται με μία τιμή (value).

34

ArrayList (JavaDocs link)

- Constructors
  - `ArrayList<T> myList = new ArrayList<T>();`
  - `ArrayList<T> myList = new ArrayList<T>(10); //λιστα με χωρητικότητα 10`
- Μέθοδοι
  - `add(T x)`: προσθέτει το στοιχείο `x` στο τέλος του πίνακα.
  - `add(int i, T x)`: προσθέτει το στοιχείο `x` στη θέση `i` και μετατοπίζει τα υπόλοιπα στοιχεία κατά μια θέση.
  - `remove(int i)`: αφαιρεί το στοιχείο στη θέση `i`
  - `set(int i, T x)`: αλλάζει την τιμή της θέσης `i` με την τιμή `x`
  - `get(int i)`: επιστρέφει την τιμή στη θέση `i`.
  - `size()`: ο αριθμός των στοιχείων του πίνακα.
- Διατρέχοντας τον πίνακα:
  - `ArrayList<T> myList = new ArrayList<T>();`
  - `for(T x: myList){…}`

35

HashSet (JavaDocs link)

- Constructors
  - `HashSet<T> mySet = new HashSet<T>();`
- Μέθοδοι
  - `add(T x)`: προσθέτει το στοιχείο `x` αν δεν υπάρχει ήδη στο σύνολο.
  - `remove(T x)`: αφαιρεί το στοιχείο `x`.
  - `contains(T x)`: boolean αν το σύνολο περιέχει το στοιχείο `x` ή όχι.
  - `size()`: ο αριθμός των στοιχείων στο σύνολο.
  - `isEmpty()`: boolean αν έχει στοιχεία το σύνολο ή όχι.
  - `Object[] toArray()`: επιστρέφει πίνακα με τα στοιχεία του συνόλου (επιστρέφει πίνακα από Objects – χρειάζεται casting μετά).
- Διατρέχοντας τα στοιχεία του συνόλου:
  - `HashSet<T> mySet = new HashSet<T>();`
  - `for(T x: mySet){…}`

36

HashMap (JavaDocs link)

- Constructors
  - `HashMap<K,V> myMap = new HashMap<K,V>();`
- Μέθοδοι
  - `put(K key, V value)` : προσθέτει το ζευγάρι (`key,value`) (δημιουργεί μία συσχέτιση)
  - `V get(K key)` : επιστρέφει την τιμή για το κλειδί `key`.
  - `remove(K key)` : αφαιρεί το ζευγάρι με κλειδί `key`.
  - `containsKey(K key)` : boolean αν το σύνολο περιέχει το κλειδί `key` ή όχι.
  - `containsValue(V value)` : boolean αν το σύνολο περιέχει την τιμή `value` ή όχι. (αργό)
  - `size()` : ο αριθμός των στοιχείων (ζεύγη από κλειδιά-τιμές) στο map.
  - `isEmpty()` : boolean αν έχει στοιχεία το map ή όχι.
  - `Set<K> keySet()`: επιστρέφει ένα Set με τα κλειδιά.
  - `Collection<V> values()`: επιστρέφει ένα Collection με τις τιμές

37

Java code conventions

Construct	Form. Syntax	Essence	Example
package	lowercase	**ουσιαστικό** που αφορά στα περιεχόμενα του πακέτου	`mainengine, dataload`
Class	CamelCase	**Ουσιαστικό** που περιγράφει τι αναπαριστά η κλάση στον πραγματικό κόσμο, ή τι ρόλο έχει στον κώδικα	`MainEngine, DataLoader`
Interface	CamelCase	**Επίθετο** (συχνά) ή **ουσιαστικό** που εξηγεί τι ρόλο μπορεί να φέρει εις πέρας όποια κλάση υλοποιεί το interface (frequently starts with an I)	`IReporter, ISortedList`
method	mixedCase	**ρήμα** (ενεργητικό) που περιγράφει τι κάνει η μέθοδος. Σύνταξη: **ρήμαΠεριγραφήΑντικειμένου** ΔΕΝ ΒΑΡΙΟΜΑΣΤΕ ΝΑ ΤΟ ΔΩΣΟΥΜΕ ΣΩΣΤΑ!	`getMonetarySum(), produceTotalEuroAmount()`
variable	mixedCase	**ουσιαστικό** που περιγράφει επαρκώς το ρόλο του πεδίου στην κλάση Κατ' αντιστοιχία με τις μεθόδους!	`receivedPackages, euroAmountSpent`
CONSTANT	UPPERCASE	σαν variable, αλλά για να διακρίνεται ότι είναι σταθερά, όλα κεφαλαία. Συχνά ξεκινά και με _. Το μόνο construct where _ is allowed	`_TOTAL_NUM_OWNERS`

38

Java Code Conventions

- Οι κανόνες αναμένεται να τηρηθούν με θρησκευτική ευλάβεια
- Δεν είναι αποδεκτές παραβιάσεις, ούτε του format, ούτε της ουσίας των κανόνων ονοματοδοσίας
- Το ίδιο θα συμβεί σε οποιοδήποτε software project θα κληθείτε να εργαστείτε, παντού στον κόσμο
- Οι κανόνες όπως είχαν δοθεί από τη Sun – subsequently, Oracle:

https://www.oracle.com/technetwork/java/codeconventions-150003.pdf

39

ΕΝ ΚΑΤΑΚΛΕΊΔΙ

Οι πυλώνες του αντικειμενοστρεφούς προγραμματισμού…

- Κλάσεις, αντικείμενα και ενθυλάκωση
- Πολυμορφισμός
- Σύνθετα αντικείμενα και συλλογές αντικειμένων
- Ιεραρχίες κλάσεων

Είναι απολύτως απαραίτητο να κατανοείτε και να μπορείτε να χειριστείτε αυτά τα θεμέλια!

Hands-on

- ΔΕΙΤΕ
  - Δείτε ασκήσεις επανάληψης στην Ενότητα 2 του υλικού
  - Δείτε τον κώδικα των μικρών projects που περιγράφονται εκεί
- ΚΑΝΤΕ…
  - … ΜΟΝΟΙ ΣΑΣ την υλοποίηση για κάτι(όλα) από αυτά ΠΛΗΡΩΣ, all the way to the end
  - … ΛΑΘΗ

Είναι απολύτως αποδεκτό και κατανοητό ότι για κάποιους, ο κώδικας, και δη ο αντικειμενοστρεφής, είναι στρεσογόνος και πιεστικός. Και δεν μπορείς να κρυφτείς από το αποτέλεσμα.

```
Repeat{
    γράψτε λίγο (όχι κατεβατά);
    κάντε compile;
    repeat{
        διορθώστε;
    }(μέχρι να μην υπάρχουν λάθη);
    ελέγξτε το (προσωρινό) αποτέλεσμα;
}(until done).
```

http://www.cs.uoi.gr/~pvassil/courses/sw_dev/readings.html

ΣΗΜΕΙΩΣΕΙΣ

Ακολουθεί ένα παράδειγμα στο οποίο μπορείτε να δείτε πώς δουλεύει:

- Η κατασκευή αντικειμένων (constructors)
- Η ενθυλάκωση εργασιών σε μεθόδους που κάνουν τη δουλειά μας(encapsulation)
- Η συνάθροιση και η σύνθεση αντικειμένων μέσω συλλογών (aggregation /composition of objects in object collections)
- Οι στατικές μεταβλητές

Το παράδειγμα δείχνει την οργάνωση ενός κειμένου σε ενότητες, καθώς και τη συσχέτιση κειμένων με τα θέματα τα οποία πραγματεύεται. Μια απλή main απλώς επιδεικνύει την κατασκευή και χρήση σχετικών αντικειμένων.

```
import java.util.ArrayList;
import java.util.Iterator;

public final class Document {
  private static final String OWNER = "Univ. Ioannina";
  private String docName;
  private Topic docTopic;
  private ArrayList<Section> sections;
  private int totalNumPages = 0;

  public Document(){
    docName = "noName"; docTopic = null;
    sections = new ArrayList<Section>();
  }
  public Document(String aName, Topic aTopic){
    docName = aName; docTopic = aTopic;
    sections = new ArrayList<Section>();
  }

  public static String getOwner(){return OWNER;}

  public void addSection(Section aSection){
    sections.add(aSection);
    System.out.println("New section added, total # sections: " + sections.size());
  }

  public void showDocumentOverview(){
    System.out.println(OWNER + " has ths nice document titled: " + docName + " with the topic: " + docTopic.getTitle() + "\n");
  }

  public double computeStats(double hoursSpentPerPage){
    double totalEffortToRead = 0;

    Iterator <Section> si = sections.iterator();
    Section s = null;
    while (si.hasNext()){
      s = si.next();
      totalNumPages += s.getNumPages();
    }
    totalEffortToRead = totalNumPages * hoursSpentPerPage;
    return totalEffortToRead;
  }
```

Useless getters

```
// public String getName(){return docName;}
// public Topic getTopic(){return docTopic;}
// public int getNumSections(){return sections.size();}
```

```
  public void showDocumentDetails(){
    double hrsPerPage = 0.1; double totalEffort = 0;
    for(Section s:sections){
      System.out.println(s.getTitle() + "\t" + s.getNumPages());
    }
    totalEffort = computeStats(hrsPerPage);
    System.out.println("num. pages: " + totalNumPages);
    System.out.println("estimated effort to read (hrs): " + totalEffort);
  }
}

public final class Topic {
  private String title ;
  public Topic(){title = "NoTitle";}
  public Topic (String aTitle){title = aTitle;}
  public String getTitle(){return title;}
}

public final class Section {
  private String title;
  private int numPages;
  public Section(){title = "noName"; numPages = 0;}
  public Section(String aTitle, int nPages){title = aTitle; numPages = nPages;}
  public String getTitle(){return title;}
  public int getNumPages(){return numPages;}
}

public final class DocManagerDemoApp {
  public static void main(String args[]){
    Topic botany = new Topic("Botany");
    Document doc1 = new Document("Peri fyton by Aristotle", botany);
    System.out.println("\n---------------------------- SECTIONS OF A DOCUMENT ----------------------------");
    Section sec1 = new Section("Intro", 6);  Section sec2 = new Section ("Background", 10);
    Section sec3 = new Section ("Actual contribution", 30);Section sec4 = new Section ("Experiments", 10);
    Section sec5 = new Section ("Fin", 4);
    doc1.addSection(sec1); doc1.addSection(sec2); doc1.addSection(sec3);doc1.addSection(sec4);doc1.addSection(sec5);
    doc1.showDocumentOverview();
    doc1.showDocumentDetails();
    System.out.println("\nOffered by the " + Document.getOwner() );
  }
}
```

Output:

```
---------------------------- SECTIONS OF A DOCUMENT ----------------------------
New section added, total # sections: 1
New section added, total # sections: 2
New section added, total # sections: 3
New section added, total # sections: 4
New section added, total # sections: 5
Univ. Ioannina has ths nice document titled: Peri fyton by Aristotle with the
topic: Botany

Intro  6
Background 10
Actual contribution   30
Experiments10
Fin  4
num. pages: 60
estimated effort to read (hrs): 6.0

Offered by the Univ. Ioannina
```

ΣΗΜΕΙΩΣΕΙΣ

Ακολουθεί ένα παράδειγμα υλοποιημένο με δύο διαφορετικούς τρόπους.
Έστω ότι θέλετε να δημιουργήσετε ένα ηλεκτρονικό βιβλιοπωλείο το οποίο πουλάει δύο διαφορετικά είδη προϊόντων, βιβλία και CD. Κάθε ένα από αυτά περιέχει έναν (μη κενό) τίτλο και μια τιμή (μεγαλύτερη του μηδενός). Κάθε βιβλίο περιέχει και επιπλέον πληροφορίες, όπως τον συγγραφέα του βιβλίου, αν περιέχει μαλακό ή σκληρό εξώφυλλο και το έτος δημοσίευσής του. Όσον αφορά τα CD, προσφέρεται η δυνατότητα της έκπτωσης σε κάποια επιλεγμένα κομμάτια.

Θέλουμε να δημιουργήσουμε μια εφαρμογή που να εμφανίζει στο χρήστη μια λίστα από τα διαθέσιμα βιβλία και CD του ηλεκτρονικού βιβλιοπωλείου, μαζί με τις λεπτομέρειές τους. Επιπλέον, πρέπει να δίνεται η δυνατότητα προσθήκης και διαγραφής των προϊόντων σε ένα καλάθι αγορών. Στη συνέχεια, ο χρήστης της εφαρμογής πρέπει να μπορεί να προβάλει τα προϊόντα τα οποία έχουν προστεθεί στο καλάθι μαζί με τις τιμές τους.

Η πρώτη υλοποίηση (οργανωμένη στο `package bookstoreCounterExample`), δείχνει μια απλή υλοποίηση με τυπικές, παραδοσιακές κλάσεις. Η δεύτερη υλοποίηση (οργανωμένη στο `package bookstoreacceptable`), δείχνει μια υλοποίηση με τη χρήση μιας αφηρημένης κλάσης.

```
package bookstorecounterexample;

public class Item {
  public Item(){title="";price=-1.0;}
  public Item(String aTitle, double aPrice){title = aTitle; price=aPrice;}
  public void showDetails() { System.out.println(title + "\t\t Price:" + price);}
  public double getOriginalPrice() {return price;}
  public double getFinalPrice() {return price;}

  protected String title;
  protected double price;
}
```

```
package bookstorecounterexample;

public class Book extends Item {
  private String author;
  private int packaging; // 0 for simple, 1 for hard box
  private int yearPublished;

  public Book(String aTitle, String anAuthor, int aDate, double
aPrice,
      int aPackage) {
    super(aTitle, aPrice); //constructor of Item!!!
    author = anAuthor;
    packaging = aPackage;
    yearPublished = aDate;
  }

  public void showDetails() {
    super.showDetails();
    System.out.println("by " + author + " at " + yearPublished);
    String packString;
    if (packaging == 1)
      packString = "hard box";
    else
      packString = "simple";
    System.out.println("with " + packString + " packaging.\n");
  }

}
```

```
package bookstorecounterexample;

public class CD extends Item {
  private String artist;
  private double discount;

  public CD(String aTitle, double aPrice, String anArtist, double
            aDiscount) {
    super(aTitle, aPrice);
    artist = anArtist;
    discount = aDiscount;
  }

  public double getFinalPrice() {
    return (price - discount);
  }

  public void showDetails() {
    super.showDetails();
    System.out.println("by " + artist);
    System.out.println("final price: " + getFinalPrice()+ "\n");
  }
}
```

```java
package bookstorecounterexample;

public class ItemManager {
  private ArrayList<Item> allItems;

  public ItemManager(){
    allItems = new ArrayList<Item>();
  }

  public void addItem(Item anItem){
    allItems.add(anItem);
  }

  public void removeItem(int index){
    this.allItems.remove(index);
  }

  public Item getItem(int index){
    return this.allItems.get(index);
  }

  public void reportAllItems(){
    Iterator <Item> ii = allItems.iterator();
    while (ii.hasNext()){
      Item i = ii.next();
      i.showDetails();
    }
    System.out.println("Total number of items: " +
this.allItems.size());
  }
}
```

```java
package bookstorecounterexample;

public class ShoppingCart {

  private ArrayList<Item> items;
  private float totalCost;

  public ShoppingCart(){
    this.items = new ArrayList<Item>();
    this.totalCost = 0;
  }

  public void addItem(Item item){
    this.items.add(item);
  }

  public void removeItem(int i){
    this.items.remove(i);
  }

  private void computeTotalCost(){
    float cost = 0;
    for(Item item: this.items){
      cost += item.getFinalPrice();
    }
    this.totalCost = cost;
  }

  public float getTotalCost(){
    this.computeTotalCost();
    return this.totalCost;
  }

  public void showDetails(){
    System.out.println("------------------------");
    System.out.println("       CART ITEMS       ");
    System.out.println("------------------------");
    for(Item item: this.items){
      item.showDetails();
    }
    System.out.println("Total cost: " + this.getTotalCost());
  }
}
```

ΤΡΟΦΗ ΓΙΑ ΣΚΕΨΗ
Όλα αυτά τα return types που είναι void, μήπως θα μπορούσαν να ανασκευασθούν ώστε να γίνουν λίγο πιο χρήσιμα?

```
package bookstorecounterexample;

public class SimpleBookstoreApplication {

  public static void main(String args[]){
    ItemManager amazon = new ItemManager();
    Book bookRef;
    bookRef = new Book("Discours de la methode", "Rene Descartes", 1637, 50.00, 0);
    amazon.addItem(bookRef);
    bookRef = new Book("The Meditations", "Marcus Aurelius", 180, 30.00, 1);
    amazon.addItem(bookRef);
    bookRef = new Book("The Bacchae","Euripides", -405, 30.00, 1);
    amazon.addItem(bookRef);
    bookRef =  new Book("The Trojan Women","Euripides",-415, 40.00, 0);
    amazon.addItem(bookRef);

    CD cdRef;
    cdRef = new CD("Piece of Mind", 10.0,"Iron Maiden",4.0);
    amazon.addItem(cdRef);
    cdRef = new CD("Matter of Life and Death", 12.0,"Iron Maiden",2.0);
    amazon.addItem(cdRef);
    cdRef = new CD("Perfect Strangers", 12.00, "Deep Purple",1.0);
    amazon.addItem(cdRef);

    amazon.reportAllItems();


    ShoppingCart cart = new ShoppingCart();

    cart.addItem(amazon.getItem(0));
    cart.addItem(amazon.getItem(2));
    cart.addItem(amazon.getItem(3));
    cart.showDetails();

    cart.removeItem(0);
    cart.showDetails();
  }
}
```

Output

```
Discours de la methode   Price:50.0
by Rene Descartes at 1637
with simple packaging.

The Meditations    Price:30.0
by Marcus Aurelius at 180
with hard box packaging.

The Bacchae   Price:30.0
by Euripides at -405
with hard box packaging.

The Trojan Women     Price:40.0
by Euripides at -415
with simple packaging.

Piece of Mind    Price:10.0
by Iron Maiden
final price: 6.0

Matter of Life and Death    Price:12.0
by Iron Maiden
final price: 10.0

Perfect Strangers    Price:12.0
by Deep Purple
final price: 11.0

Total number of items: 7
------------------------
       CART ITEMS
------------------------
Discours de la methode   Price:50.0
by Rene Descartes at 1637
with simple packaging.

The Bacchae   Price:30.0
by Euripides at -405
with hard box packaging.

The Trojan Women     Price:40.0
by Euripides at -415
with simple packaging.

Total cost: 120.0
------------------------
       CART ITEMS
------------------------
The Bacchae   Price:30.0
by Euripides at -405
with hard box packaging.

The Trojan Women     Price:40.0
by Euripides at -415
with simple packaging.

Total cost: 70.0
```

Now using abstract class Item

```java
package bookstoreAcceptable;

public abstract class Item {

  protected String title;
  protected double price;
  protected int id;

  public Item(){title="";price=-1.0;id=-1;}
  public Item(String aTitle, double aPrice,int id){
    title = aTitle; price=aPrice;this.id =id;
}

  public abstract String showDetails();
  public abstract String getDetails();
  public abstract double getFinalPrice();

  public String getTitle(){return title;}
  public int getId(){return id;}
  public double getOriginalPrice(){return price;}

}
```

```java
package bookstoreAcceptable;

public final class Book extends Item {

  public enum PackageType{
    SIMPLE, HARD;
  }

  private String author;
  private PackageType packaging;
  private int yearPublished;

  public Book(String aTitle, String anAuthor, int aDate, double
      aPrice,PackageType aPackageType, int id) {

    super(aTitle, aPrice, id); //constructor of Item!!!
    author = anAuthor;
    packaging = aPackageType;
    yearPublished = aDate;
  }

  @Override
  public String showDetails() {
    String packString = this.packaging.name().toLowerCase() +
        " packaging";
    String result = "***Item id: " + id + "***\n" +
         title + "\t\t Price:" + price + "\n" +
         " by " + author + " at " + yearPublished +  "\n" +
         "with " + packString + ".\n\n";
    return result;
  }

  @Override
  public String getDetails(){
    String packString = this.packaging.name().toLowerCase() +
        " packaging";
    return (title + "\nby " + author + " at " + yearPublished
       + "\nwith " + packString  + "\n");
  }

  @Override
  public double getFinalPrice() {
    return price;
  }
}//end Book
```

```java
package bookstoreAcceptable;

public final class CD extends Item {
  private String artist;
  private double discount;

  public CD(String aTitle, double aPrice, String anArtist,
            double aDiscount, int id) {
    super(aTitle, aPrice, id);
    artist = anArtist;
    discount = aDiscount;
  }

  @Override
  public double getFinalPrice() {
    return (price - discount);
  }

  @Override
  public String showDetails() {
    String result = "***Item id: " + id + "*** \n" +
        title + "\t\t Price:" + price + "\n" +
        "by " + artist + "\n" +
        "final price: " + getFinalPrice()+ "\n\n";
    return result;
  }

  @Override
  public String getDetails(){
    return (title + "\n by " + artist);
  }
}
```

```java
package bookstoreAcceptable;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public final class ItemManager {
  private List<Item> allItems;

  public ItemManager(){
    allItems = new ArrayList<Item>();
  }

  public int addItem(Item anItem){
    allItems.add(anItem);
    return allItems.size();
  }

  public int removeItem(int index){
    this.allItems.remove(index);
    return allItems.size();
  }

  public Item getItem(int index){
    return this.allItems.get(index);
  }

  public List<Item> getAllItems(){
    return this.allItems;
  }

  public String reportAllItems(){
    String result ="";
    for(Item item: this.allItems) {
      result = result + item.showDetails();
    }
    result = result + "Total number of items: " +
      this.allItems.size()+ "\n";
    System.out.println(result);
    return result;
  }
}
```

```java
package bookstoreAcceptable;
import java.util.ArrayList;
import java.util.List;

public final class ShoppingCart {
  private List<Item> items;
  private float totalCost;

  public ShoppingCart(){
    this.items = new ArrayList<Item>();
    this.totalCost = 0;
  }

  public int addItem(Item item){
    this.items.add(item);
    return this.items.size();
  }

public int removeItem(int anId){
    int pos = -1;
    for(int i = 0; i < this.items.size(); i++){
      if (items.get(i).getId() == anId){
        pos = i;
        break;
      }
    }
    if (pos>=0)
      this.items.remove(pos);
    else{
      System.out.println("The id you have specified is not in the
cart");
    }
    return this.items.size();
  }

  private void computeTotalCost(){
    float cost = 0;
    for(Item item: this.items){
      cost += item.getFinalPrice();
    }
    this.totalCost = cost;
  }
```

```java
  public float getTotalCost(){
    this.computeTotalCost();
    return this.totalCost;
  }


  public List<Item> getItems(){
    return this.items;
  }

    public String showDetails(){
    String intro = "------------------------\n" +
                   "       CART ITEMS       \n" +
                   "------------------------\n";

    String itemsString = "";
    for(int i = 0; i < this.items.size(); i++){
      //System.out.println("***Item id: " + i + "***");
      itemsString = itemsString + items.get(i).showDetails();
    }
    String costString = "Total cost: " + this.getTotalCost() +
"\n";

    String result = intro + itemsString + costString;
    System.out.println(result);

    return result;
  }
}
```

```
package bookstoreacceptable;

import java.util.Scanner;

public class SimpleBookstoreApplication {

private Scanner reader;

    public SimpleBookstoreApplication(){
         reader = new Scanner(System.in);
    }

    public int printMenu(){
       int answerOperation = 0;
       while( answerOperation > 5 || answerOperation <= 0){
        System.out.println("Choose(1-4)\n 1. Show items\n 2. Add item to cart\n
3. Show cart\n 4. Remove item from cart\n 5. Exit");
        answerOperation = reader.nextInt();
        if(answerOperation > 5 || answerOperation <= 0)
             System.out.println("Wrong answer! Try again...");
       }

       return answerOperation;
    }

    public Scanner getReader(){ return reader; }

    public static void main(String args[]){
        SimpleBookstoreApplication app = new SimpleBookstoreApplication();

        ItemManager itemManager = new ItemManager();

        Item item = new Book("Discours de la methode","Rene Descartes",
                                  1637, 50.00, 0, 0);
        itemManager.addItem(item);
        item = new Book("The Meditations", "Marcus Aurelius", 180,30.00,
                         1,1);
        itemManager.addItem(item);
        item = new Book("The Bacchae","Euripides", -405,30.00,  1,2);
        itemManager.addItem(item);
        item = new Book("The Trojan Women","Euripides",-415,40.00, 0,3);
        itemManager.addItem(item);
        item = new CD("Piece of Mind", 10.0,"Iron Maiden",4.0,4);
        itemManager.addItem(item);
        item = new CD("Matter of Life and Death", 12.0,"Iron Maiden",2.0,5);
        itemManager.addItem(item);
        item = new CD("Perfect Strangers", 12.00, "Deep Purple",1.0,6);
        itemManager.addItem(item);

        ShoppingCart cart = new ShoppingCart();

        while(true){
      int operation = app.printMenu();
      if(operation == 1){
        itemManager.reportAllItems();
      }
      else if(operation == 2){
        System.out.println("Choose the id of the item that you"
                                                 + " want to add to your cart");
        int id = app.getReader().nextInt();
        System.out.println(id);
        if(id > itemManager.getAllItems().size())
          System.out.println("Error: there is no product"
                                                   + " with the specified id");
        else
          cart.addItem(itemManager.getItem(id));
           }
      else if(operation == 3){
        cart.showDetails();
      }
      else if(operation == 4){
        System.out.println("Choose the id of the item that you"
                                           + " want to remove from your cart");
        int id = app.getReader().nextInt();
        System.out.println(id);
        if(id > itemManager.getAllItems().size())
          System.out.println("Error: there is no product"
                                                   + " with the specified id");
        else
          cart.removeItem(id);
      }
      else{
        break;
      }
        }

        } //endofmain
}
```

Output

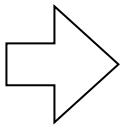
```
Choose(1-4)
 1. Show items
 2. Add item to cart
 3. Show cart
 4. Remove item from cart
 5. Exit
1

***Item id: 0***
Discours de la methode   Price:50.0
 by Rene Descartes at 1637
with simple packaging.

***Item id: 1***
The Meditations   Price:30.0
 by Marcus Aurelius at 180
with hard box packaging.

***Item id: 2***
The Bacchae    Price:30.0
 by Euripides at -405
with hard box packaging.

***Item id: 3***
The Trojan Women    Price:40.0
 by Euripides at -415
with simple packaging.

***Item id: 4***
Piece of Mind    Price:10.0
by Iron Maiden
final price: 6.0

***Item id: 5***
Matter of Life and Death   Price:12.0
by Iron Maiden
final price: 10.0

***Item id: 6***
Perfect Strangers   Price:12.0
by Deep Purple
final price: 11.0
```

```
Choose(1-4)
 1. Show items
 2. Add item to cart
 3. Show cart
 4. Remove item from cart
 5. Exit
2
Choose the id of the item that you want to add to your cart
2
Choose(1-4)
 1. Show items
 2. Add item to cart
 3. Show cart
 4. Remove item from cart
 5. Exit
3
------------------------
       CART ITEMS
------------------------
***Item id: 2***
The Bacchae    Price:30.0
 by Euripides at -405
with hard box packaging.

Total cost: 30.0
```

ΣΗΜΕΙΩΣΕΙΣ

Ακολουθούν παραδείγματα χρήσεως αρχείων και I/O εντολών.

```
 /*
 * author: P. Tsaparas
 * edited: PV
 */
import java.io.FileNotFoundException;
import java.util.Scanner;
import java.io.FileInputStream;
import java.io.PrintWriter;
import java.io.FileOutputStream;
import java.util.StringTokenizer;

public class ReadWriteTest {
  public static void main(String[] args) {
    Scanner inputStream = null;
    PrintWriter outputStream = null;

    Scanner keyboard = new Scanner(System.in);
    System.out.print("Give the input file name: ");
    String inputFileName = keyboard.next();  //also: nextInt, nextDouble, nextLine
    inputFileName = inputFileName.trim();
    System.out.print("Give the output file name: ");
    String outputFileName = keyboard.next();
    outputFileName = outputFileName.trim();

    //Opening files for read and write, checking exception
    try {
      inputStream = new Scanner(new FileInputStream(inputFileName));
      outputStream = new PrintWriter(new FileOutputStream(outputFileName));
      //APPEND would be .... (new FileOutputStream(outputFileName, true));
    } catch (FileNotFoundException e) {
      System.out.println("Problem opening files.");
      System.exit(0);
    }
    //converts tokens to uppercase, adds tabs in between and writes with a line number
    int count = 0;
    while (inputStream.hasNextLine()) {
      String line = inputStream.nextLine();
      String newLine = new String();
      count++;
      StringTokenizer tokenizer = new StringTokenizer(line);
      while (tokenizer.hasMoreTokens()){
        String s = tokenizer.nextToken();
        s =s.toUpperCase();    //ALWAYS: s = s.doSth, else, the immutable s misses the new value
        s= s.replace("|", "\t");
        newLine = newLine + "\t" + s;
      }
      outputStream.println(count + " " + newLine);
    }

    inputStream.close();
    outputStream.close();
    keyboard.close();//because eclipse has a warning :)
    System.out.println("Done, writing " + count + " lines.");
  }
}
```

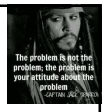


```java
/**
 * A simple example where we open a file, read its contents, make them tab-separated and
 * turn them to upper case, before writing them to an output file.
 * @author pvassil
 */
package examples;

import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;

public final class LineIOExample {
  public static void main(String[] args) {
    String inputFileName = "Books.txt";
    String outputFileName = "BooksProcessed.txt";
    BufferedReader inputStream = null;
    PrintWriter outputStream = null;

    try {
      inputStream = new BufferedReader(new FileReader(inputFileName));
      try {
        outputStream = new PrintWriter(new FileWriter(outputFileName));
      } catch (IOException e) {
        System.err.println("Could not open output file stream");
      }

      String nextInputLine = "";
      try {
        while ((nextInputLine = inputStream.readLine()) != null) {
          String[] stringParts = nextInputLine.split("\\|");
                        // just "|" is a wild-character in reg. ex. => would not do it
          String outputLine = "";
          if (stringParts.length > 0) {
            for(String s: stringParts) {
              outputLine += s.toUpperCase() + "\t";
            }
          } //can you spot the small glitch and fix it before the println?
          outputStream.println(outputLine);
        }
      } catch (IOException e) {
        System.err.println("Could not read line from input file stream");
      }
    } //of the outermost try, to open the inputStream
    catch (FileNotFoundException e) {
      System.err.println("Could not open input file stream");
    }
    finally {
      if (inputStream != null) {
        try {
          inputStream.close();
        } catch (IOException e) {
          System.err.println("Could not close input file stream");
        }
      }
      if (outputStream != null) {
        outputStream.close();  //nothing to throw, if sth is closed already
      }
  }//end main()
}//end class
```

Use Cases: μια σύντομη εισαγωγή

Heavily based on “UML & the UP” by Arlow and Neustadt, Addison Wesley, 2002

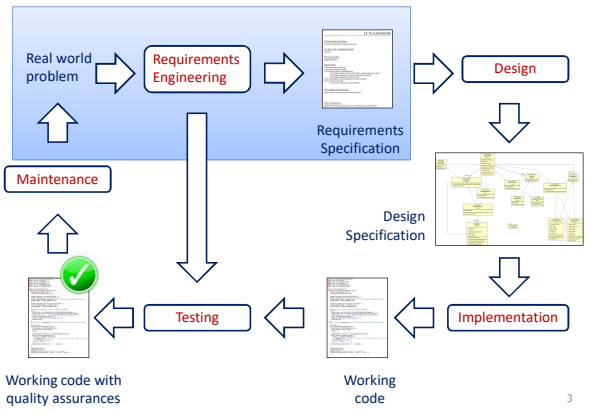
Requirements Engineering



- Η διαδικασία συλλογής και ανάλυσης απαιτήσεων, μεταφράζει το πώς οι χρήστες εκλαμβάνουν ένα πρόβλημα του πραγματικού κόσμου, σε μια **δομημένη** περιγραφή τους

Use case name	**Use case: PayVAT**
Unique identifier	**ID:** UC1
The actors involved in the use case	**Actors:** Time, Government
The system state before the use case can begin	**Preconditions:** 1. It is the end of a business quarter.
The actual steps of the use case	**Flow of events:** 1. The use case starts when it is the end of the business quarter. 2. The system determines the amount of Value Added Tax (VAT) owed to the Government. 3. The system sends an electronic payment to the Government.
The system state when the use case is over	**Postconditions:** 1. The Government receives the correct amount of VAT.

Figure 4.8



Τι / Γιατί / Πώς

- **Τι**: Πρέπει να μεταφράσουμε το πρόβλημα του πραγματικού κόσμου που έχουμε να επιλύσουμε σε μια **δομημένη περιγραφή**
- Γιατί: Όπως θα δούμε σε επόμενες ενότητες, η εν λόγω περιγραφή θα αξιοποιηθεί στην κατασκευή κλάσεων και μεθόδων

  Έχει τεράστια σημασία, η περιγραφή να ΜΗΝ είναι ελεύθερο κείμενο, αλλά να ακολουθεί πολύ αυστηρούς κανόνες δομής και έκφρασης
- Πώς: θα χρησιμοποιήσουμε την τεχνική των use cases που μας δίνουν νοητικά εργαλεία για να μεταφράσουμε κάθε δουλειά / αρμοδιότητα του προγράμματος που θα κατασκευαστεί σε μια σειρά βημάτων και αλληλεπιδράσεων μεταξύ συστήματος και χρήστη

Roadmap

- Αρχικά θα πούμε μερικά πράγματα για τη συλλογή απαιτήσεων
- Στη συνέχεια θα δούμε τα βασικά νοητικά εργαλεία που μας δίνουν οι use cases για να δομήσουμε τις απαιτήσεις
  - Structured description
  - Actors and system boundary
  - Sequencing, branching, looping
  - Alternatives
  - Reusability and extensibility

(γενικές εισαγωγικές ιδέες)

ΣΥΛΛΟΓΗ ΑΠΑΙΤΗΣΕΩΝ

Ανάλυση απαιτήσεων

- Λειτουργικές απαιτήσεις: τι πρέπει να κάνει το σύστημα
- Μη Λειτουργικές Απαιτήσεις: τι εγγυήσεις / περιορισμούς παρέχει το σύστημα (από πλευράς επίδοσης, αξιοπιστίας, κλπ)

Use cases

- Οι use cases είναι το βασικό εργαλείο με το οποίο καταγράφουμε με δομημένο τρόπο τις λειτουργικές απαιτήσεις του συστήματος
- Σε ένα έργο, παράγουμε ένα σύνολο από use case diagrams & use case documents που μας επιτρέπουν:
  - Να καθορίσουμε ποιοι είναι οι εξωτερικοί χρήστες / συστήματα που αλληλεπιδρούν με το σύστημά μας
  - Ποια τα όρια του συστήματος που πάμε να φτιάξουμε
  - Με ποια «σενάρια» γίνεται αυτή η αλληλεπίδραση
  - Ποιες οι λεπτομέρειες για κάθε use case

Συλλογή και Ανάλυση Απαιτήσεων

- Η διαδικασία της συλλογής και ανάλυσης απαιτήσεων, μετά από συνεντεύξεις με τους εμπλεκόμενους στο σύστημα (χρήστες, διαχειριστές, κλπ) πρέπει να καταλήξει στην οριστικοποίηση και καταγραφή των παρακάτω αποτελεσμάτων
  - Λειτουργικές απαιτήσεις
  - Μη Λειτουργικές απαιτήσεις
  - Ανάθεση προτεραιοτήτων και σημαντικότητας στις απαιτήσεις
  - Συσχέτιση των απαιτήσεων με τις παραχθείσες use-cases

Κοινά λάθη

- Η συλλογή απαιτήσεων είναι μια εργασία που απαιτεί πολλές δεξιότητες (σε τεχνικό, διοικητικό και κοινωνικό επίπεδο), μία όμως από αυτές είναι αξιομνημόνευτη εδώ: η αυστηρότητα
- Κοινά λάθη λόγω έλλειψης αυστηρότητας στην καταγραφή των απαιτήσεων
  - Διαγραφή: λείπει κάτι από το ποιος κάνει τι και πότε (π.χ., ο κωδικός εισάγεται στο σύστημα – από ΠΟΙΟΝ?)
  - Παραμόρφωση: κάποιος περιορισμός έχει εναλλακτικές εκτελέσεις (π.χ., αν ο κωδικός εισαχθεί λάθος, ξαναζητείται από το σύστημα – αντί για «…λάθος μέχρι και 3 φορές …»)
  - Γενίκευση: κάποιος κανόνας γενικεύεται σε ένα σύνολο χρηστών, ως μη όφειλε (π.χ., ΟΛΟΙ οι χρήστες έχουν ένα κωδικό – ξεχνώντας ότι οι διαχειριστές μπορεί να μοιράζονται ένα κοινό κωδικό

10

(ορισμός, στοιχεία, δομή και παραδείγματα)

USE CASES

11

Use Case Model

- Το Use Case Model περιλαμβάνει:
  - Actors: ρόλοι που εκτελούνται είτε από τους χρήστες του συστήματος ή από άλλα συστήματα εξωτερικά στο υπό μελέτη σύστημα
  - Use cases: σενάρια χρήσης του συστήματος από τους actors
  - Relationships: με ποιους τρόπους οι use cases σχετίζονται με τους actors
  - System boundary: η απόφαση για το τι είναι εντός τους συστήματος και τι εκτός (το οποίο δεν είναι τόσο προφανές όσο ακούγεται

12

Διαδικασία εκπόνησης

- Βρες τα όρια του συστήματος
- Βρες τους actors
- Βρες τα use cases
  - Ορισμός του use case
  - Κατασκευή σεναρίων μέσα σε ένα use case

13

System boundary

- System boundary defines what is part of the system and what is external to the system
- External to the system: actors (persons or systems
- Internal to the system: use cases = what the system offers to the actors
- Notation in diagrams: box with the name of the system, with actors outside the box and use cases inside

14

Actors

- Actor: ρόλος που υιοθετείται από μια εξωτερική του συστήματος οντότητα για να αλληλεπιδράσει με το σύστημα

- Ένας actor
  - είναι ΠΑΝΤΑ εξωτερικός του συστήματος
  - Αλληλεπιδρά άμεσα με το σύστημα
  - Αναπαριστά ρόλους και όχι συγκεκριμένους ανθρώπους ή συστήματα
  - Χρειάζεται ένα σύντομο όνομα
  - Ζωγραφίζεται με τα γνωστά μπαρμπαδάκια

15

Και πού να τους βρεις?

- Ποιος χρησιμοποιεί το σύστημα?
- Ποιος διαχειρίζεται το σύστημα?
- Ποια άλλα συστήματα αλληλεπιδρούν με το σύστημα?
- Συμβαίνει κάποιο γεγονός σε τακτά χρονικά διαστήματα? (τότε ο χρόνος είναι ένας actor)

16

Use Cases

- A use case is “a specification of sequences of actions, including variant sequences and error sequences, that a system, subsystem, or class can perform by interacting with outside actors”
- Use cases:
  - Come with a short name (starting with a verb) and an ID
  - Are **always started by an actor**
  - Are **always written from an actor’s point of view**

17

Και πού τα βρίσκεις? Ρωτώντας:

- Τι λειτουργικότητα θέλει ένας χρήστης από το σύστημα?
- Ειδοποιείται ένας χρήστης αν αλλάξει η κατάσταση του συστήματος?
- Επηρεάζεται το σύστημα από εξωτερικά γεγονότα?
- Αποθηκεύει / ανακτά πληροφορία το σύστημα, και αν ναι, με ποιο ερέθισμα?

18

Ζωγραφική για μια use case

- Φούσκα με το όνομά της
- Γραμμή που τη συσχετίζει με κάθε actor που σχετίζεται μαζί της

- Το διάγραμμα των use cases του συστήματος βάζει μαζί system boundary, actors and use cases, all in one

Use case diagram

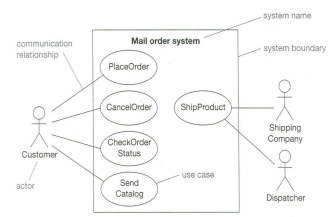


Figure 4.6

Καθορισμός μιας use case

- Το διάγραμμα με τις φούσκες δίνει μόνο μια γενική εποπτεία
- **Ο καθορισμός των use cases γίνεται με κείμενα σαν το διπλανό!!!**

Use case name	**Use case: PayVAT**
Unique identifier	**ID: UC1**
The actors involved in the use case	**Actors:** Time Government
The system state before the use case can begin	**Preconditions:** 1. It is the end of a business quarter.
The actual steps of the use case	**Flow of events:** 1. The use case starts when it is the end of the business quarter. 2. The system determines the amount of Value Added Tax (VAT) owed to the Government. 3. The system sends an electronic payment to the Government.
The system state when the use case is over	**Postconditions:** 1. The Government receives the correct amount of VAT.

e 4.8

Καθορισμός μιας use case

- **Όνομα και ID**
- **Actors** (ordered by frequency of usage + highlight the primary actor)
- **Preconditions** = σύνολο συνθηκών που πρέπει να αποτιμάται σε true πριν να ξεκινήσει η εκτέλεση της use case – αλλιώς δεν ξεκινά (προσοχή: όχι υποχρεωτικά – μόνο όπου χρειάζεται)

Use case name	**Use case: PayVAT**
Unique identifier	**ID: UC1**
The actors involved in the use case	**Actors:** Time Government
The system state before the use case can begin	**Preconditions:** 1. It is the end of a business quarter.
The actual steps of the use case	**Flow of events:** 1. The use case starts when it is the end of the business quarter. 2. The system determines the amount of Value Added Tax (VAT) owed to the Government. 3. The system sends an electronic payment to the Government.
The system state when the use case is over	**Postconditions:** 1. The Government receives the correct amount of VAT.

e 4.8

Καθορισμός μιας use case

- **Primary flow of events** = sequence of steps to implement the use case
- **Alternative Flows**: για τις περιπτώσεις που υπάρχουν exceptions ή πολλά λογικά μονοπάτια εκτέλεσης
- **Postconditions**: τι εγγυήσεις δίνει η εκτέλεση κάθε flow μετά την ολοκλήρωσή της (όπως τα preconditions, πάλι σαν ένα σύνολο από κατηγορήματα που αποτιμώνται σε true και πάλι προαιρετικά, μόνο όπου χρειάζεται)

Use case name	**Use case: PayVAT**
Unique identifier	**ID: UC1**
The actors involved in the use case	**Actors:** Time Government
The system state before the use case can begin	**Preconditions:** 1. It is the end of a business quarter.
The actual steps of the use case	**Flow of events:** 1. The use case starts when it is the end of the business quarter. 2. The system determines the amount of Value Added Tax (VAT) owed to the Government. 3. The system sends an electronic payment to the Government.
The system state when the use case is over	**Postconditions:** 1. The Government receives the correct amount of VAT.

- **Comments**: σχόλια, ρίσκα, μη λειτουργικές απαιτήσεις, …

Δομημένη έκφραση για ροές και use cases

- Χαρακτηρισμός του ποιος ξεκινά την use case

```
1. The use case starts when an <actor> <function>
```

- Αριθμημένες προτάσεις, με αυστηρή σύνταξη (υποκείμενο = actor / (sub)system, ενεργητικό ρήμα (όχι παθητική φωνή)…

```
<id> The <actor / (sub)system> <performs function>
```

- … και απαντήσεις στα ερωτήματα: Ποιος? Τι? Πότε? Πού? σε σχέση με το αντικείμενο της πρότασης

```
4. The main engine shall load the records from the log
file to the engine's record placeholder
```

Δομημένη έκφραση

- Reserve **if, else, for, while** as keywords για να μπορείτε να προδιαγράψετε branching & (rarely) loops
- Δύο τρόποι (συμπληρωματικοί) για το branching:
  - Αν είναι μικρό το branching, μέσα στο primary flow (κάποιοι modelers διαφωνούν – θέλουν καθαρό το primary flow)
  - Αλλιώς, χωριστό **alternative flow**

25

Branching in two ways

Use case: ManageBasket
ID: UC10
Actors: Customer
Preconditions: 1. The shopping basket contents are visible.
Flow of events: 1. The use case starts when the Customer selects an item in the basket. 2. If the Customer selects "delete item" 2.1 The system removes the item from the basket. 3. If the Customer types in a new quantity 3.1 The system updates the quantity of the item in the basket.
Postconditions: 1. The basket contents have been updated.

Use case: DisplayBasket
ID: UC11
Actors: Customer
Preconditions: 1. The Customer is logged on the system.
Flow of events: 1. The use case starts when the Customer selects "display basket". 2. If there are no items in the basket 2.1 The system informs the Customer that there are no items in the basket yet. 2.2 The use case terminates. 3. The system displays a list of all items in the Customer's shopping basket including product ID, name, quantity and item price.
Postconditions:
Alternative flow 1: 1. At any time the Customer may leave the shopping basket screen.
Postconditions:
Alternative flow 2: 1. At any time the Customer may leave the system.
Postconditions:

26

For and While within use cases (rarely, however)

Use case: FindProduct
ID: UC12
Actors: Customer
Preconditions:
Flow of events: 1. The Customer selects "find product". 2. The system asks the Customer for search criteria. 3. The Customer enters the requested criteria. 4. The system searches for products that match the Customer's criteria. 5. If the system finds some matching products then 5.1. For each product found 5.1.1. The system displays a thumbnail sketch of the product. 5.1.2. The system displays a summary of the product details. 5.1.3. The system displays the product price. 6. Else 6.1. The system tells the Customer that no matching products could be found.
Postconditions:
Alternative flow: 1. At any point the Customer may move to different page.
Postconditions:

Use case: ShowCompanyDetails
ID: UC13
Actors: Customer
Preconditions:
Flow of events: 1. The use case starts when the Customer selects "show company details". 2. The system displays a web page showing the company details. 3. While the Customer is browsing the company details 3.1. The system plays some background music. 3.2. The system displays special offers in a banner ad.
Postconditions:

Missing: the use case begins when ....

27

Σενάρια

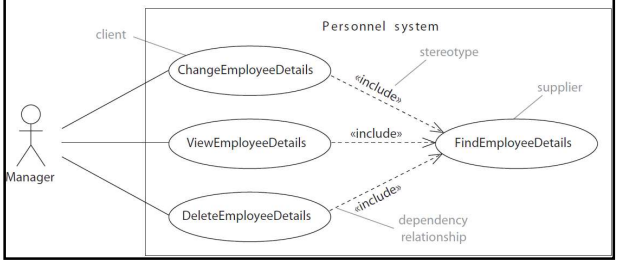
Use case: Checkout
ID: UC14
Actors: Customer
Preconditions:
Primary scenario: 1. The use case begins when the Customer selects "go to checkout". 2. The system displays the customer order. 3. The system asks for the customer identifier. 4. The Customer enters a valid customer identifier. 5. The system retrieves and displays the Customer's details. 6. The system asks for credit card information – name on card, card number and expiry date. 7. The Customer enters the credit card information. 8. The system asks for confirmation of the order. 9. The Customer confirms the order. 10. The system debits the credit card. 11. The system displays an invoice.
Secondary scenarios: InvalidCustomerIdentifier InvalidCreditCardDetails CreditCardLimitExceeded CreditCardExpired
Postconditions:

Use case: Checkout Secondary scenario: InvalidCustomerIdentifier
ID: UC15
Actors: Customer
Preconditions:
Secondary scenario: 1. The use case begins in step 3 of the use case Checkout when the Customer enters an invalid customer identifier. 2. For three invalid entries 2.1. The system asks the Customer to enter the customer identifier again. 3. The system informs the Customer that their customer identifier was not recognized.
Postconditions:

Όταν η δομή γίνεται πολύπλοκη και το branching έχει παραγίνει, βάζουμε **σενάρια**, αναθέτοντας μια ροή σε κάθε ένα εκ των οποίων και χωρίζοντας τη use case σε primary / secondary scenarios

Κλήση use case μέσα από άλλο use case

- Πολύ κοινό όταν έχουμε λειτουργικότητα που επαναλαμβάνεται σε περισσότερα του ενός use cases
- Απλοποιητικό για την σύνταξη των use cases



Use case: ChangeEmployeeDetails
ID: 1
Brief description: The Manager changes the employee details.
Primary actors: Manager
Secondary actors: None.
Preconditions: 1. The Manager is logged on to the system.
Main flow: 1. include(FindEmployeeDetails) 2. The system displays the employee details. 3. The Manager changes the employee details. ...
Postconditions: 1. The employee details have been changed.
Alternative flows: None.

Use case: ViewEmployeeDetails
ID: 2
Brief description: The Manager views the employee details.
Primary actors: Manager
Secondary actors: None.
Preconditions: 1. The Manager is logged on to the system.
Main flow: 1. include(FindEmployeeDetails) 2. The system displays the employee details. ...
Postconditions: 1. The system has displayed the employee details.
Alternative flows: None.

Use case: DeleteEmployeeDetails
ID: 3
Brief description: The Manager deletes the employee details.
Primary actors: Manager
Secondary actors: None.
Preconditions: 1. The Manager is logged on to the system.
Main flow: 1. include(FindEmployeeDetails) 2. The system displays the employee details. 3. The Manager deletes the employee details. ...
Postconditions: 1. The employee details have been deleted.
Alternative flows: None.

Use case: FindEmployeeDetails
ID: 4
Brief description: The Manager finds the employee details.
Primary actors: Manager
Secondary actors: None.
Preconditions: 1. The Manager is logged on to the system.
Main flow: 1. The Manager enters the employee's ID. 2. The system finds the employee details.
Postconditions: 1. The system has found the employee details.
Alternative flows: None.

Η «καλούμενη» use case, από τις προηγούμενες use cases.
Ορολογία: η παρούσα «καλούμενη» use case ονομάζεται "supplier" και οι καλούσες use cases ονομάζονται "client" use cases.

Όπως βλέπετε, ΔΕΝ είναι πλήρης ("the use case begins when …")

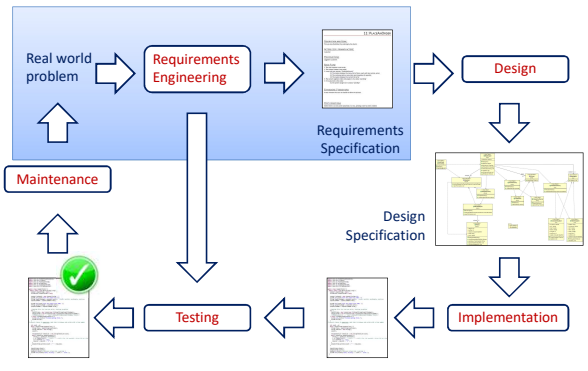
Η supplier use case ΔΕΝ είναι υποχρεωτικό να είναι πλήρης, αλλά μπορεί να αποτελεί αυτό που ονομάζουμε behavior fragment. Φυσικά, πάντα μπορεί μια supplier use case να είναι πλήρης και να ενεργοποιείται από την κατάλληλη ενέργεια του primary actor της.

ΠΡΟΣΟΧΗ: η μόνη περίπτωση που επιτρέπουμε behavior fragments as use cases είναι να χρησιμοποιηθούν μέσω include «κλήσεων»

31

ΣΥΝΟΨΗ

32



33

Σύνοψη (1/3)

- **Τι**: Πρέπει να μεταφράσουμε το πρόβλημα του πραγματικού κόσμου που έχουμε να επιλύσουμε σε μια **δομημένη περιγραφή**
- Γιατί: Όπως θα δούμε σε επόμενες ενότητες, η εν λόγω περιγραφή θα αξιοποιηθεί στην κατασκευή κλάσεων και μεθόδων

Έχει τεράστια σημασία, η περιγραφή να ΜΗΝ είναι ελεύθερο κείμενο, αλλά να ακολουθεί πολύ αυστηρούς κανόνες δομής και έκφρασης

- Πώς: είδαμε την τεχνική των use cases που μας δίνουν νοητικά εργαλεία για να μεταφράσουμε κάθε δουλειά / αρμοδιότητα του προγράμματος που θα κατασκευαστεί σε μια σειρά βημάτων και αλληλεπιδράσεων μεταξύ συστήματος και χρήστη

Σύνοψη (2/3): βασικά χαρ/κα

- Χαρακτηρισμός του ποιος ξεκινά την use case

  `1. The use case starts when an <actor> <function>`

- Αριθμημένες προτάσεις, με αυστηρή σύνταξη (υποκείμενο = actor / (sub)system, ενεργητικό ρήμα (όχι παθητική φωνή)…

  `<id> The <actor / (sub)system> <performs function>`

- … και απαντήσεις στα ερωτήματα: Ποιος? Τι? Πότε? Πού? σε σχέση με το αντικείμενο της πρότασης

  `4. The main engine shall load the records from the log file to the engine's record placeholder`

Σύνοψη (3/3): κοινά λάθη

- Όνομα: κάτι που δεν είναι ρήμα

  ~~`διάθεσηΠροϊόντος`~~ αντί για `πρόσφερεΠροϊόν (ρήμα)`

- Actor: δηλώνετε το σύστημα για actor

  ~~`Το σύστημα`~~ Ποτέ!! Εξ' ορισμού ο actor είναι ΕΞΩ από το σύστημα

- Ξεχνάτε το ποιος ξεκινά την use case (αν δεν υπάρχει αυτό, ΔΕΝ ΣΤΟΙΧΕΙΩΘΕΤΕΙΤΑΙ USE CASE)

  `1. The use case starts when an <actor> <function>`

- Ξεχνάτε ότι οι προτάσεις πρέπει να είναι αριθμημένες !!
- Παραβιάζετε την αυστηρή σύνταξη (υποκείμενο = actor / (sub)system, ενεργητικό ρήμα (όχι παθητική φωνή)…

  `<id> The <actor / (sub)system> <performs function>`

~~`4. Γίνεται εκτύπωση των στοιχείων (παθητική φωνή, χωρίς υποκείμενο)`~~

ΣΗΜΕΙΩΣΕΙΣ

USE CASE NAME

DESCRIPTION AND GOAL

ACTORS (ESP. PRIMARY ACTOR)

PRECONDITIONS

BASIC FLOW

EXTENSIONS / VARIATIONS

POST CONDITIONS

SPECIAL REQUIREMENTS, ISSUES, RISKS AND OTHER COMMENTS

USE CASE NAME

DESCRIPTION AND GOAL

Describes the context of the goal.

ACTORS (ESP. PRIMARY ACTOR)

List of actors, listed according to the frequency they are expected to initiate the use case.

PRECONDITIONS

List of items that must be fulfilled or true before the use case is initiated.

BASIC FLOW

1. Step 1.
2. Step 2.
3. Step 3.

EXTENSIONS / VARIATIONS

1a. Variations or extensions to the steps listed above. “1a” here means that there is an extension or variation to step 1, above.

1b. The second variation of step 1. Avoid more than one level of variation.

POST CONDITIONS

The state of the system after performing the use case.

SPECIAL REQUIREMENTS, ISSUES, RISKS AND OTHER COMMENTS

Enumerations of any special requirements, especially items that developers or technical writers are unlikely to be aware of. List of open issues, questions or risks. These do not have to be resolved before work on the use case begins, but must be resolved before the release containing the use is considered complete.

ΣΗΜΕΙΩΣΕΙΣ

Ακολουθεί ένα παράδειγμα για το πώς

- δοθείσης μιας περιγραφής ενός προβλήματος που έχουμε καταγράψει σε φυσική γλώσσα,
- εξάγουμε κάποια use cases

ΗΛΕΚΤΡΟΝΙΚΟ ΒΙΒΛΙΟΠΩΛΕΙΟ

Έστω ότι θέλετε να δημιουργήσετε ένα ηλεκτρονικό βιβλιοπωλείο το οποίο πουλάει δύο διαφορετικά είδη προϊόντων, βιβλία και CD. Κάθε ένα από αυτά περιέχει έναν (μη κενό) τίτλο και μια τιμή (μεγαλύτερη του μηδενός). Κάθε βιβλίο περιέχει και επιπλέον πληροφορίες, όπως τον συγγραφέα του βιβλίου, αν περιέχει μαλακό ή σκληρό εξώφυλλο και το έτος δημοσίευσής του. Όσον αφορά τα CD, προσφέρεται η δυνατότητα της έκπτωσης σε κάποια επιλεγμένα κομμάτια.

Θέλουμε να δημιουργήσουμε μια εφαρμογή που να εμφανίζει στο χρήστη μια λίστα από τα διαθέσιμα βιβλία και CD του ηλεκτρονικού βιβλιοπωλείου, μαζί με τις λεπτομέρειές τους. Επιπλέον, πρέπει να δίνεται η δυνατότητα προσθήκης και διαγραφής των προϊόντων σε ένα καλάθι αγορών. Στη συνέχεια, ο χρήστης της εφαρμογής πρέπει να μπορεί να προβάλει τα προϊόντα τα οποία έχουν προστεθεί στο καλάθι μαζί με τις τιμές τους.

ΕΜΦΑΝΙΣΕΠΡΟΪΟΝΤΑΚΑΤΑΣΤΗΜΑΤΟΣ

ID: UC 1

DESCRIPTION AND GOAL

Η use case «ΕμφάνισεΠροϊόνταΚαταστήματος» εμφανίζει τα διαθέσιμα προϊόντα του καταστήματος στην οθόνη.

ACTORS (ESP. PRIMARY ACTOR)

Ο χρήστης.

PRECONDITIONS

Πρέπει να έχουν φορτωθεί και να υπάρχουν διαθέσιμα προϊόντα στο κατάστημα.

BASIC FLOW

1. Το use case ξεκινάει όταν ο χρήστης επιλέξει από μενού την επιλογή «Εμφάνιση προϊόντων».
2. Το σύστημα εμφανίζει τα προϊόντα στην οθόνη.

EXTENSIONS / VARIATIONS

1. Στην περίπτωση κατά την οποία δεν υπάρχουν προϊόντα στο κατάστημα εμφανίζεται ένα μήνυμα που ενημερώνει ότι δεν υπάρχουν διαθέσιμα προϊόντα.

POST CONDITIONS

-

ΠΡΟΣΘΕΣΕΠΡΟΪΟΝΣΤΟΚΑΛΑΘΙ

ID: UC 2

DESCRIPTION AND GOAL

Ο χρήστης επιλέγει το προϊόν που επιθυμεί και το προσθέτει στο καλάθι.

ACTORS (ESP. PRIMARY ACTOR)

Ο χρήστης.

PRECONDITIONS

Πρέπει να έχουν φορτωθεί και να υπάρχουν διαθέσιμα προϊόντα στο κατάστημα.

BASIC FLOW

1. Το use case ξεκινάει όταν ο χρήστης επιλέξει την προσθήκη προϊόντος στο καλάθι από το μενού επιλογών.
2. Ο χρήστης επιλέγει το προϊόν που επιθυμεί να αγοράσει από την διαθέσιμη λίστα.
3. Το σύστημα προσθέτει το προϊόν στο καλάθι.

POST CONDITIONS

Τα προϊόντα που βρίσκονται στο καλάθι έχουν ενημερωθεί.

ΔΙΑΓΡΑΨΕΠΡΟΙΟΝΑΠΟΚΑΛΑΘΙ

ID: UC 3

DESCRIPTION AND GOAL

Ο χρήστης επιλέγει το προϊόν που επιθυμεί και το διαγράφει από το καλάθι.

ACTORS (ESP. PRIMARY ACTOR)

Ο χρήστης.

PRECONDITIONS

Πρέπει να υπάρχουν προϊόντα στο καλάθι.

BASIC FLOW

1. Το use case ξεκινάει όταν ο χρήστης επιλέγει να διαγράψει κάποιο προϊόν από το καλάθι αγορών από το μενού επιλογών.
2. Ο χρήστης επιλέγει το προϊόν που επιθυμεί να διαγραφεί από το καλάθι.
3. Το σύστημα διαγράφει το προϊόν από το καλάθι.

POST CONDITIONS

Τα προϊόντα που βρίσκονται στο καλάθι έχουν ενημερωθεί.

ΕΜΦΑΝΙΣΕΠΡΟΪΟΝΤΑΤΟΥΚΑΛΑΘΙΟΥ

ID: UC 4

DESCRIPTION AND GOAL

Οι αναλυτικές πληροφορίες για κάθε προϊόν που βρίσκεται στο καλάθι αγορών εμφανίζονται στην οθόνη.

ACTORS (ESP. PRIMARY ACTOR)

Ο χρήστης.

PRECONDITIONS

Πρέπει να έχουν προστεθεί προϊόντα στο καλάθι αγορών.

BASIC FLOW

1. Το use case ξεκινάει όταν ο χρήστης επιλέξει να εμφανίσει τα προϊόντα που βρίσκονται στο καλάθι αγορών χρησιμοποιώντας την επιλογή από το μενού επιλογών.
2. Το σύστημα εμφανίζει σε μια λίστα όλα τα προϊόντα τα οποία έχουν τοποθετηθεί στο καλάθι αγορών μαζί με τις αναλυτικές τους πληροφορίες.

EXTENSIONS / VARIATIONS

1. Στην περίπτωση κατά την οποία δεν υπάρχουν προϊόντα στο καλάθι αγορών, το σύστημα εμφανίζει το ανάλογο μήνυμα.

POST CONDITIONS

-

Βασικά διαγράμματα σχεδίασης με UML

Ανάπτυξη Λογισμικού (Software Development)

www.cs.uoi.gr/~pvassil/courses/sw_dev/

ΜΥΥ301/ ΠΛΥ 308

Σχεδίαση

- Η σχεδίαση λαμβάνει ως είσοδο τις συγκροτημένες απαιτήσεις των χρηστών ενός (υπο)συστήματος και **παράγει μια αφαιρετική αναπαράσταση του πώς δομείται το λογισμικό εσωτερικά**, με στόχο να ανταποκριθεί στις απαιτήσεις αυτές

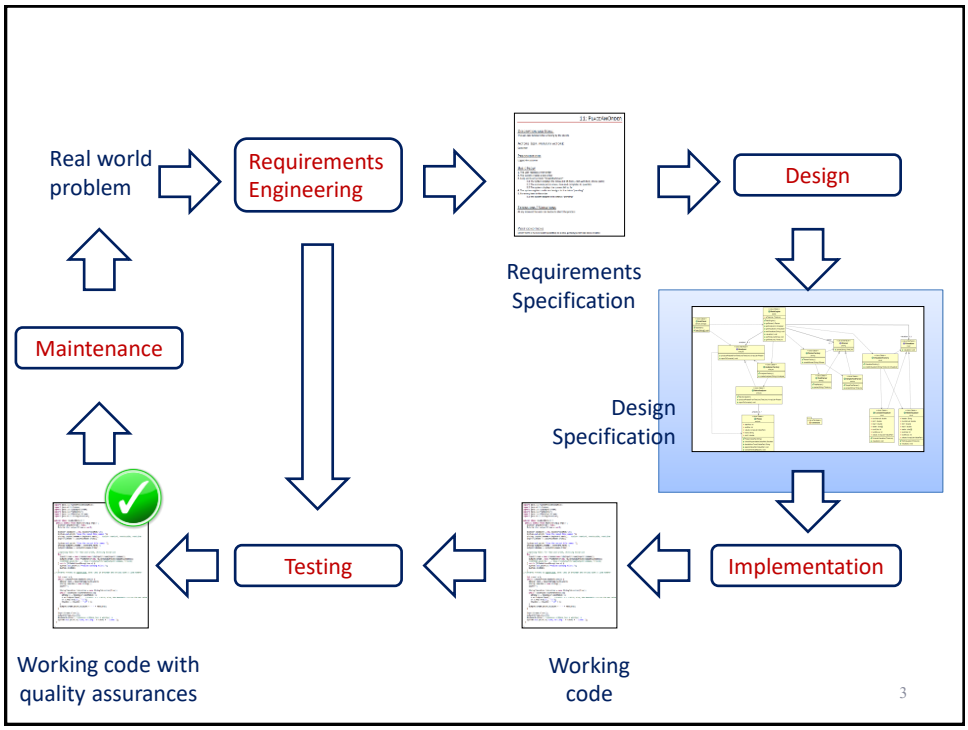
Leslie Lamport,

Turing Award Recipient, 2013

Leslie Lamport. Who Builds a House without Drawing Blueprints? Comm. ACM, 58(4), pp. 38-41, Apr. 2015

http://cacm.acm.org/magazines/2015/4/184705-who-builds-a-house-without-drawing-blueprints/

2



Τι / Γιατί / Πώς

- Στην παρούσα ενότητα θα υποστηρίξουμε τη διαδικασία σχεδίασης με τα βασικά (όχι όλα) **σχεδιαστικά εργαλεία** που έχουμε για το αντικειμενοστρεφές λογισμικό
  / μαθαίνουμε τι είναι το πριόνι, πριν μάθουμε να πριονίζουμε */*

Αυτά είναι τα **σχεδιαστικά διαγράμματα UML**, από τα οποία θα καλύψουμε τα πιο βασικά

- Τελειώνοντας αυτή την ενότητα, θα πρέπει να είστε εις θέση αφενός να κατανοείτε και αφετέρου να μπορείτε να κατασκευάσετε ένα διάγραμμα UML που αναπαριστά αφαιρετικά ένα (υπο)σύστημα αντικειμενοστρεφούς λογισμικού

4

Δομή

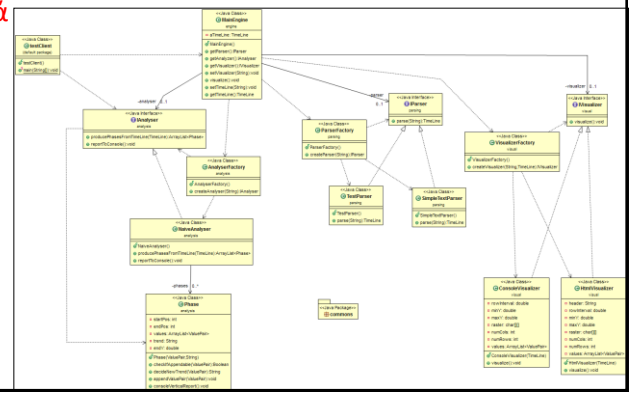
- Περί διαγραμμάτων και σχεδίασης λογισμικού
- Στατικά Διαγράμματα Κλάσεων
- Δυναμικά Διαγράμματα Ακολουθιών

- Στα πλαίσια της σχεδίασης διαγραμμάτων κλάσεων θα εντρυφήσουμε και στα interfaces

Πολλές ευχαριστίες στον Α. Ζάρρα για μια πρώτη μορφή των διαφανειών αυτών

Διαγράμματα

- Όπως σε όλες τις επιστήμες των μηχανικών, έτσι και στη μηχανική της σχεδίασης του λογισμικού, κατασκευάζουμε ένα σχέδιο του προς εκτέλεση συστήματος
- Όπως σε όλες τις επιστήμες των μηχανικών, έτσι και στη μηχανική της σχεδίασης του λογισμικού, το εν λόγω σχέδιο γίνεται σε μεγάλο βαθμό διαγραμματικά



Διαγράμματα

- Η Unified Modeling Language (UML) είναι η γλώσσα που μας προσφέρει τα building blocks με τα οποία κατασκευάζουμε τα διαγράμματα αυτά
- Η UML είναι ευρύτατα διαδεδομένη και είναι η γλώσσα προτυποποίησης της σχεδίασης που χρησιμοποιείται παντού
  - Όταν δεν ακολουθούνται εναλλακτικές μορφές διεκπεραίωσης της ανάπτυξης (τις οποίες θα συναντήσετε στο μάθημα της Τεχν. Λογισμικού)

Γιατί διαγράμματα?

- **Για μας!**
  - Για να αποτυπώσουμε γρήγορα και κατανοητά το σχέδιο του τι θα υλοποιήσουμε
  - Για να μπορούμε να κατανοήσουμε τη δομή ενός λογισμικού που πρέπει να συντηρήσουμε
  - Για να κρύψουμε γενικά την πολυπλοκότητα του κώδικα σε μια φιλική, διαγραμματική και γρήγορα αντιληπτή αποτύπωση που λειτουργεί ως blueprint / σκελετός / σχέδιο της υλοποίησης!

Γιατί διαγράμματα?

- **Για τους άλλους!**
  - Για να μπορούμε, σε ένα συνεργατικό περιβάλλον, να συνεργαστούμε με άλλους, να κατανοούμε όλοι γρήγορα, εύκολα και με ακρίβεια τι έχει / πρόκειτα να υλοποιηθεί από τους άλλους

Μην ξεχνάτε

- Τα διαγράμματα τα ζωγραφίζουμε με άπλα στο χώρο, χωρίς οπτικό θόρυβο, και με ακρίβεια (που συνεπάγεται ενημέρωση όταν γίνονται αλλαγές)

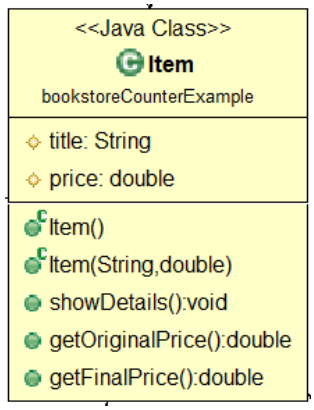
- **ΜΗΝ ΞΕΧΝΑΤΕ: ο σκοπός των διαγραμμάτων είναι να υποβοηθήσουν την κατανόηση του κώδικα και τη συνεργασία των developers (αλλιώς είναι απλώς επιπλέον φόρτος)**

(Class diagrams)

ΔΙΑΓΡΑΜΜΑΤΑ ΚΛΑΣΕΩΝ

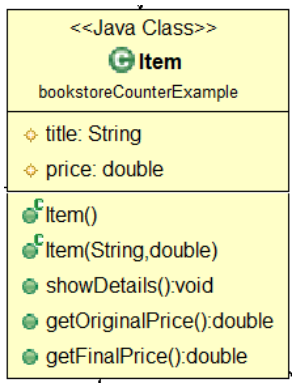
Στατική Άποψη - Κλάσεις

Classes



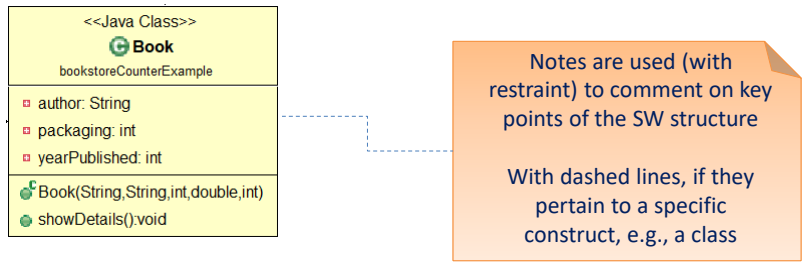
- Class (package, if abstract, …)
- State properties (attributes)
- Dynamic behavior (methods)
- Annotate:
  - Class if abstract
  - For methods && attributes: + for public, - for private, # for protected
  - Method if constructor, abstract, …
  - Method signature
  - Attribute type

Classes




```
package bookstorecounterexample;

public class Item {
    public Item(){title="";price=-1.0;}

    public Item(String aTitle, double aPrice){
          title = aTitle; price=aPrice;
    }

    public void showDetails() {
          System.out.println(title + "\t\t
                    Price:" + price);}
    public double getOriginalPrice() {return price;}

    public double getFinalPrice() {
          return price;
    }

    protected String title;
    protected double price;
}
```

Comment



Στατική Άποψη - Εξάρτηση

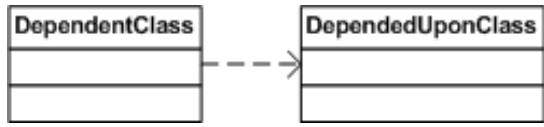
Σχέση εξάρτησης (dependency)

Η σχέση εξάρτησης καταγράφει ότι ο εξαρτημένος χρησιμοποιεί κώδικα του εξαρτώμενου

Dependent typically depends upon DependedUpon for class def., method invocation and stability

Αν αλλάξει ο εξαρτώμενος, ίσως πρέπει να αλλάξει και ο εξαρτημένος

17



dependency)

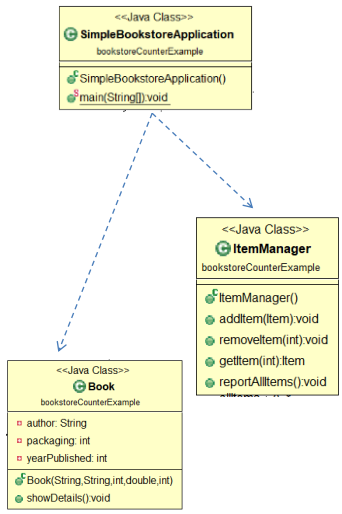
- Class DC (DependentClass) depends upon class DUC (DependedUponClass) if even one of the following holds:
  - There exists a method `DC.f` that takes as parameter an object of DCU

    `DC.f(DUC x): …`
  - There exists a method `DC.f` that returns an object of DUC

    `DC.f(…): DUC`
  - There exists a method `DC.f` that uses an object of DUC as an internally declared variable

18

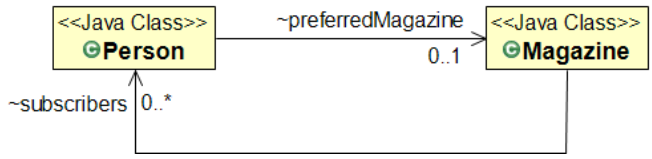
Παράδειγμα




```
public class SimpleBookstoreApplication {

   public static void main(String args[]){
          ItemManager amazon = new ItemManager();
          Book bookRef;
          bookRef = new Book("Discours de la methode",
                       "Rene Descartes", 1637, 50.00, 0);
          amazon.addItem(bookRef);
          bookRef = new Book("The Meditations", "Marcus
                            Aurelius", 180, 30.00, 1);
          amazon.addItem(bookRef);
          bookRef = new Book("The Bacchae","Euripides",
                          -405, 30.00, 1);
          amazon.addItem(bookRef);

          CD cdRef;
          cdRef = new CD("Piece of Mind", 10.0,"Iron
                            Maiden",4.0);
          amazon.addItem(cdRef);
          cdRef = new CD("Matter of Life and Death",
                    12.0,"Iron Maiden",2.0);
          amazon.addItem(cdRef);

          amazon.reportAllItems();
   }
}
```

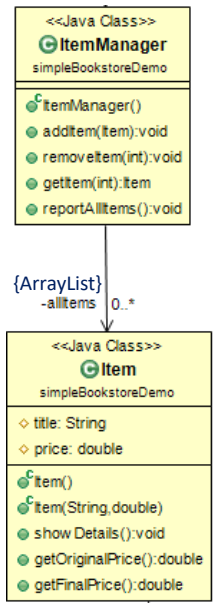

Στατική Άποψη – Συσχέτιση

UML Class Diagrams – Συσχέτιση



- Μια απλή σχέση συσχέτισης (Association) σημαίνει ότι κατά τη διάρκεια εκτέλεσης κάποια αντικείμενα των δύο κλάσεων συνεργάζονται
- Το multiplicity (αν υπάρχει) δηλώνει ένα εύρος επιτρεπόμενων τιμών σχετικών με το πόσα αντικείμενα συνεργάζονται
  - 5, 10, 0..1, 1..10, 1..*, 0..*, κλπ.
- Υπάρχει επίσης δυνατότητα να ονοματίσουμε το ρόλο που παίζουν τα αντικείμενα κάθε κλάσης στη συνεργασία
  - Εδώ: a Person subscribes to 0..1 Magazine to which he refers as "preferred Magazine"; at the same time, a Magazine can have 0 to many Persons to which it refers to as "subscribers".
- Η κατεύθυνση (αν υπάρχει, φαίνεται με βέλος) υποδηλώνει ότι αντικείμενα της κλάσης απ' όπου ξεκινά το βέλος γνωρίζουν την ύπαρξη και έχουν τη δυνατότητα πρόσβασης στα αντικείμενα της κλάσης όπου καταλήγει το βέλος

Παράδειγμα

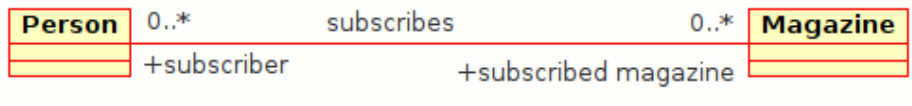



```
public class ItemManager {
   private ArrayList<Item> allItems;
   …
}
```

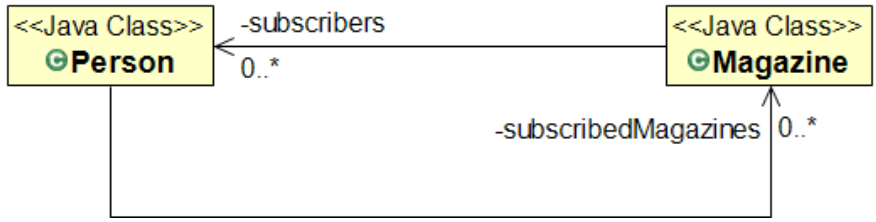
- Το βέλος υποδηλώνει ότι αντικείμενα της κλάσης Item Manager
- Η επισημείωση 0..* δηλώνει ότι ένα αντικείμενο της κλάσης Item Manager γνωρίζει 0 ... πολλά αντικείμενα της κλάσης Item, στα οποία αναφέρονται ως allItems. Πρακτικά αυτό μας λέει ότι κάθε αντικείμενο της κλάσης ItemManager συσχετίζεται με μια συλλογή από αντικείμενα της Item
- Τα αντικείμενα Item δεν ξέρουν τίποτα για αντικείμενα τύπου ItemManager

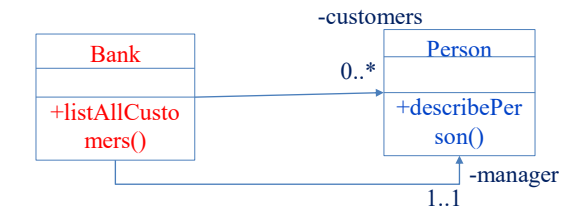
22

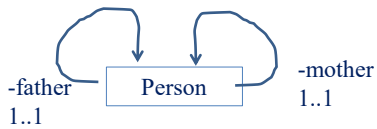
Εναλλακτικοί συμβολισμοί



https://en.wikipedia.org/wiki/Class_diagram





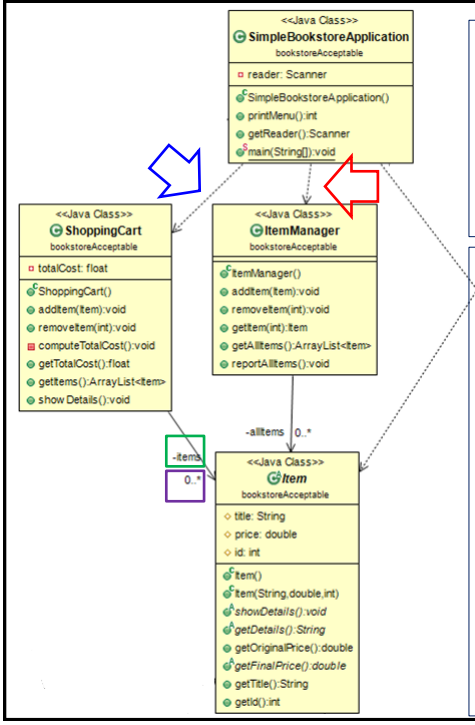



```
Public class Bank{
  private List<Person> allCustomers;

  public void listAllCustomers(){
     for (Person p: this.allCustomers){
          …. Do sth with each customer…
          p.describePerson();
     }
  }
```

```
public void listAllCustomers(List<Person>
customers){
     for (Person p: this.allCustomers){
          …. Do sth with each customer…
          p.describePerson();
     }
}
```

Συσχέτιση vs Εξάρτηση

- Η συσχέτιση είναι κατά βάση δομική σχέση: μία κλάση Α έχει ένα πεδίο που είναι τύπου μιας κλάσης Β
- Η εξάρτηση υποδηλώνει ότι μια κλάση Α εξαρτάται από μια άλλη κλάση Β, και αυτό κυρίως λέει ότι αν χαθεί / αλλάξει η Β, πρέπει να συντηρήσουμε την Α.
- Η εξάρτηση είναι υποχρεωτικά κατευθυνόμενη
  - Στη συσχέτιση μπορεί να μην υπάρχει κατεύθυνση, στην περίπτωση αυτή θεωρείται ακαθόριστη




```
public class ShoppingCart {
   private ArrayList<Item> items;
}
/* Attribute => association
 * Observe the concrete association
 * (not dependency), the role name
 * and the multiplicity => ArrayList
 */
```

```
public class SimpleBookstoreApplication {
   …
public static void main(String args[]){
  ItemManager itemManager = new ItemManager();
  …
  ShoppingCart cart = new ShoppingCart();
  …
  cart.showDetails();
}
```

Παρατηρείστε το **διακεκομμένο** βελάκι από τον SimpleBookstoreApplication προς τον ItemManager, που υποδηλώνει την εξάρτηση, η οποία προκύπτει από αυτό το new.
Οποιαδήποτε από τις μπλε εντολές αρκεί για να δημιουργήσει την εξάρτηση από τον SimpleBookstoreApplication προς τον Shopping Cart

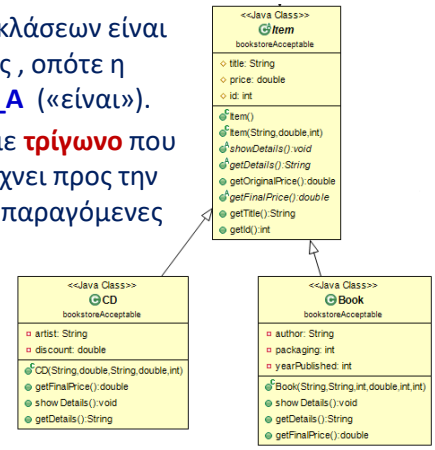
Στατική Άποψη - Κληρονομικότητα

Κληρονομικότητα

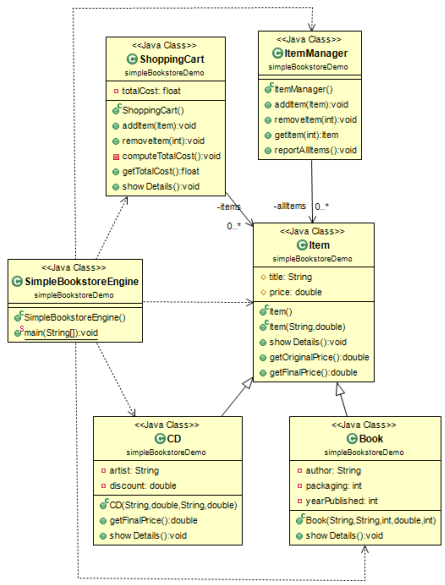
- Για καλύτερη οργάνωση και συντήρηση του κώδικα (για να μην επαναλαμβάνεται ο ίδιος κώδικας πολλές φορές)
  - φτιάχνουμε μια **γενική/βασική κλάση** που να περιλαμβάνει τα κοινά χαρακτηριστικά/πεδία και μεθόδους δύο ή περισσοτέρων κλάσεων και
  - εν συνεχεία **επεκτείνουμε** τη βασική φτιάχνοντας κλάσεις που **κληρονομούν** από αυτή.
- Στις **παραγόμενες κλάσεις** δηλώνουμε επιπλέον χαρακτηριστικά και λειτουργίες.
  - Τα αντικείμενα έχουν όλα τα χαρακτηριστικά και λειτουργίες που δηλώνονται στην βασική κλάση,
  - καθώς και τα επιπλέον χαρακτηριστικά και λειτουργίες που δηλώνονται στην παραγόμενη κλάση

Κληρονομικότητα & UML

- Τα αντικείμενα των παραγόμενων κλάσεων είναι και αντικείμενα της βασικής κλάσης , οπότε η επέκταση ονομάζεται και σχέση **IS_A** («είναι»).
- Η κληρονομικότητα συμβολίζεται με **τρίγωνο** που τοποθετείται με την κορυφή να δείχνει προς την αρχική κλάση και τη βάση προς τις παραγόμενες



- Εκτός εξαιρετικού απροόπτου (για εσάς: **ΥΠΟΧΡΕΩΤΙΚΑ**), **η μητρική κλάση ζωγραφίζεται ψηλά και οι παραγόμενες από κάτω**




```
package simpleBookstoreDemo;

public class Item {
...
}

public class Book extends Item {
...
}

public class CD extends Item {
...
}

public class ItemManager {
     private ArrayList<Item> allItems;
...
}
```

Προσοχή! Εδώ οι CD & Book κληρονομούν από την κλάση Item.
Θα δούμε αργότερα ότι **η σωστή σχεδίαση επιτάσσει η μαμά-κλάση να είναι αφηρημένη!**

Στατική Άποψη – Αφηρημένες κλάσεις / Πολυμορφισμός

Γιατί πολυμορφισμός?

- Γενικά είναι επιθυμητό είναι να έχουμε τη δυνατότητα να προσθέτουμε νέες δυνατότητες σε ένα πρόγραμμα χωρίς να χρειαστεί να αλλάξουμε δραστικά τον υπάρχοντα κώδικα …
- Πώς γίνεται αυτό ??
  - συνδυάζουμε **κληρονομικότητα** & **πολυμορφισμό (και για την ακρίβεια: επαναορισμό μεθόδων)**
- **method overriding (επαναορισμός μεθόδων)**
  - **η μέθοδος μιας βασική κλάσης ορίζεται ξανά στην παραγόμενη κλάση με νέα υλοποίηση και το ίδιο πρωτότυπο**
  - **με αυτή τη τεχνική μπορούμε να πετύχουμε τον επεκτασιμότητα του κώδικα!!**

Κληρονομικότητα & Πολυμορφισμός

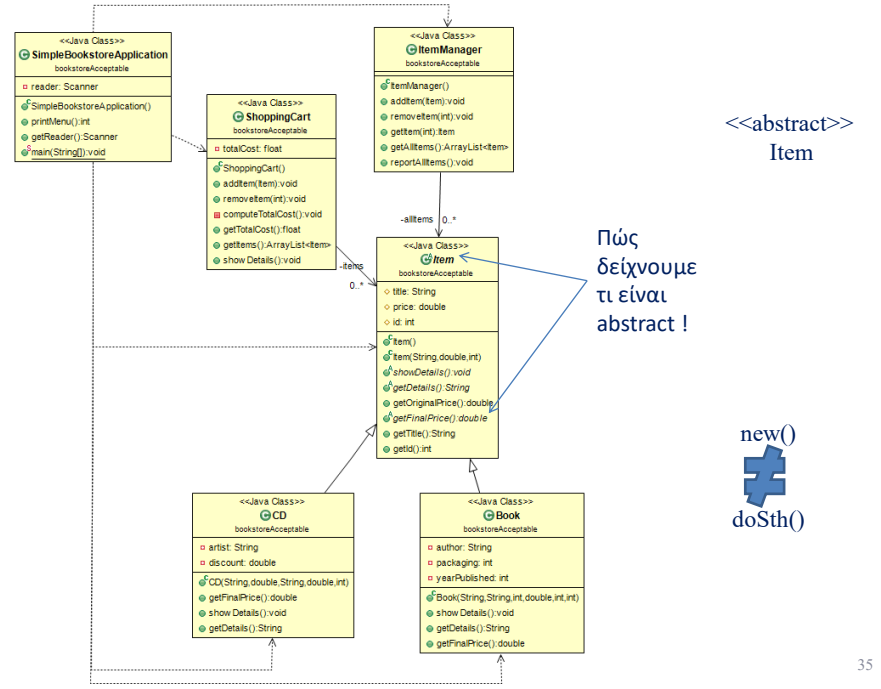
Πώς μπορούμε να ελαχιστοποιήσουμε τη συντήρηση του κώδικα μέσω του πολυμορφισμού?

Σημεία πόνου: κάθε αρμοδιότητα του προγράμματος για την οποία υπάρχει δυνατότητα διαφορετικών εναλλακτικών υλοποιήσεων και κατά συνέπεια η δυνατότητα μελλοντικών επεκτάσεων στο πρόγραμμά μας (π.χ., διαφορετικές μορφές αποθήκευσης των δεδομένων σε ένα αρχείο)

1. για κάθε σημείο πόνου ορίζουμε μια βασική αφηρημένη κλάση
2. Εν συνεχεία για κάθε διαφορετική εναλλακτική υλοποίηση κατασκευάζουμε μια παραγόμενη κλάση η οποία ξανα-ορίζει (overrides) τις public μεθόδους της βασικής αφηρημένης κλάσης
3. υλοποιούμε / παραμετροποιούμε τον υπόλοιπο κώδικα χρησιμοποιώντας αναφορές στη βασική αφηρημένη κλάση
   - οι αναφορές αυτές μπορούν να δείχνουν σε αντικείμενα οποιασδήποτε κλάσης προκύπτει από τη βασική
   - επομένως τα μόνα σημεία τα οποία πρέπει να αλλαχθούν σε μια μελλοντική επέκταση είναι τα σημεία στο οποία αρχικοποιούνται οι αναφορές αυτές
   - ο υπόλοιπος κώδικας στον οποίο καλούνται μέθοδοι σε αντικείμενα στα οποία δείχνουν οι αναφορές αυτές δεν χρειάζεται αλλαγές

Αφηρημένες κλάσεις

- Δηλώνοντας μια κλάση ως αφηρημένη (abstract)
  - μια κλάση ονομάζεται αφηρημένη αν περιέχει τουλάχιστον μια αφηρημένη μέθοδο που δεν περιλαμβάνει υλοποίηση
    - Η κλάση δηλώνεται public abstract class MyAbstractClass
    - με τη μέθοδο public abstract returnType methodName();
  - η αφηρημένη κλάση λειτουργεί σαν καλούπι για την κατασκευή παραγόμενων που προσφέρουν εναλλακτικές υλοποιήσεις στις αφηρημένες μεθόδους….
  - η δημιουργία αντικειμένων αφηρημένης κλάσης δεν επιτρέπεται από τον compiler
  - η μη υλοποίηση αφηρημένων μεθόδων δεν επιτρέπεται από τον compiler



Abstract class

```
package bookstoreAcceptable;

public abstract class Item {
   protected String title;
   protected double price;
   protected int id;

   public Item(){title="";price=-1.0;id=-1;}
   public Item(String aTitle, double aPrice,int id){
        title = aTitle; price=aPrice; this.id =id;
   }

   public abstract String showDetails();
   public abstract String getDetails();
   public abstract double getFinalPrice();

   public double getOriginalPrice(){return price;}
   public String getTitle(){return title;}
   public int getId(){return id;}
}
```

One implementation

```
package bookstoreAcceptable;

public class Book extends Item {
      private String author;

…
      @Override
      public double getFinalPrice() {
            return price;
      }

}
```


Another implementation

```
public class CD extends Item {
        private String artist;
        …
        @Override
        public double getFinalPrice() {
                return (price - discount);
        }

        @Override
        public String showDetails() {
                String result = "***Item id: " + id + "*** \n" +
                        title + "\t\t Price:" + price + "\n" +
                        "by " + artist + "\n" +
                        "final price: " + getFinalPrice()+ "\n\n";
                return result;
        }
}
```

Abstract coupling

```
public class ItemManager {
  private ArrayList<Item> allItems;
…
}
```

The “client” class ItemManager uses ONLY the abstract class and is completely agnostic to any subclasses the abstract class has.

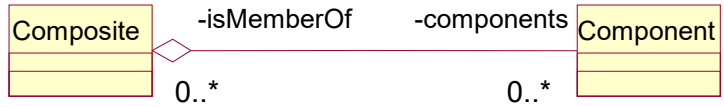
=> Zero maintenance cost when new subclasses appear

Στατική Άποψη - Συνάθροιση

EN: aggregation

Γνωστή και ως «συσσωμάτωση»

UML Class Diagrams - Συνάθροιση

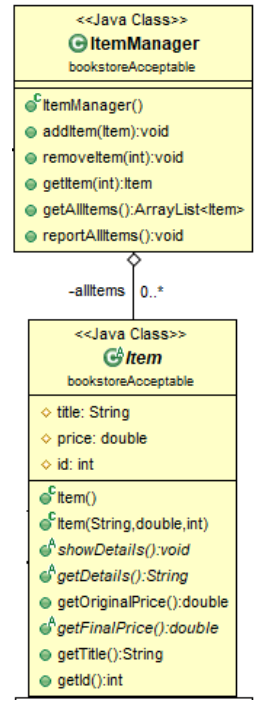


- Ένα αντικείμενο της κλάσης **Composite** (πχ μια ομάδα εργασίας) είναι συνάθροιση αντικειμένων της **Component αν χαρακτηρίζεται/αποτελείται από αντικείμενα της κλάσης Component** (πχ τα μέλη μιας ομάδας)
  - **Σε μια σχέση συνάθροισης** ένα αντικείμενο της κλάσης Component (π.χ., το μέλος της ομάδας) μπορεί να χαρακτηρίζει πολλαπλά αντικείμενα της κλάσης Composite (π.χ., ένας υπάλληλος συμμετέχει σε πολλές ομάδες εργασίας) - ή και αντικείμενα μιας άλλης κλάσης (π.χ., της διαχείρισης μισθοδοσίας) AnotherCompositeClass…
  - Σύμφωνα με τα παραπάνω η ύπαρξη των αντικειμένων της Component δεν εξαρτάται άμεσα από την ύπαρξη των αντικειμένων της Composite
    - (π.χ., το πρόγραμμα μισθοδοσίας διαχειρίζεται μια λίστα από υπαλλήλους ακόμα και αν δεν συμμετέχουν σε ομάδες εργασίας)

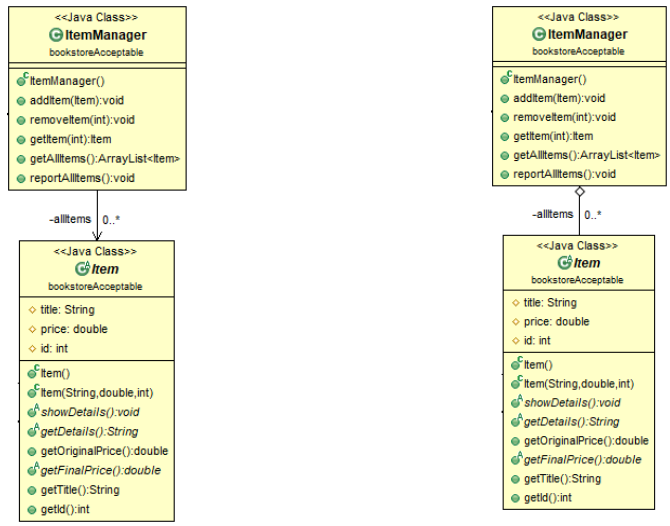
Παράδειγμα

```
public class ItemManager {
  private ArrayList<Item> allItems;
…
}
```

- Προσοχή στον άσπρο ρόμβο:
  - Στη μεριά του σύνθετου αντικειμένου
  - Προσοχή στο χρώμα: άσπρος



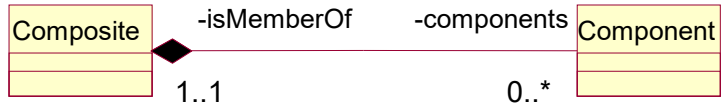
Η συνάθροιση είναι μια ειδική περίπτωση συσχέτισης



Στατική Άποψη - Σύνθεση

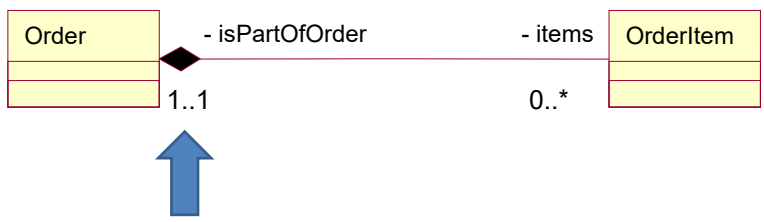
EN: composition

UML Class Diagrams - Σύνθεση



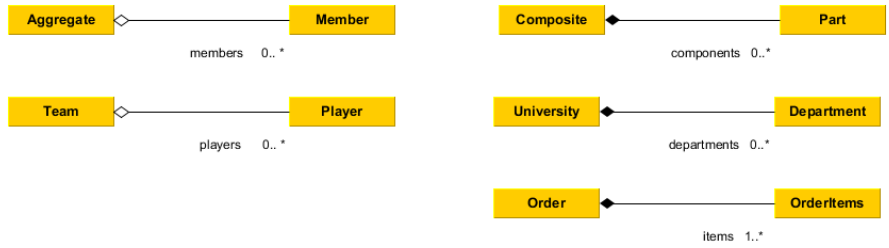
- Ένα αντικείμενο της κλάσης Composite (πχ μια εταιρεία) **αποτελεί σύνθεση αντικειμένων** της Component αν χαρακτηρίζεται/αποτελείται από αντικείμενα της κλάσης Component (πχ τα τμήματα της εταιρείας)
  - **Σε μια σχέση σύνθεσης** ένα αντικείμενο Component ανήκει το **πολύ σε ένα** αντικείμενο της Composite
  - **Σύμφωνα με τα παραπάνω ένα αντικείμενο της κλάσης Component δεν έχει λόγο ύπαρξης αν δεν υπάρχει το αντικείμενο της Composite το οποίο χαρακτηρίζει**

UML Class Diagrams - Σύνθεση



- Σε ότι αφορά τη σημειογραφία προσέξτε ότι εδώ το διαμαντάκι είναι μαυρισμένο (σε αντιδιαστολή με το διαμαντάκι της συνάθροισης που είναι άσπρο)

Aggregation vs Composition



- Σε μια συνάθροιση, τα μέλη παίζουν κάποιο ρόλο.
  - Έτσι μπορεί να συμμετέχουν σε πολλών ειδών συναθροίσεις και η ύπαρξή τους είναι ανεξάρτητη της συνάθροισης
- Τα συστατικά μιας σύνθεσης δεν έχουν νόημα ύπαρξης χωρίς το σύνθετο αντικείμενο που τα «στεγάζει».
  - Η διαγραφή του σύνθετου κόμβου στη σύνθεση, συνεπάγεται την διαγραφή του συστατικού του.

Στατική Άποψη – Interfaces / Πολυμορφισμός

Interfaces

definition by Arlow & Neustadt

- **An interface specifies a named set of public features** (practically speaking: **operations**)
- The key idea behind interfaces is to separate the specification of functionality (the interface) from its implementation by a classifier such as a class or subsystem.
- An interface can't be instantiated - it simply declares a **contract** that may be realized by zero or more classifiers.
- **Anything that realizes an interface accepts and agrees to abide by the contract that the interface defines**.

Interfaces: what-it-is & an example

- Πρακτικά, ένα interface ορίζει ένα σύνολο δημόσιων αφηρημένων μεθόδων τις οποίες, οι κλάσεις που υλοποιούν το interface υποχρεούνται να υλοποιήσουν και να παρέχουν
  - μπορείτε να το σκέφτεστε ως συμβόλαιο, με τον interpreter/compiler στο ρόλο του δικαστή/συμβολαιογράφου που εντοπίζει παραβιάσεις του συμβολαίου
- Όπως θα δούμε στη συνέχεια, η χρήση interfaces στην υλοποίηση ενός συστήματος βοηθά ιδιαίτερα στη συντήρηση του συστήματος μέσω της αρχής του διαχωρισμού των interfaces

Παράδειγμα

An interface is an abstraction of a contract, which specifies guaranteed methods to be offered by materializations of the interface

```
public interface IBreaks {
    public abstract double getSpeedReduction(Double excertedForce);
    public abstract boolean reportIfBroken(double threshold);
}

public class BreakFactory {
    public IBreaks constructBreak(String concreteClassName){
        if (concreteClassName.equals("NiceBreaks"))
                return new NiceBreaks();
        else if (concreteClassName.equals("DuperBreaks"))
                return new DuperBreaks();

        System.out.println("If you got up to here, you passed a wrong argument to BreakFactory");
        return null;
    }
}
```


Παράδειγμα

```
public class NiceBreaks implements IBreaks {
    @Override
    public double getSpeedReduction(Double excertedForce){
        return 45*excertedForce;
    }
    @Override
    public boolean reportIfBroken (double threshold) {
        if ((threshold > 1.0) || (threshold > 1.0)){
                System.out.println("Threshold for breaks' health should be between 0 & 1");
                System.exit(-1);
        }
        Random randomDice = new Random();
        if (randomDice.nextDouble() > threshold)
                return false;
        else
                return true;
    }
}
```

Παράδειγμα

```
public class DuperBreaks implements IBreaks {
    @Override
    public double getSpeedReduction(Double excertedForce){
        return 45*excertedForce;
    }

    @Override
    public boolean reportIfBroken (double threshold) {
        //Intentional issue, food for thought: parameter unused!
        return false;
    }
}
```


Παράδειγμα

```
import bicycleBreaks.IBreaks;
import bicycleBreaks.BreakFactory;
...
public class BicycleManager {
    ...
    public double setBreaking(double force){
        velocity -= breaks.getSpeedReduction(force);
        return velocity;
    }
    public boolean reportIfDamageExists(){
        boolean brokenStatus = false;
        if (breaks.reportIfBroken(0.7) == true){
            brokenStatus = true;
            System.out.println("breaks are broken");
        }
        ...
        return brokenStatus;
    }
    private double velocity;
    private IBreaks breaks;
    private BreakFactory breakFactory;
    ...
}
```

A client class:
- has an attribute typed by the interface (here: IBreaks)
- uses a factory to generate concrete objects of the concrete classes
- uses the abstract methods of the contract to complete its task (here: getSpeedReduction and reportIfBroken)

=>

The client code is completely independent of the concrete classes (just knows exactly two elements, the interface and the factory) …
… thus allowing the addition of new classes or maintaining the existing ones, without any impact to the client!

```
//see how the constructor uses the factory
public BicycleManager( ..., String breaksName){
  breakFactory = new BreakFactory();
  ...
  breaks = breakFactory.constructBreak(breaksName);
  ...
  velocity = 0.0;
}
```

Interfaces: why & how

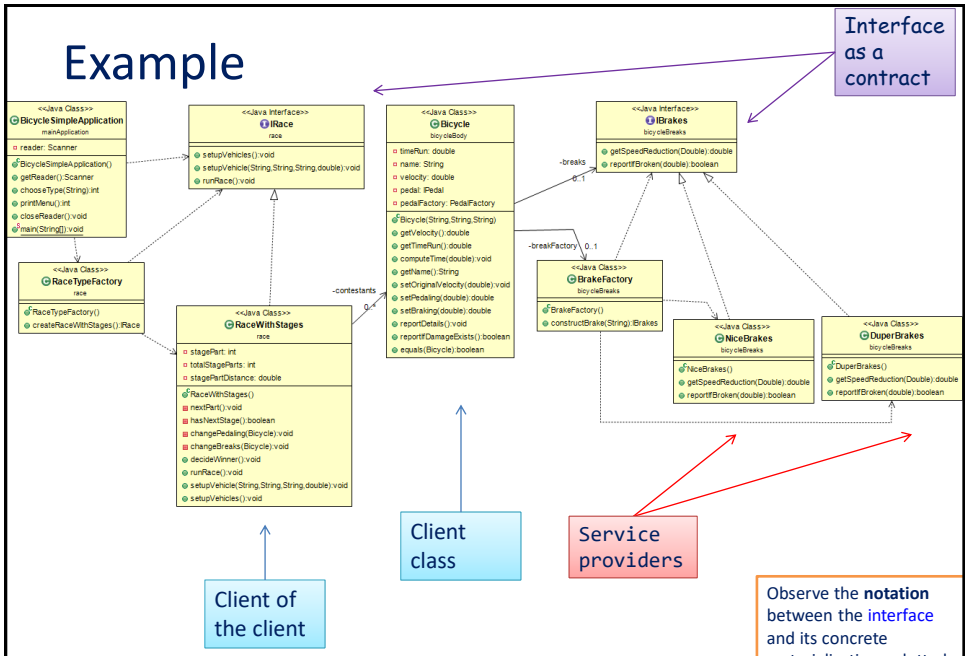
- Client code specifies the service it wants to use
- Service-provision code implements the specified functionality

`Specification ≠ Implementation`

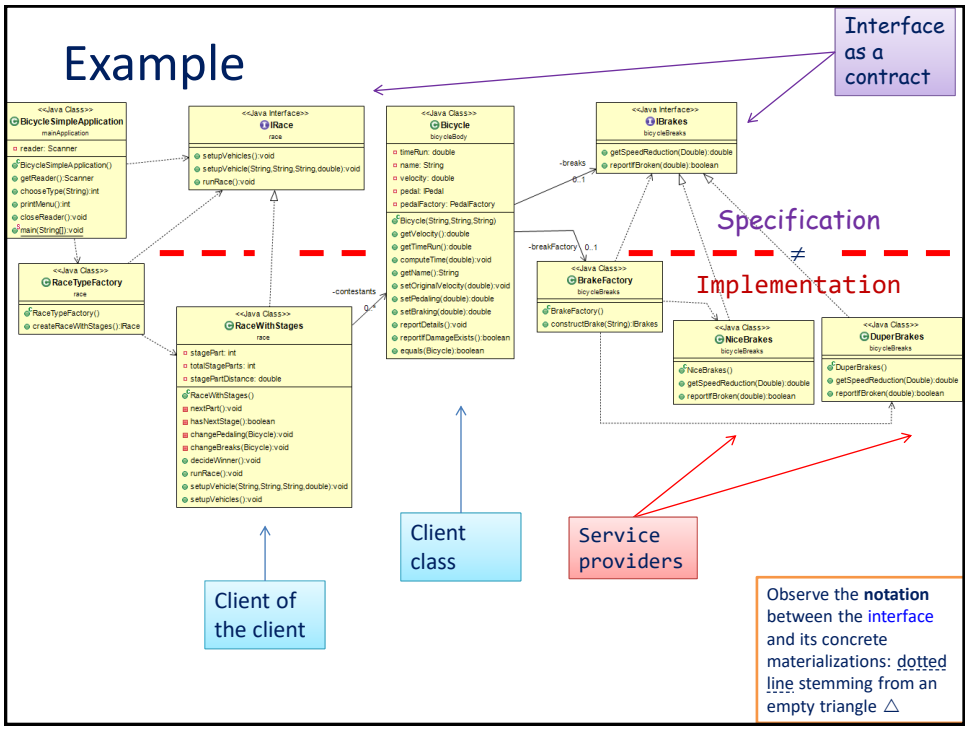
- The interface is a **contract** between the service provider and the service consumer
- As service provision evolves (new versions, new alternatives, bug fixes, …) **the client is immune to change if and only if the contract is respected**

55

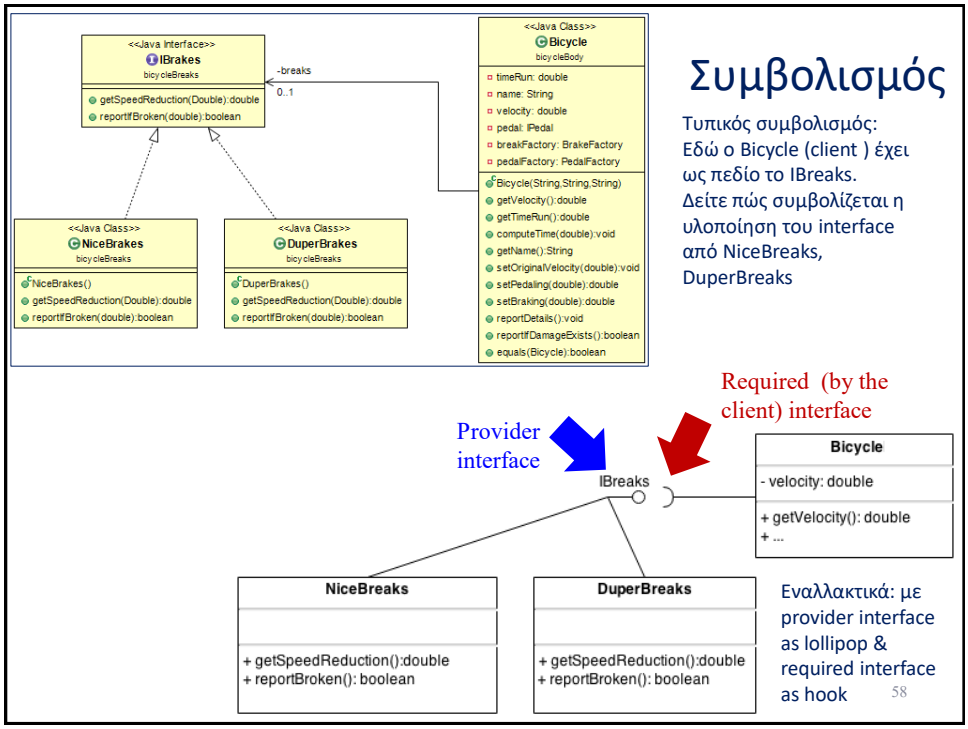
Example



Example



Συμβολισμός



Τυπικός συμβολισμός:
Εδώ ο Bicycle (client) έχει ως πεδίο το IBreaks.
Δείτε πώς συμβολίζεται η υλοποίηση του interface από NiceBreaks, DuperBreaks

Εναλλακτικά: με provider interface as lollipop & required interface as hook

Interfaces: a checklist

- **Avoid:**
  - Attributes and implementation details
    - just stick to the fundamental set of methods needed
  - Any kind of associations to other classes
  - The temptation of reusing code (!!)

Java Interfaces

- Στη Java,
  - εν αντιθέσει με την κληρονομικότητα όπου μια κλάση μπορεί να κληρονομεί από μια μόνο βασική κλάση
    - μια κλάση μπορεί να υλοποιεί περισσότερα από ένα interfaces
  - επίσης ένα interface μπορεί να κληρονομεί από άλλα interfaces.

Abstract classes vs. interfaces

- Abstract classes have a predefined behavior for some of their functionality; interfaces don't
- Interfaces are used for:
  - Stable contract-as-API between subsystems
  - Dissimilar objects that have common functionalities
  - Small bits of functionality

61

Στατική Άποψη – Ιεραρχία ή Συνάθροιση?

(ένα από τα κλασικά προβλήματα σχεδίασης)

Οι ιεραρχίες κληρονομικότητας ΔΕΝ είναι φίλοι μας, ΜΟΝΟ ο πολυμορφισμός είναι!

- Στο μάθημα έχουμε ήδη μιλήσει πολλές φορές για το πόσο ισχυρό εργαλείο είναι το **abstract coupling και η εκμετάλλευση του πολυμορφισμού** για να κατασκευάζουμε client κώδικα που είναι subclass agnostic (και κατά συνέπεια απαιτεί μηδενική συντήρηση όταν αλλάζει η ιεραρχία).
- Έχουμε πει επίσης τα πλεονεκτήματα του write & test once, use by many για την περίπτωση του κώδικα που κληροδοτείται από τις μητρικές κλάσεις στις κλάσεις – παιδιά.
- Όμως …

Οι ιεραρχίες κληρονομικότητας ΔΕΝ είναι φίλοι μας

- Η κληρονομικότητα είναι η ισχυρότερη μορφή εξάρτησης μεταξύ δύο κλάσεων
- Οι αλλαγές στη μητρική κλάση «κατεβαίνουν» προς τις κληρονομούσες κλάσεις => **αν η μητρική κλάση αλλάξει, όλες οι κλάσεις που την κληρονομούν καθώς και οι clients τους θέλουν συντήρηση**

Οι ιεραρχίες κληρονομικότητας ΔΕΝ είναι φίλοι μας,
η συνάθροιση ΕΊΝΑΙ!

Πριν ορίσετε ιεραρχία κληρονομικότητας, σκεφτείτε τη συνάθροιση!

- Η συνάθροιση (συσχέτιση μεταξύ δύο κλάσεων που είναι ανεξάρτητες κατά τα λοιπά) πολύ συχνά μπορεί να κάνει πολύ καλύτερα τη δουλειά από μια ιεραρχία.
- Αν τα αντικείμενα των κλάσεων σχετίζονται με προσωρινή σχέση (π.χ., ένας υπάλληλος τράπεζας εργάζεται ως ταμίας μόνο για ένα διάστημα), τότε η λύση είναι η συσχέτιση των κλάσεων με συνάθροιση και όχι με ιεραρχίες!
- Θυμηθείτε το κριτήριο ISA – εν αντιθέσει, η συσχέτιση μπορεί να ονομαστεί HAS-A (object has a role)

ΆΠΟΨΗ ΑΛΛΗΛΕΠΊΔΡΑΣΗΣ

Άποψη Αλληλεπίδρασης

- Στη στατική άποψη απεικονίζεται η δομή ενός λογισμικού (κλάσεις, σχέσεις).
- Δεν υπάρχει επαρκής πληροφορία για τον τρόπο με τον οποίο υλοποιούνται οι διάφορες λειτουργίες που προσφέρονται στο χρήστη μέσω της αλληλεπίδρασης των αντικειμένων των κλάσεων του λογισμικού
  - Με τον όρο αλληλεπίδραση εννοούμε την κλήση μεθόδων των αντικειμένων / ή με άλλα λόγια την ανταλλαγή μηνυμάτων μεταξύ των αντικειμένων των κλάσεων του λογισμικού.

Άποψη Αλληλεπίδρασης

- Διαγράμματα ακολουθίας (sequence diagrams)
- Διαγράμματα επικοινωνίας (collaboration diagrams (UML 1.x) / communication diagrams (UML 2))

Sequence Diagrams

- Απεικόνιση της αλληλεπίδρασης των αντικειμένων του λογισμικού σε 2 άξονες
  - Ο οριζόντιος άξονας απεικονίζει ένα σύνολο αντικειμένων κατά τη διάρκεια εκτέλεσης του λογισμικού.
  - Ο κατακόρυφος άξονας απεικονίζει τη διάρκεια ζωής των αντικειμένων.
    - Δηλαδή ?

Sequence Diagrams

```
public class AClass {
    public int aa1;
    public float aa2;

    public AClass(int a1, float a2){
        aa1 = a1;
        aa2 = a2;
    }
    public void ma1(BClass bo, int flag){
        bo.mb1(flag);
    }
    public void ma2(){
        System.out.println(aa1+aa2);
    }
}
```

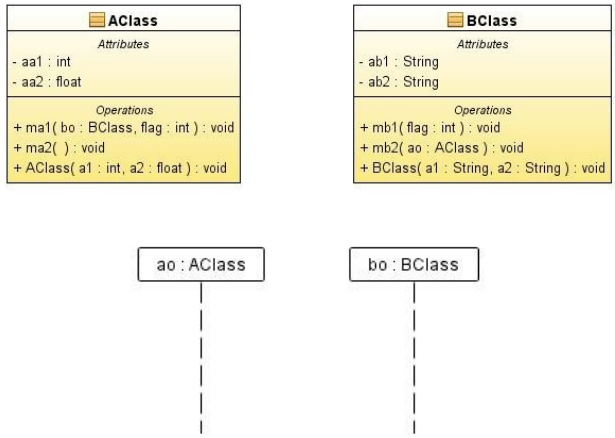
Sequence Diagrams

```
public class BClass {
    public String ab1;
    public String ab2;

    public BClass(String a1, String a2){
        ab1 = a1;
        ab2 = a2;
    }
    public void mb1(int flag){
        System.out.println(ab1 + flag);
    }
    public void mb2(AClass ao){
        System.out.println(ab2);
        ao.ma2();
    }
}
```


Sequence Diagrams

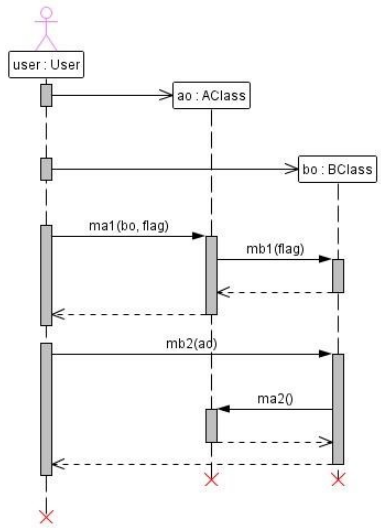
```
public class Main {
   public static void main(String[] args) {
        InputStreamReader in = new InputStreamReader(System.in);
        BufferedReader br = new BufferedReader(in);
        String inData1, inData2;
        try {
            inData1 = br.readLine();
            inData2= br.readLine();
            AClass ao = new AClass(Integer.parseInt(inData1),
                                   Integer.parseInt(inData2));
            inData1 = br.readLine();
            inData2= br.readLine();
            BClass bo = new BClass(inData1, inData2);

            inData1 = br.readLine();
            ao.ma1(bo, Integer.parseInt(inData1));
           bo.mb2(ao);
        } catch (IOException ex) {
           System.out.println(ex.getMessage());
        }
    }
}
```

Sequence Diagrams



- Απεικόνιση αντικειμένων
  - διάρκεια ζωής αντικειμένων
  - … η διάρκεια κατά την οποία είναι δεσμευμένη μνήμη για αυτά…
- Γενικά μπορούμε να απεικονίσουμε και ένα actor
  - αν θέλουμε να δείξουμε ότι λόγω της αλληλεπίδρασης με αυτόν ξεκινά μια διαδικασία

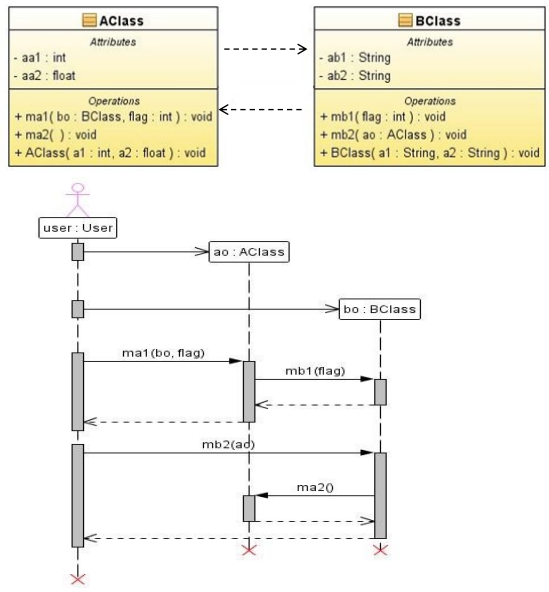
Sequence Diagrams



- Αλληλεπίδραση αντικειμένων
  - αναπαράσταση χρήστη
  - δημιουργία αντικειμένων
  - καταστροφή αντικειμένων
  - κλήση μεθόδου
    - επιστροφή
  - διάρκεια εκτέλεσης μεθόδου (πλαίσιο ενεργοποίησης)
  - κλήση μιας μεθόδου στο πλαίσιο της εκτέλεσης μιας άλλης μεθόδου

→ Είναι σωστό βάσει του κώδικα ??
→ Τι σημαίνει αυτό το διάγραμμα σε σχέση με τη στατική άποψη του συστήματος ???

Sequence Diagrams



- Αλληλεπίδραση αντικειμένων
  - για να είναι σωστό ένα διάγραμμα αλληλεπίδρασης το οποίο απεικονίζει αντικείμενα που καλούν μεθόδους σε άλλα αντικείμενα πρέπει
    - να υπάρχει αντίστοιχο διάγραμμα κλάσεων στο οποίο να υπάρχουν κατάλληλες συσχετίσεις/συναθροίσεις/συνθέσεις μεταξύ των κλάσεων των αντικειμένων

Sequence Diagrams

```
public class AClass {
    public int aa1;
    public float aa2;

    public AClass(int a1, float a2){
        aa1 = a1;
        aa2 = a2;
    }
    public void ma1(BClass bo, int flag){
        if(flag >= 0)
            bo.mb1(flag);
        else
            bo.mb2(this);
    }
    public void ma2(){
        System.out.println(aa1+aa2);
    }
}
```

εναλλακτικές αλληλεπιδράσεις

Sequence Diagrams

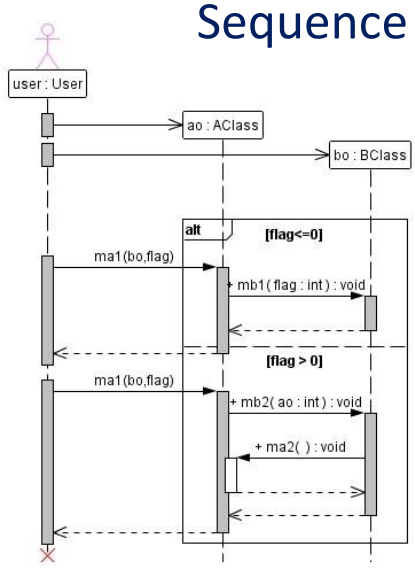
```
public class BClass {
    public String ab1;
    public String ab2;

    public BClass(String a1, String a2){
        ab1 = a1;
        ab2 = a2;
    }
    public void mb1(int flag){
        System.out.println(ab1 + flag);
    }
    public void mb2(AClass ao){
        System.out.println(ab2);
        ao.ma2();
    }
}
```


Sequence Diagrams

```
public class Main {
   public static void main(String[] args) {
        InputStreamReader in = new InputStreamReader(System.in);
        BufferedReader br = new BufferedReader(in);
        String inData1, inData2;
        try {
            inData1 = br.readLine();
            inData2= br.readLine();
            AClass ao = new AClass(Integer.parseInt(inData1),
                                   Integer.parseInt(inData2));
            inData1 = br.readLine();
            inData2= br.readLine();
            BClass bo = new BClass(inData1, inData2);

            inData1 = br.readLine();
            ao.ma1(bo, Integer.parseInt(inData1));
        } catch (IOException ex) {
           System.out.println(ex.getMessage());
        }
    }
}
```

IF

Sequence Diagrams



- Αλληλεπίδραση αντικειμένων
  - πλαίσια λογικής ελέγχου (if)
    - εναλλακτικές αλληλεπιδράσεις
    - εκτελείται μόνο αυτή για την οποία ισχύει η συνθήκη

Sequence Diagrams

```
public class AClass {
    public int aa1;
    public float aa2;

    public AClass(int a1, float a2){
        aa1 = a1;
        aa2 = a2;
    }
    public void ma1(BClass bo, int flag){
        while(flag >= 0){
            bo.mb1(flag);
            flag--;
        }
    }
    public void ma2(){
        System.out.println(aa1+aa2);
    }
}
```

επαναλαμβανόμενες αλληλεπιδράσεις

Sequence Diagrams

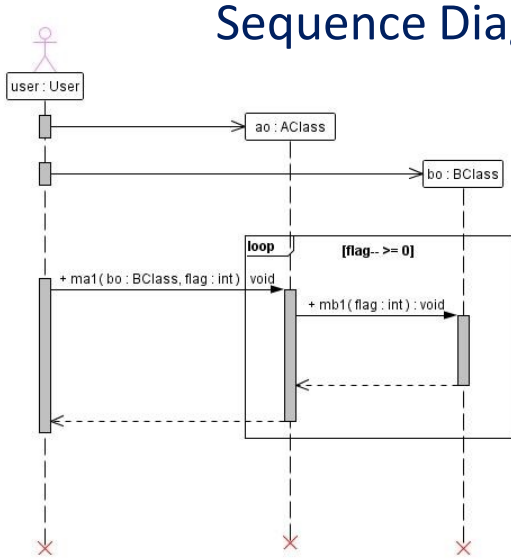
```
public class BClass {
    public String ab1;
    public String ab2;

    public BClass(String a1, String a2){
        ab1 = a1;
        ab2 = a2;
    }
    public void mb1(int flag){
        System.out.println(ab1 + flag);
    }
    public void mb2(AClass ao){
        System.out.println(ab2);
        ao.ma2();
    }
}
```


Sequence Diagrams

```
public class Main {
   public static void main(String[] args) {
        InputStreamReader in = new InputStreamReader(System.in);
        BufferedReader br = new BufferedReader(in);
        String inData1, inData2;
        try {
            inData1 = br.readLine();
            inData2= br.readLine();
            AClass ao = new AClass(Integer.parseInt(inData1),
                                   Integer.parseInt(inData2));
            inData1 = br.readLine();
            inData2= br.readLine();
            BClass bo = new BClass(inData1, inData2);

            inData1 = br.readLine();
            ao.ma1(bo, Integer.parseInt(inData1));
        } catch (IOException ex) {
           System.out.println(ex.getMessage());
        }
    }
}
```

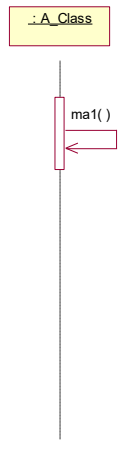
While

Sequence Diagrams



- Αλληλεπίδραση αντικειμένων
  - πλαίσια λογικής ελέγχου
    - επαναλαμβανόμενες αλληλεπιδράσεις
  - αλλά πλαίσια
    - neg → Μη έγκυρη αλληλεπίδραση
    - opt → Αλληλεπίδραση που εκτελείται προαιρετικά αν ισχύει μια συνθήκη
    - par → παράλληλη εκτέλεση αλληλεπιδράσεων

Sequence Diagrams



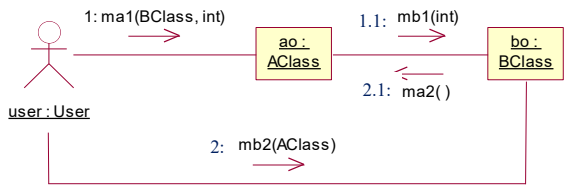
- Αλληλεπίδραση αντικειμένων
  - ανώνυμα αντικείμενα
  - αυτοκλήση

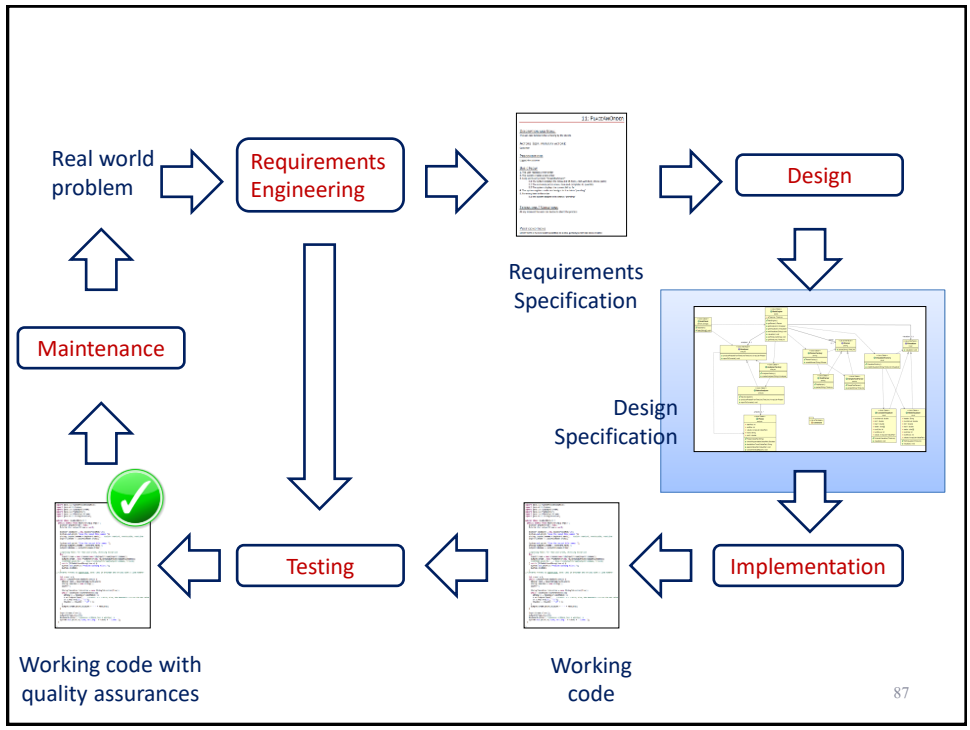
Communication Diagrams

- Ο στόχος είναι ο ίδιος με τα sequence diagrams – απεικόνιση της αλληλεπίδρασης μεταξύ αντικειμένων για την υλοποίηση των περιπτώσεων χρήσης του λογισμικού
- Ο τρόπος απεικόνισης είναι διαφορετικός.
  - Αντικείμενα.
  - Στιγμιότυπα συσχετίσεων μεταξύ των αντικειμένων.
  - Κλήσεις μεθόδων - μηνύματα.
    - Η χρονική σειρά των κλήσεων καθορίζεται με ένα αύξοντα αριθμό που χαρακτηρίζει την κάθε κλήση – μήνυμα.
      - Ένθετη αρίθμηση για να δείξουμε ότι μια κλήση γίνεται κατά την εκτέλεση μιας άλλης κλήσης
        - » 1, 1.1, 1.2, 2, 2.1, …..

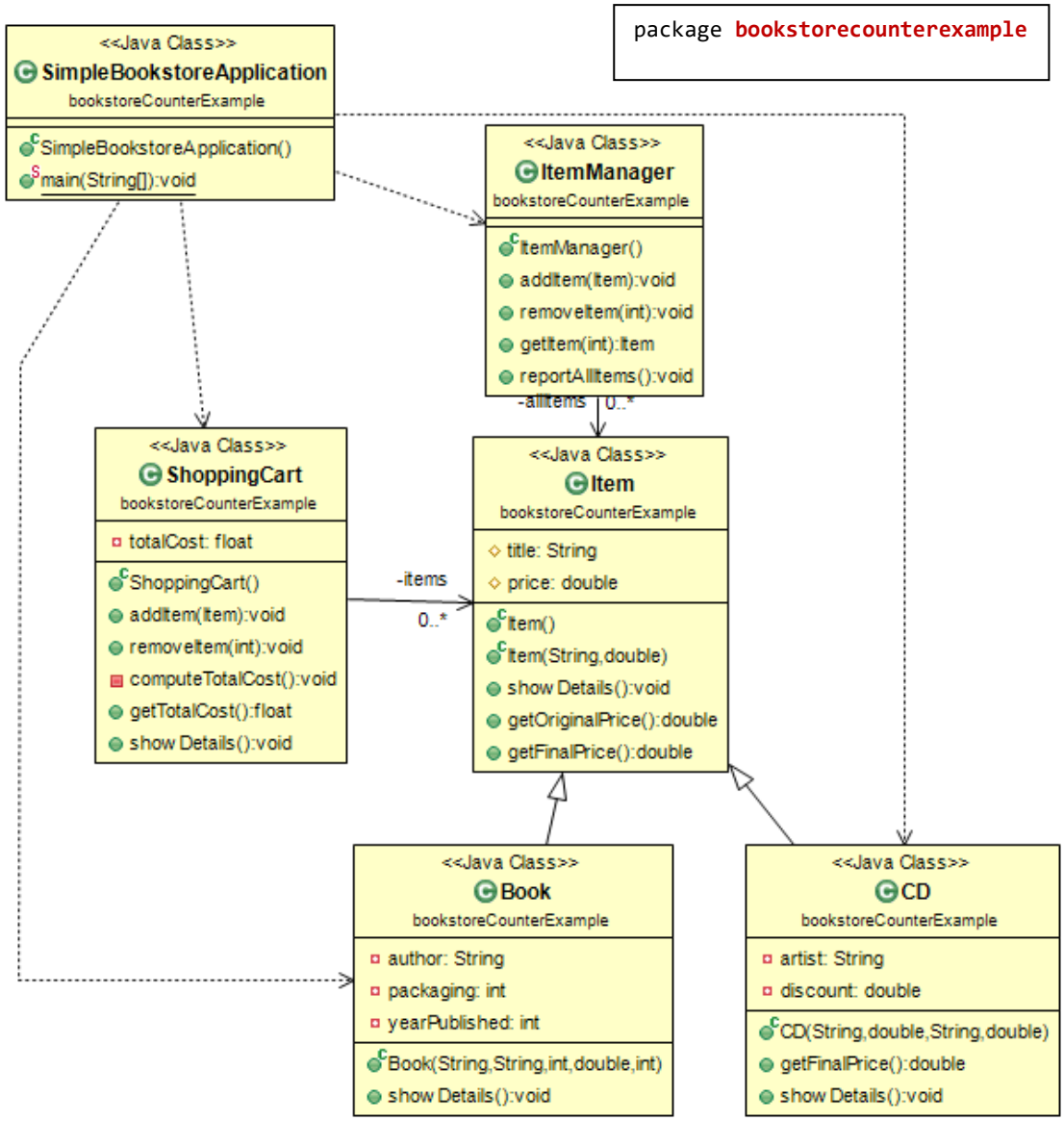
Collaboration Diagrams

- Αλληλεπίδραση αντικειμένων



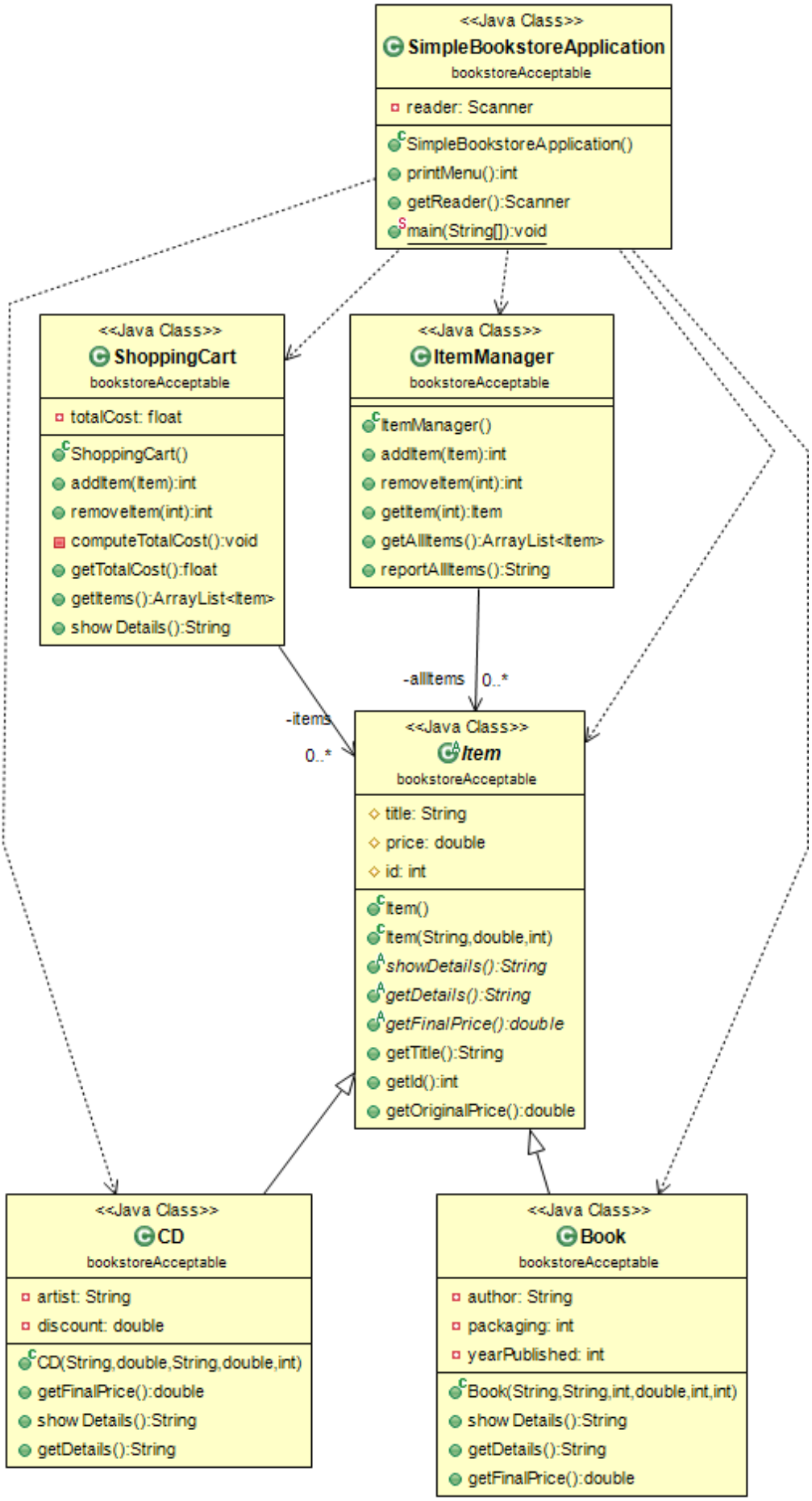

Real world problem
Requirements Engineering
Design
Requirements Specification
Maintenance
Design Specification
Testing
Implementation
Working code with quality assurances
Working code
87

Zooming into some of the previous diagrams



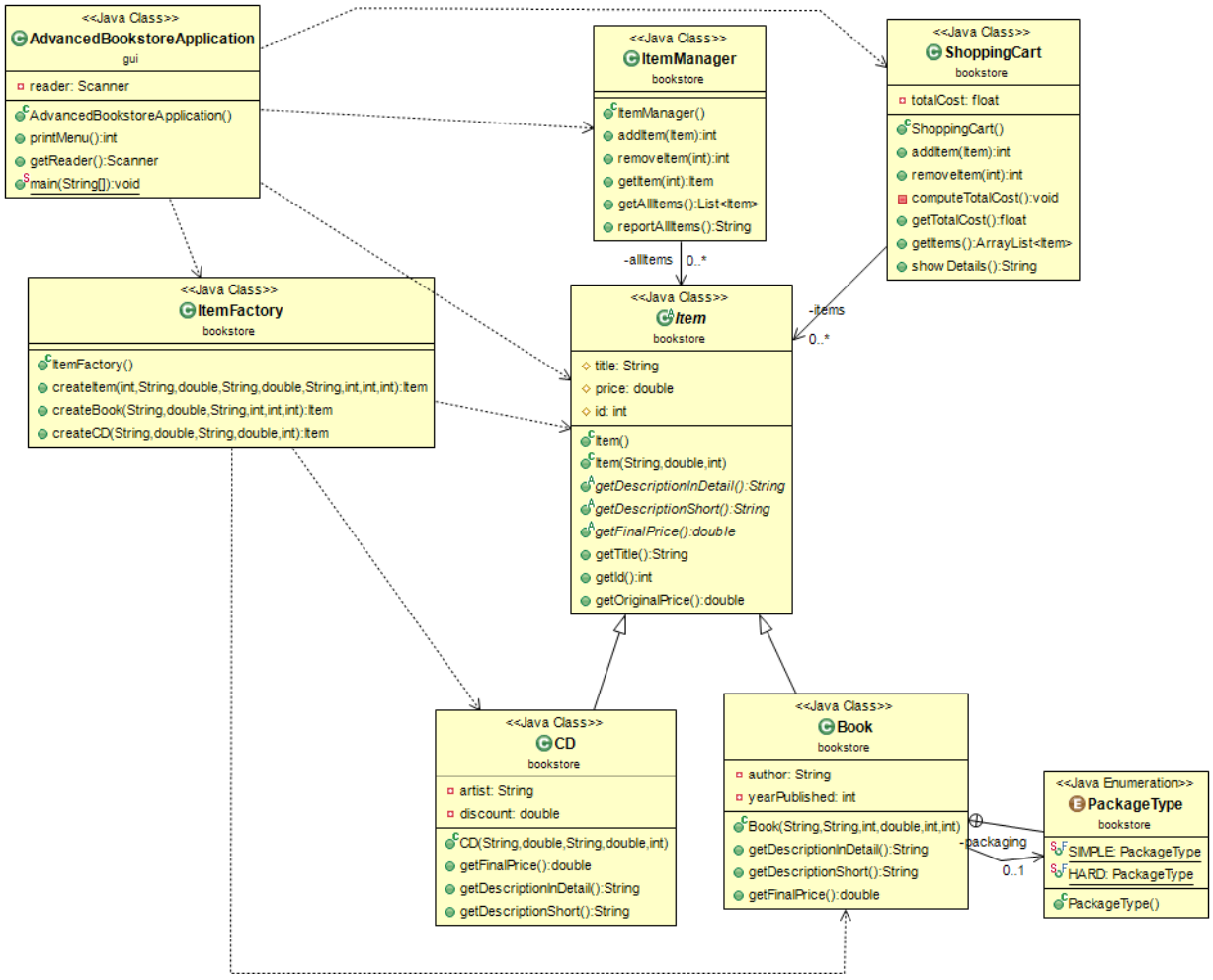

```
package bookstorecounterexample;

public class Item {
...
}
public class Book extends Item {
...
}
public class CD extends Item {
...
}
public class ItemManager {
     private ArrayList<Item> allItems;
...
}
```

```
package bookstoreAcceptable
```



```
package bookstoreAcceptable;

public abstract class Item {
...
       public abstract double getFinalPrice();
}
public final class Book extends Item {
...
       @Override
       public double getFinalPrice() {
              return price;
       }
}
public final class CD extends Item {
...
@Override
       public double getFinalPrice() {
          return (price - discount);
       }
}
public class ItemManager {
     private List<Item> allItems;
...
```

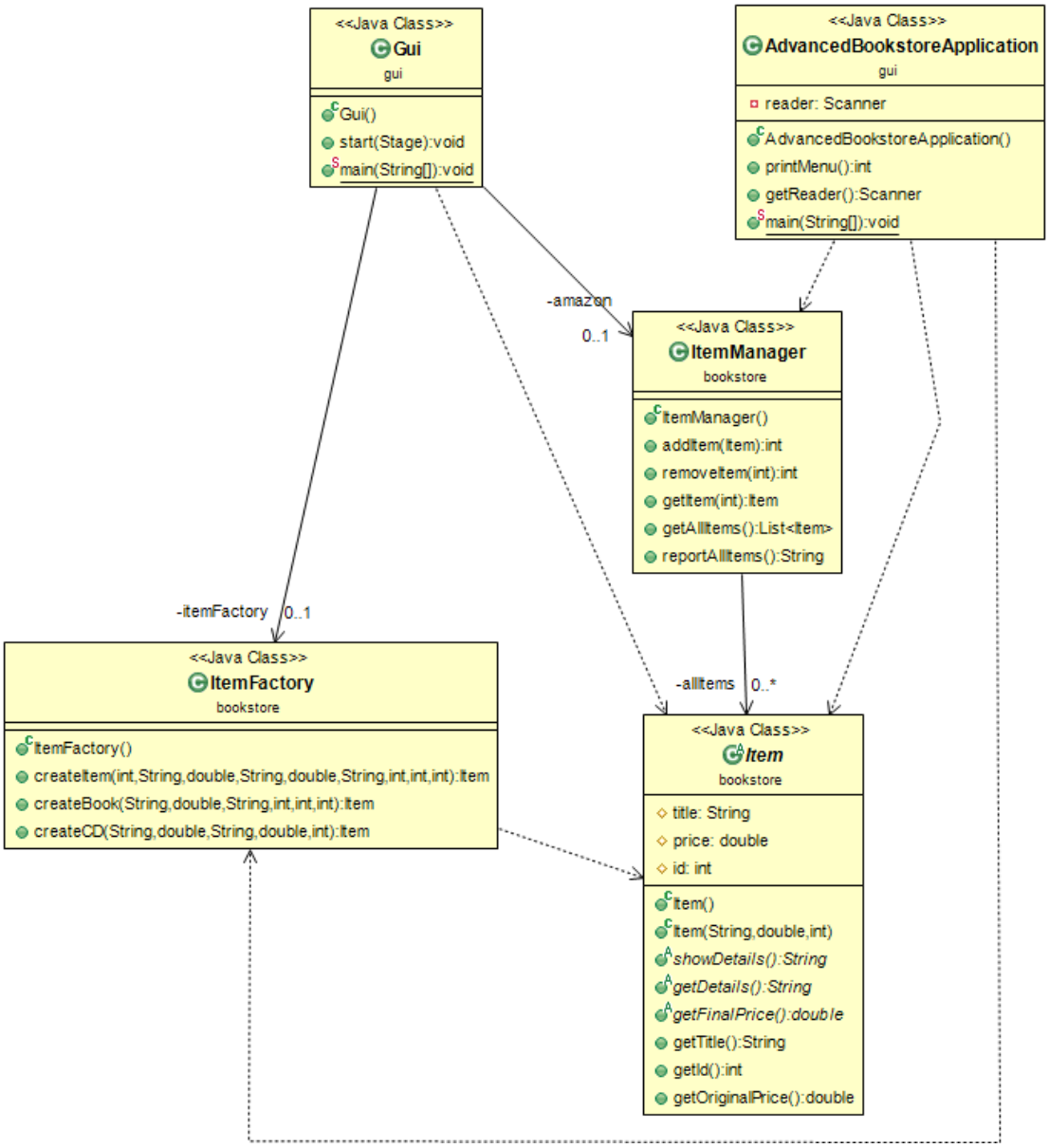
package **bookstore**




```
package bookstore;

public abstract class Item {
...
        public abstract double getFinalPrice();
}
public final class Book extends Item {
...
        @Override
        public double getFinalPrice() {
                return price;
        }
}
public final class CD extends Item {
...
        @Override
        public double getFinalPrice() {
           return (price - discount);
        }
}
public final class ItemManager {
     private List<Item> allItems;
...
```

```
public class ItemFactory {
        public Item createBook(String aTitle,
                        double aPrice, String
                        anAuthor, int aDate, int
                        aPackage,int id)
        public Item createCD(String aTitle,
                        double aPrice, String
                        anArtist, double
                        aDiscount, int id)
}
```

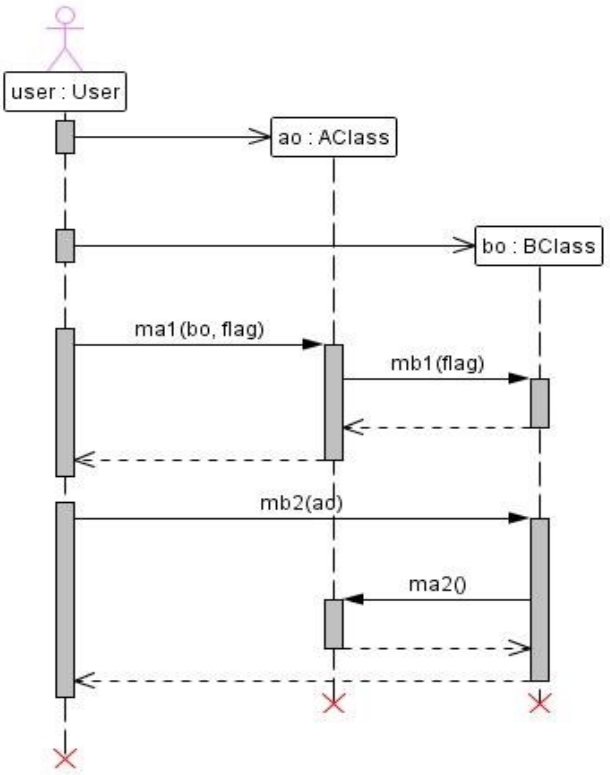
Why an Item Manager is useful



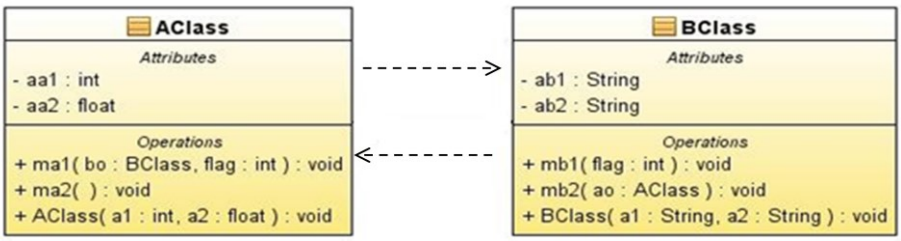
Observe how **_all_** client classes (the ones interacting with the user), i.e., both class **Gui** and class AdvancedBookstoreApplication, need to invoke methods of **only** ItemManager, i.e., avoiding calls to several other classes.

Also observe that ItemManager is also the data holder class (which is reasonable, esp., in applications of this small size)

```
public class AClass {
	public int aa1;
	public float aa2;
	public AClass(int a1, float a2){
		aa1 = a1;
		aa2 = a2;
	}
	public void ma1(BClass bo, int flag){
		bo.mb1(flag);
	}
	public void ma2(){
		System.out.println(aa1+aa2);
	}
}

public class BClass {
	public String ab1;
	public String ab2;
	public BClass(String a1, String a2){
		ab1 = a1;
		ab2 = a2;
	}
	public void mb1(int flag){
		System.out.println(ab1 + flag);
	}
	public void mb2(AClass ao){
		System.out.println(ab2);
		ao.ma2();
	}
}
```

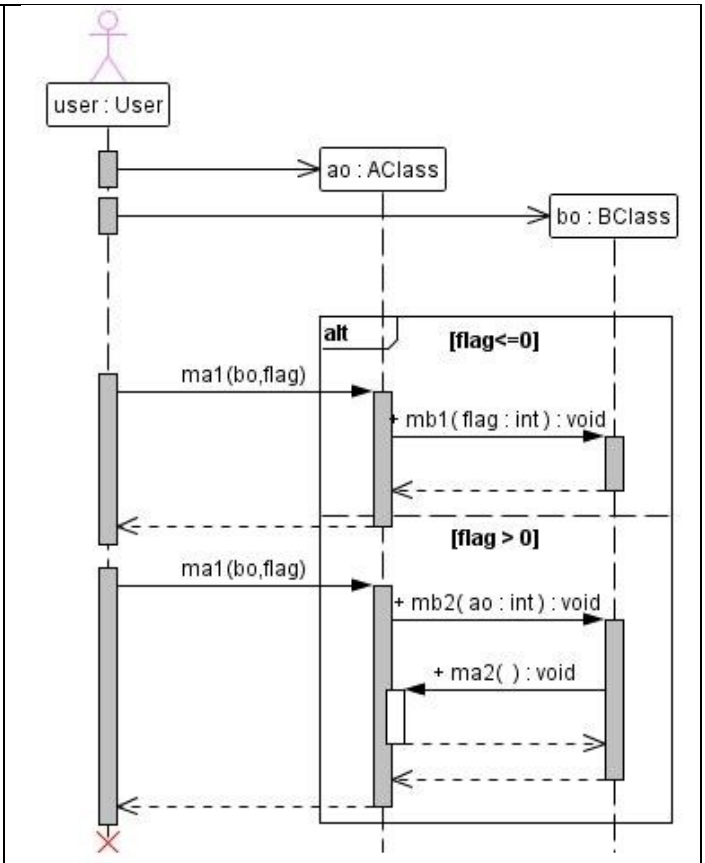



```
public class Main {
	public static void main(String[] args) {
		InputStreamReader in = new InputStreamReader(System.in);
		BufferedReader br = new BufferedReader(in);
		String inData1, inData2;
		try {
			inData1 = br.readLine();
			inData2= br.readLine();
			AClass ao = new AClass(Integer.parseInt(inData1),
					Integer.parseInt(inData2));
			inData1 = br.readLine();
			inData2= br.readLine();
			BClass bo = new BClass(inData1, inData2);
			inData1 = br.readLine();
			ao.ma1(bo, Integer.parseInt(inData1));
			bo.mb2(ao);
		} catch (IOException ex) {
			System.out.println(ex.getMessage());
		}
	}
}
```



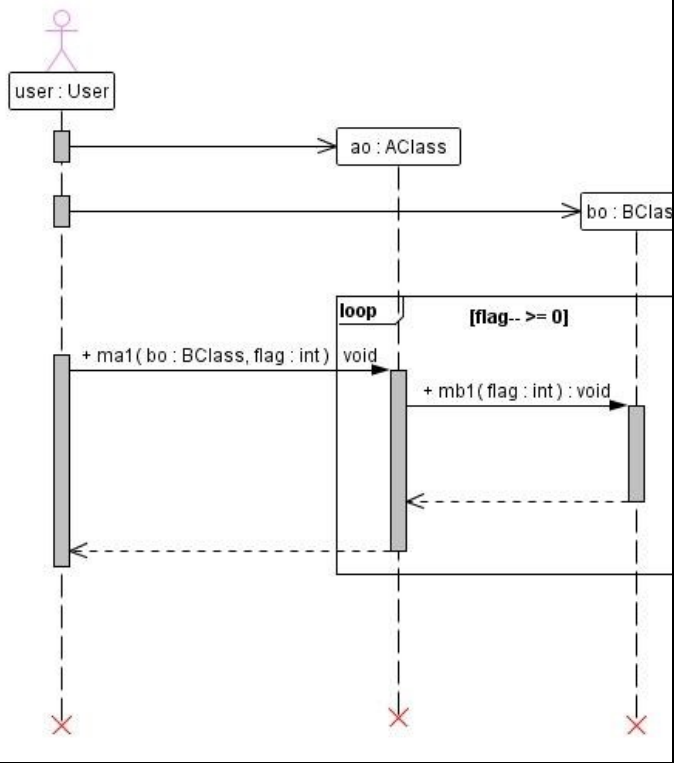
```
public class AClass {
        public int aa1;
        public float aa2;
        public AClass(int a1,
float a2){
                aa1 = a1;
                aa2 = a2;
        }
        public void ma1(BClass
bo, int flag){
                if(flag >= 0)
                        bo.mb1(fl
        ag);
                else
                        bo.mb2(th
        is);

        }
        public void ma2(){
                System.out.print
        ln(aa1+aa2);
        }
}
```

@main: we invoke only
ao.ma1(bo,
Integer.parseInt(inData1));




```
public class AClass {
        public int aa1;
        public float aa2;
        public AClass(int a1,
float a2){
                aa1 = a1;
                aa2 = a2;
        }
        public void ma1(BClass
bo, int flag){
                while(flag >=
        0){
                        bo.mb1(fl
        ag);
                        Flag--;
                }
        }
        public void ma2(){
                System.out.print
        ln(aa1+aa2);
        }
}
```

@main: we invoke only
ao.ma1(bo,
Integer.parseInt(inData1));



ΣΗΜΕΙΩΣΕΙΣ

Ακολουθεί ένα παράδειγμα για το πώς

- δοθείσης μιας περιγραφής ενός προβλήματος που έχουμε καταγράψει σε φυσική γλώσσα,
- αρχικά εξάγονται κάποια use cases, και στη συνέχεια,
- εξάγουμε τις κλάσεις ανάλυσης και κάποια sequence diagrams.

Εστιατόριο στην Άκρη του Σύμπαντος

Προσοχή: η εκφώνηση έχει επίτηδες «λάθη» σε πολλά σημεία της, για να τα συζητήσουμε στο μάθημα

Στο εστιατόριο «ToDokimasesPrinToBgaleisStonKosmo?» έξω από τους δακτυλίους του Κρόνου, οι παραγγελίες γίνονται ηλεκτρονικά.

Μόλις μια παρέα πελατών κάτσει σε ένα τραπέζι, κάποιος από την παρέα κάνει log-in με τον κωδικό του και το password του (που έλαβε στην είσοδο). Το τραπέζι επικοινωνεί με το σύστημα του εστιατορίου και αν τα στοιχεία είναι ΟΚ, ενεργοποιεί το σύστημα του τραπεζιού, που πλέον λειτουργεί ως ο αντιπρόσωπός του στο ηλεκτρονικό σύστημα του εστιατορίου.

Αν αυτό έχει γίνει επιτυχώς, οποιαδήποτε στιγμή ο πελάτης μπορεί να ξεκινήσει μια παραγγελία. Μέχρι ο πελάτης να πατήσει «Νισάφι πια», κάθε φορά το σύστημα, του δείχνει τον κατάλογο, αυτός διαλέγει ένα από τα φαγητά του καταλόγου, καθώς και την ποσότητα, και το σύστημα του παρουσιάζει όλα όσα έχει ήδη επιλέξει, καθώς και το συνολικό κόστος. Μόλις ο χρήστης πατήσει το προαναφερθέν κουμπί, το σύστημα καταχωρεί την παραγγελία με status «εκκρεμεί». Το ίδιο[1] γίνεται για όλα τα επί μέρους πιάτα μιας παραγγελίας.

Ο μάγειρας στην κουζίνα βλέπει όλες τις μη ολοκληρωμένες παραγγελίες και παρασκευάζει ότι εκκρεμεί. Για κάθε κομμάτι παραγγελίας[2] που έχει ετοιμάσει, αντί να φωνάξει «σέρβιιιιις!!!!», ο μάγειρας ενημερώνει το σύστημα ότι είναι διαθέσιμο προς σερβίρισμα, το οποίο παίρνει και την status «ΜολώνΛαβέ». Επειδή το μαγαζί έχει προηγμένα σερβίτσια με RFID tags, στο σύστημα καταχωρείται και το ID του πιάτου που περιέχει το παρασκευασθέν φαγητό. Το σύστημα προβάλει στην οθόνη του τραπεζιού ότι το φαγητό είναι έτοιμο και ο πελάτης πάει και το παίρνει μόνος του από τον πάγκο. Μόλις ακουμπήσει στο τραπέζι, το RFID σύστημα του τραπεζιού αλλάζει την κατάσταση του κομματιού της παραγγελίας σε «σερβιρισμένο» και ελέγχει αν είναι το τελευταίο της παραγγελίας. Αν ναι, η παραγγελία παίρνει status «παραδοθείσα».

Όταν οι πελάτες τελειώσουν, κλικάρουν checkout στην οθόνη του τραπεζιού. Το σύστημα υπολογίζει και δείχνει το λογαριασμό, ζητά τα στοιχεία της πιστωτικής κάρτας του πελάτη, επικοινωνεί με τη διαστημική τράπεζα και χρεώνει τον πελάτη. Η παραγγελία παίρνει status «πληρωθείσα».

[1] Τι θα πει «το ίδιο»?

[2] Ελέγξτε τι θα γίνει στο κείμενο από πλευράς σαφήνειας αν αντί για «κομμάτι παραγγελίας» χρησιμοποιήσουμε τον όρο «πιάτο». Επίσης: υπάρχουν συνώνυμα στο κείμενο?

01: CUSTOMERAUTHENTICATION

DESCRIPTION AND GOAL:

This use case facilitates the login by the clients.

DEADLY ERROR!

Always an **ACTIVE VERB!**

Somewhere in the code you will need a **method** with name

`authenticateCustomer`

ACTORS (ESP. PRIMARY ACTOR):

Customer

PRECONDITIONS

Customer must have obtained login / password either at the door, or when reserving a table

BASIC FLOW

1. The user case begins when a customer enters login and password
2. The system validates login and password
3. If Step 2 is successful
    3.1 the system activates the table

EXTENSIONS / VARIATIONS

1. At step 3 if the user fails to log-in, the entire process restarts

POST CONDITIONS

MISSING: some **significant result!**

E.g., “At the end of the use case, the table has been activated”

SPECIAL REQUIREMENTS, ISSUES, RISKS AND OTHER COMMENTS

11: PLACEANORDER

DESCRIPTION AND GOAL:

This use case facilitates the ordering by the clients.

ACTORS (ESP. PRIMARY ACTOR):

Customer

ERROR!

Always " The use case begins when …"

PRECONDITIONS

Logged-in customer

BASIC FLOW

1. The user requests a new order
2. The system creates a new order
3. Loop until user presses "OxiAlloKarbouno"
    - 3.1 The system displays the menu (list of items, each with text, photo, price)
    - 3.2 The customer picks a menu item and completes its quantity
    - 3.3 The system displays the current bill so far
4. The system registers order and assigns to it a status "pending"
5. For every item in the order
    - 5.1 the system assigns to it a status "pending"

EXTENSIONS / VARIATIONS

At any moment the user can decide to abort the process

POST CONDITIONS

Either there is no new order (aborted) or a new, pending order has been created

SPECIAL REQUIREMENTS, ISSUES, RISKS AND OTHER COMMENTS

21: DeliverPlateForService

Description and Goal

The goal of this use case is to notify customers that a plate of their order is ready for them

Actors (esp. primary actor)

RFID reader of bench
RFID reader of table
Chef

Consider (not obligatorily) splitting in two:

(a) chef delivers plate to bench

(b) plate touches table

Preconditions

Basic Flow

1. The use case starts when the chef updates an order's item with a status update "ComeNGetIt" and assigns it the ID of the plate as obtained by the RFID of the chef's bench
2. The system updates the screen of the table and shows the item in red background with a tag "ComeNGetIt"
3. If the plate touches the table
    3.1 the system sets the status of the item as "Served"
4. If there are no "pending" items in the order
        4.1 the system assigns to it a status "served"

Who is the actor? The plate, or the RFID reader of the table?

Also: what happened to the RFID reader of the bench? Is it used anywhere?

Extensions / Variations

Post conditions

Special Requirements, Issues, Risks and other Comments

31: CHECKOUT

DESCRIPTION AND GOAL

The use case facilitates the payment of the bill by the customers via the interaction with the bank of their credit card

ACTORS (ESP. PRIMARY ACTOR)

Customer (primary)
Bank

PRECONDITIONS

The table from which the request to checkout comes has a customer logged-in, an order related to it and at least one item served.

BASIC FLOW

1. The use case begins when the client clicks “checkout” on his table’s screen.
2. The system retrieves the order related to the table and the price to be paid for it.
3. while (order has not been paid)
    - 3.1The system shows the amount of money to be paid and asks for the data of the customer’s credit card
    - 3.2. The customer enters the data of his credit card
    - 3.3. The system communicates with the bank to charge for the amount.
    - 3.4. If bank returns OK the order takes status “paid”

4 ~~a receipt is given to the customer~~ The system prints a receipt for the customer with the details of the transaction

EXTENSIONS / VARIATIONS

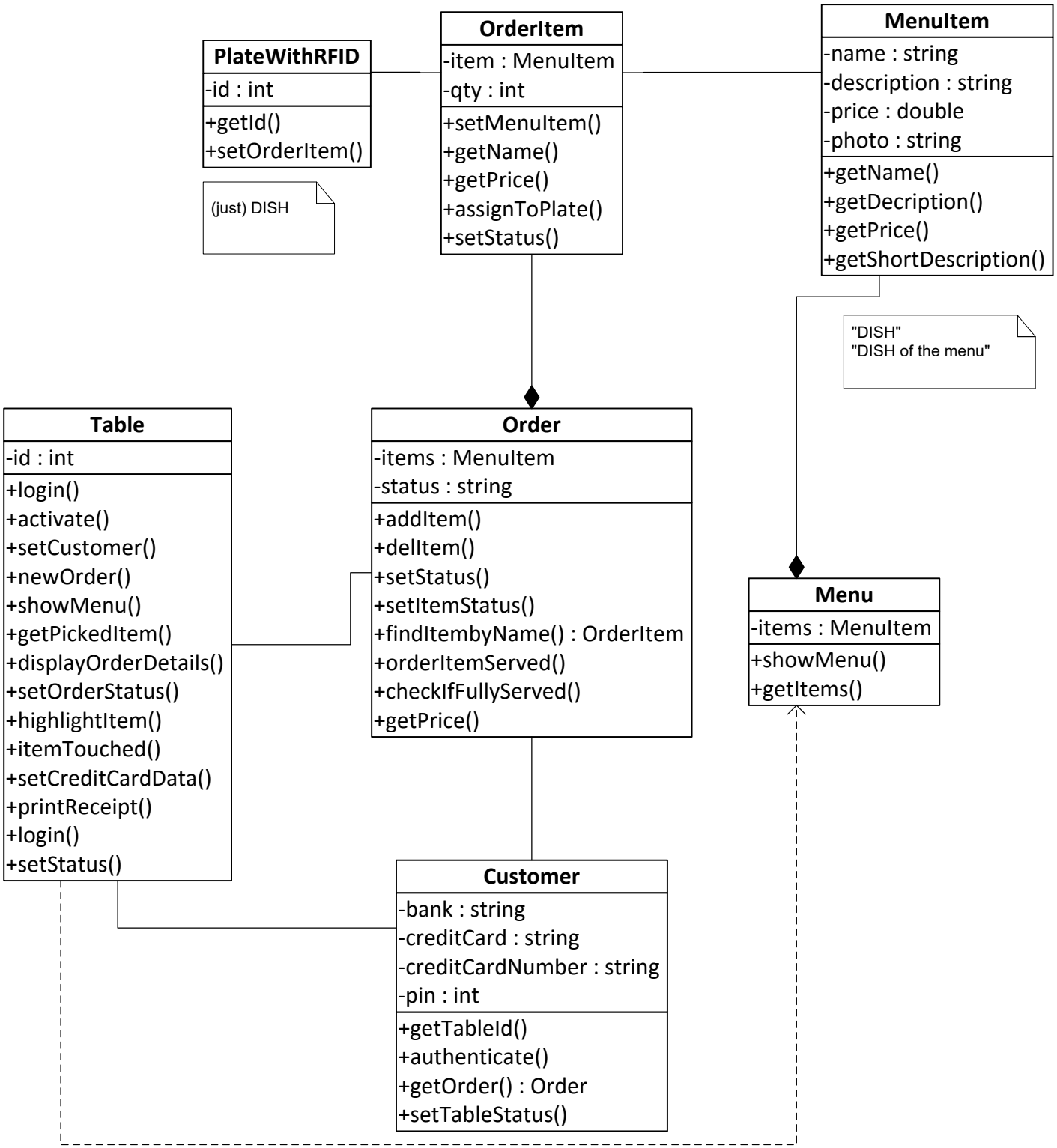
HandleArguingCustomer: At step 3, the client disagrees with the amount of money asked to pay
HandleInsufficientMoney: The bank reports “notEnoughMoney” at step 3.4, and the client must either give another card, or, wash the dishes
HandleWrongData: The bank reports “wrongData” at step 3.4, and the client must try again
HandleMissingOrder: no order is related to the table

POST CONDITIONS

At the end of the use case, either the order has been paid, or the customer washes the dishes, or a record is placed in the restaurant’s blacklist with a request to send the customer to space, lost without trace and having no chance of getting away

SPECIAL REQUIREMENTS, ISSUES, RISKS AND OTHER COMMENTS

Customers claim they would like to pay only for what has been served and not for what has been ordered. How do you facilitate this, by modifying your use case diagrams, classes and use cases?

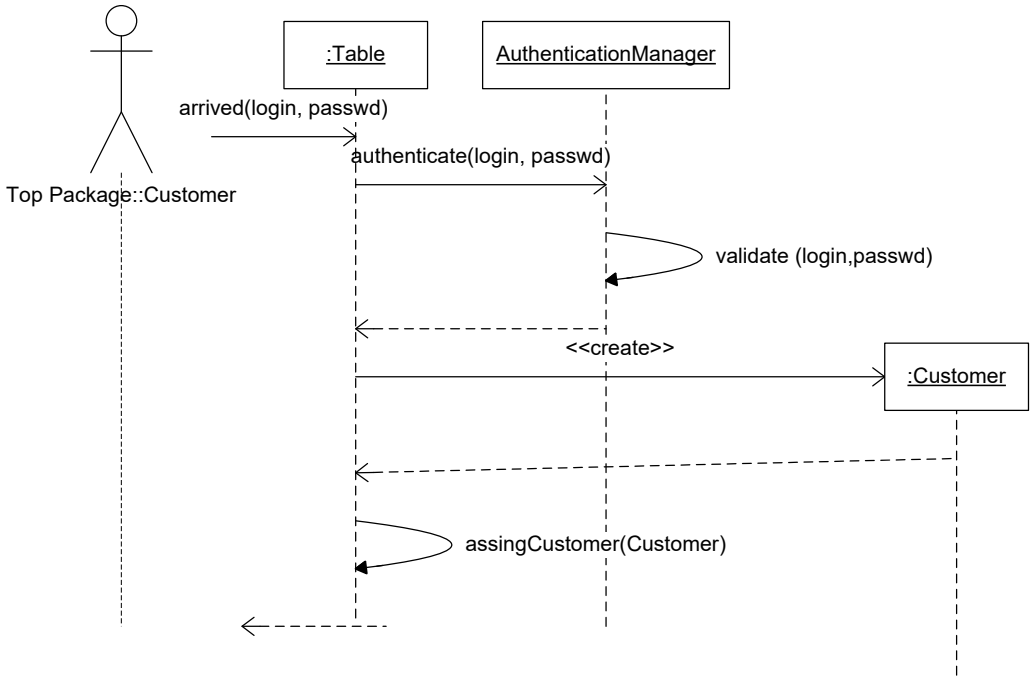


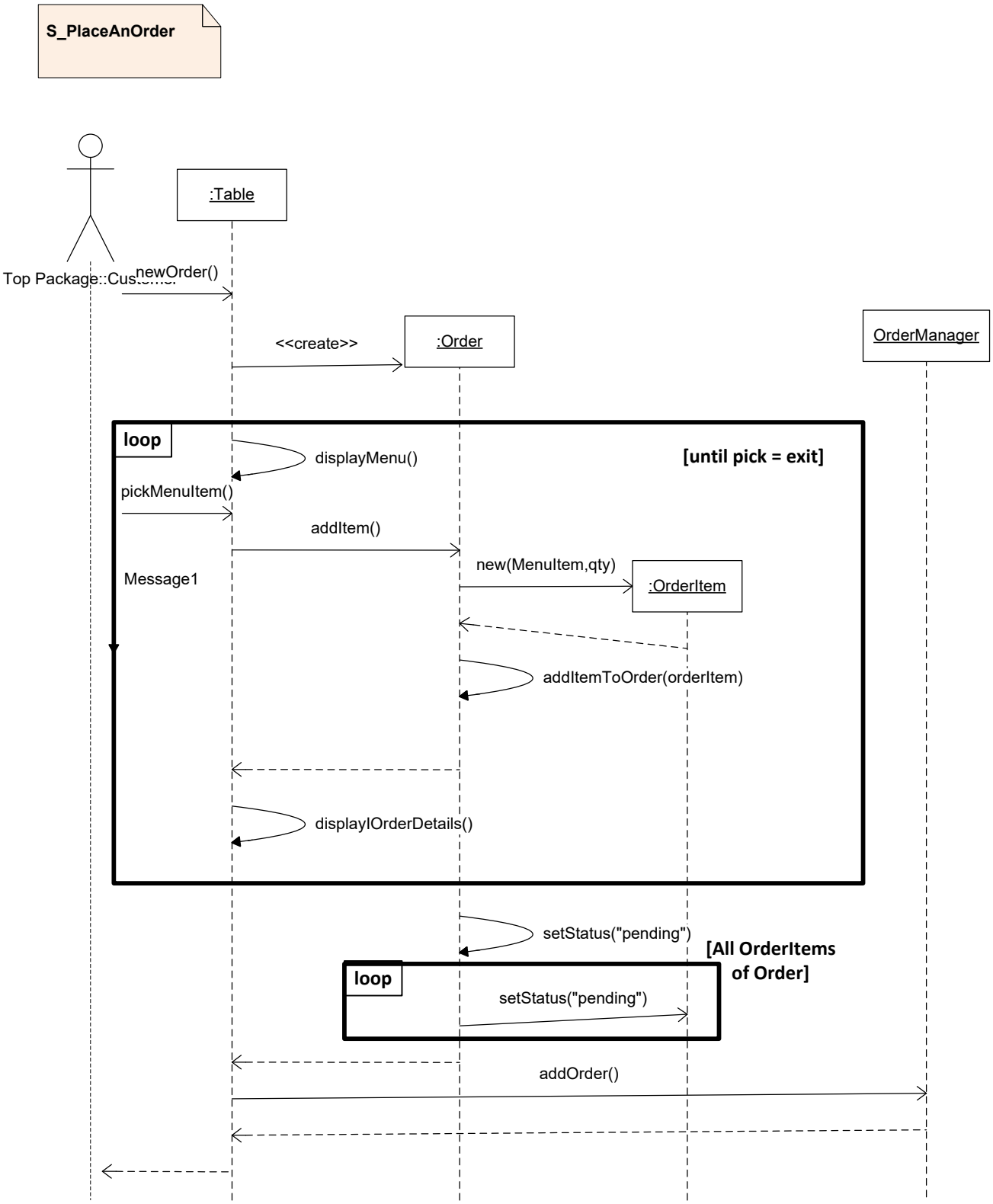
Coarse grained analysis model, reflecting the basic entities of the requirements

MUST revisit!

S_Authenticate Customer

Nobody really uses the activation rectangle...
It's OK to skip

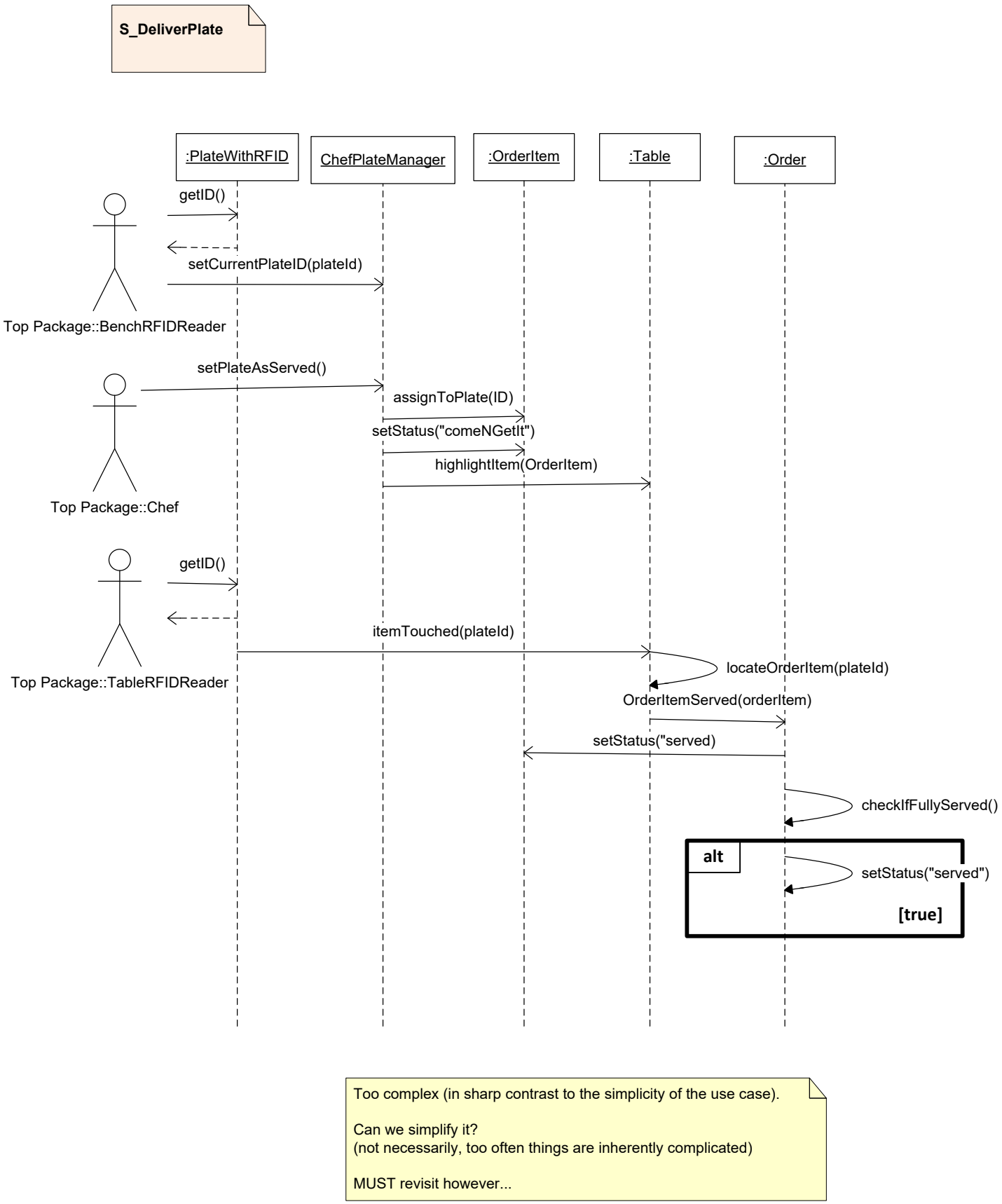




What's wrong with this sequence?

Is it the job of the Table to "create" Orders and to add them to the OrderManager?

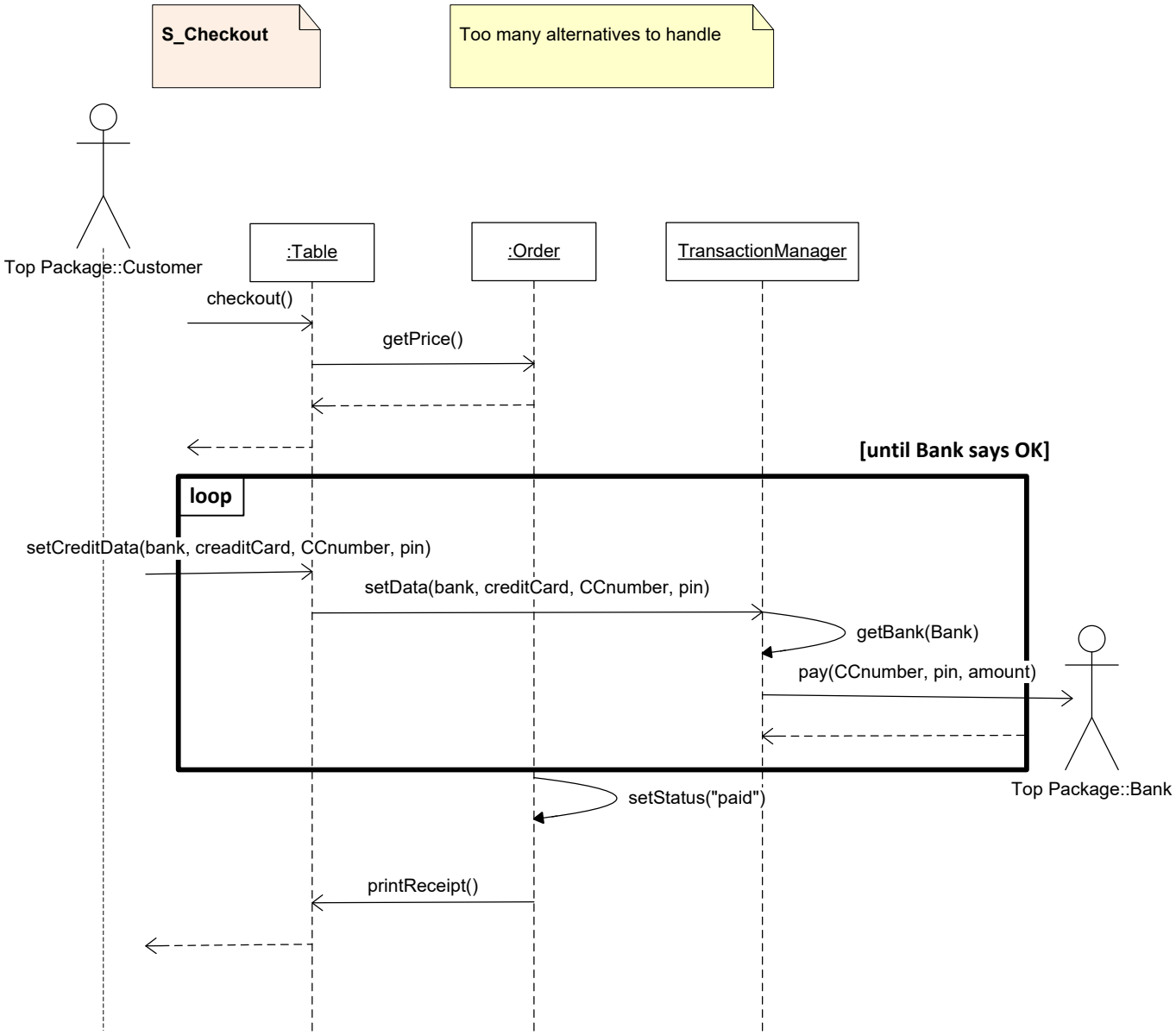
... the Manager who should really be the center of this, learns the news last...



Too complex (in sharp contrast to the simplicity of the use case).

Can we simplify it?
(not necessarily, too often things are inherently complicated)

MUST revisit however...


S_Checkout
Too many alternatives to handle
Top Package::Customer
:Table
:Order
TransactionManager
checkout()
getPrice()
[until Bank says OK]
loop
setCreditData(bank, creaditCard, CCnumber, pin)
setData(bank, creditCard, CCnumber, pin)
getBank(Bank)
pay(CCnumber, pin, amount)
Top Package::Bank
setStatus("paid")
printReceipt()

Αντικειμενοστρεφής Ανάλυση και Σχεδίαση

Ανάπτυξη Λογισμικού (Software Development)

www.cs.uoi.gr/~pvassil/courses/sw_dev/

ΜΥΥ301 / ΠΛΥ 308

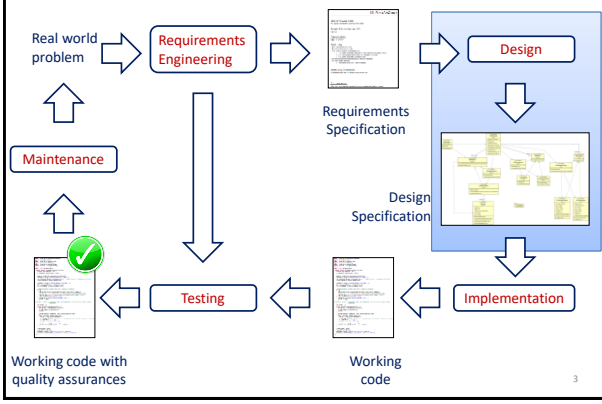
Σχεδίαση

- Η σχεδίαση λαμβάνει ως είσοδο τις συγκροτημένες απαιτήσεις των χρηστών ενός (υπο)συστήματος και **παράγει μια αφαιρετική αναπαράσταση του πώς δομείται το λογισμικό εσωτερικά**, με στόχο να ανταποκριθεί στις απαιτήσεις αυτές

Leslie Lamport, Turing Award Recipient, 2013

Leslie Lamport. Who Builds a House without Drawing Blueprints? Comm. ACM, 58(4), pp. 38-41, Apr. 2015

http://cacm.acm.org/magazines/2015/4/184705-who-builds-a-house-without-drawing-blueprints/



Τι / Γιατί / Πώς

- Στην παρούσα ενότητα θα υποστηρίξουμε τη διαδικασία σχεδίασης με τη βασική **μεθοδολογία** οργάνωσης του κώδικα που έχουμε για το αντικειμενοστρεφές λογισμικό
  / μαθαίνουμε να πριονίζουμε, τώρα που ξέρουμε τι είναι το πριόνι */*

Η ανάλυση και η σχεδίαση (συχνά κάτω από τον ενιαίο όρο **σχεδίαση**) είναι τα **βασικά σχεδιαστικά εργαλεία** που θα χρησιμοποιήσουμε

- Τελειώνοντας αυτή την ενότητα, θα πρέπει να είστε εις θέση αφενός να κατανοείτε, αλλά και να οργανώσετε τον κώδικα σε κλάσεις, interfaces, πακέτα και υποσυστήματα

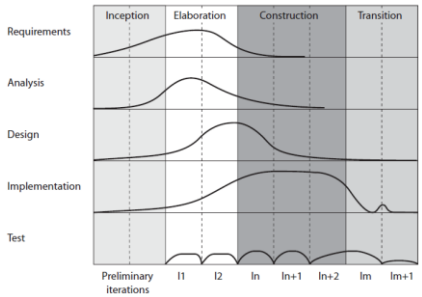
4

Σχεδίαση: in fact, ανάλυση και σχεδίαση

- Στη φάση της ανάλυσης ενδιαφερόμαστε για το **τι** πρέπει να κάνει το σύστημα (αντί για το πώς ή το γιατί) και δουλεύουμε με κλάσεις που αφορούν στο πεδίο του προβλήματος (και όχι με τεχνικές λεπτομέρειες της λύσης όπως π.χ., διασυνδέσεις με άλλα υποσυστήματα)
- Η σχεδίαση σκοπό έχει να εισάγει την απαραίτητη λεπτομέρεια στο μοντέλο μας – γι' αυτό και η έμφαση είναι πιο πολύ στα interfaces & susbsystems
- Η ανάλυση λέει τι κάνει το σύστημα και η σχεδίαση πώς το κάνει

5

Unified Process



Από Arlow & Neustadt, 2002

6

Unified Process

	Elaboration (partial but working v. of the system)	**Construction** (complete R/A/D, first build & test)
Req's (what the system should do)	Refine req's	Complete it
Analysis (req's structured in first cut classes & use case realizations)	**What to build**	Complete it
Design (System Architecture)	**Stable architecture**	Complete it
Implementation (build the SW)	Build the SW baseline	Build Initial Operational Capability
Test (test the SW)	Test the SW baseline	Alpha test

Inception (Goals, objectives, risks)

Transition (deploy to users, debug, beta test)

7

Παν μέτρον άριστον

- Η ανάλυση και η σχεδίαση είναι εργαλεία για να υλοποιήσουμε τα συστήματα και **όχι αυτοσκοπός**
- Αυτό που τελικώς μετρά είναι να παραδώσουμε ένα πρόγραμμα που τρέχει όπως πρέπει και μπορεί να συντηρηθεί στο μέλλον με ευκολία =>
- Η προσπάθεια που καταβάλουμε στην ανάλυση και στη σχεδίαση ΔΕΝ ΜΠΟΡΕΙ ΝΑ ΕΙΝΑΙ ΥΠΕΡΒΟΛΙΚΗ
  - *Συν το χρόνω, θα μπορείτε να κρίνετε καλύτερα πότε μια σχεδίαση είναι αρκετά ώριμη ώστε να αποτελεί μια σταθερή αρχιτεκτονική βάση του συστήματος και πλέον μπορούμε να προχωρήσουμε στην υλοποίηση…*

8

Roadmap

- Analysis
- Design
- Packages
- Subsystem design
- Interfaces
- Πρώτα, όμως, θα δώσουμε μια συνοπτική, πρακτική μέθοδο για την μετάβαση από απαιτήσεις σε μια σχεδίαση του συστήματος…

9

Μετατροπή των απαιτήσεων του προβλήματος σε αρχικές κλάσεις που αντανακλούν τα βασικά αντικείμενα του πραγματικού κόσμου τα οποία μοντελοποιούμε

ΑΝΆΛΥΣΗ ΠΡΟΒΛΉΜΑΤΟΣ

Φάση Ανάλυσης

- **Ανάλυση του προβλήματος**, όχι του συστήματος
- Η διαδικασία της ανάλυσης παράγει δύο βασικά αποτελέσματα
  - Κλάσεις επιπέδου ανάλυσης
  - Υλοποιήσεις use cases
- Θα χρησιμοποιούμε τον όρο «**μοντέλο**» του συστήματος για το συνδυασμό των παραπάνω αποτελεσμάτων.

Φάση Ανάλυσης

- Στη φάση της ανάλυσης ενδιαφερόμαστε για το **τι** πρέπει να κάνει το σύστημα (αντί για το πώς ή το γιατί) και δουλεύουμε με κλάσεις που αφορούν στο πεδίο του προβλήματος (και όχι με τεχνικές λεπτομέρειες της λύσης όπως π.χ., διασυνδέσεις με άλλα υποσυστήματα)
- Βασική προϋπόθεση, λοιπόν, είναι οι παραγόμενες κλάσεις να αναπαριστούν ξεκάθαρα τις οντότητες του πραγματικού κόσμου (business concepts) που μοντελοποιούμε (π.χ., Customer, Product, …) , ο οποίος και αποτελεί το πεδίο του προβλήματος που καλούμεθα να επιλύσουμε

Τελικό παραδοτέο

- Μοντέλο του συστήματος σε επίπεδο ανάλυσης είναι:
  - οι σχετικές **κλάσεις** με πεδία, μεθόδους και συσχετίσεις μεταξύ τους, και τα συνεπαγόμενα **διαγράμματα κλάσεων**
  - η σχεδίαση της **δυναμικής συμπεριφοράς** του συστήματος (π.χ., μέσω sequence diagrams) ώστε να καλύπτει όλα τα use cases
  - [για μεγάλα μοντέλα] η οργάνωση uses cases, class and sequence diagrams σε **πακέτα ανάλυσης**

13

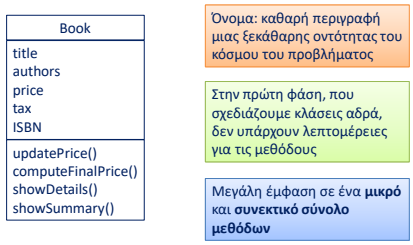
Ανατομία μιας κλάσης ανάλυσης

- Όνομα
- Σημαντικά (private) πεδία
- Σημαντικές αρμοδιότητες = δημόσιες μέθοδοι

- Δε χρειάζεται τόσο πολύ έμφαση
  - στις λεπτομέρειες ορατότητας, ή
  - στις παραμέτρους / τύπους επιστροφής των μεθόδων

14

Παράδειγμα



Όνομα: καθαρή περιγραφή μιας ξεκάθαρης οντότητας του κόσμου του προβλήματος

Στην πρώτη φάση, που σχεδιάζουμε κλάσεις αδρά, δεν υπάρχουν λεπτομέρειες για τις μεθόδους

Μεγάλη έμφαση σε ένα **μικρό** και **συνεκτικό σύνολο μεθόδων**

15

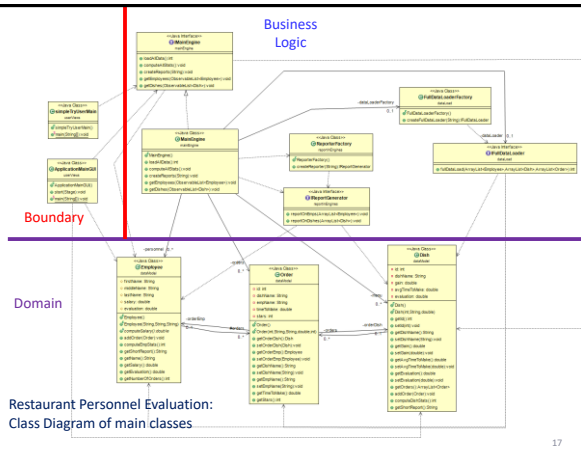
THIS ONE + THE NEXT ARE THE SINGLE TWO MOST IMPORTANT SLIDES OF THE ENTIRE COURSE!

3 types of classes

Γενικά
Πώς γίνεται η ανάλυση
Αποτύπωση σχ. & Οδηγίες
Εναλλακτικές μέθοδοι

- Once all the analysis and design is over, one can think of 3 types of classes
  - Domain classes: classes that represent the entities of the part of the real world under modeling which are also the core focus of the analysis phase
  - Boundary Classes & Interfaces: classes & interfaces that deal with the interplay with external actors (this includes the User Interface, but also all interaction with external actors)
  - Business-Logic classes & Interfaces: classes that incorporate logic (equiv.: implementation of the use cases + any other important computation) and are responsible for bridging boundary and domain classes
- Be careful: a complete specification of this list comes only after design has been completed! At the analysis phase, it is mostly domain classes that matter + early cuts of the rest

16



Restaurant Personnel Evaluation:
Class Diagram of main classes

17

How To: Classes, Relationships and Methods

- Use the requirements text to extract domain classes & their relationships.
  - Nouns are probably the best indicator for classes + their attributes
- Use the use cases to extract methods
  - Methods are directly related to the verbs of the use cases
  - You will probably (also depending on your design methodology) need a "MainEngine" class (only for starters) to host (a) data collections + (b) methods representing use-cases
  - Will probably need coarse-grained boundary classes for interactions with actors (end-users => U.I. and external subsystems)

18

Εύρεση κλάσεων από τη γραμματική

Ουσιαστικά και φράσεις που είναι στην ουσία ουσιαστικά	Κλάσεις και πεδία κλάσεων
Flight flight number	
Ρήματα και φράσεις που λειτουργούν γύρω από ρήματα	Αρμοδιότητες = δημόσιες μέθοδοι
verify verify flight number	

ΠΡΟΣΟΧΗ:
- Ομώνυμα και συνώνυμα
- Κρυμμένες κλάσεις: κλάσεις που δεν υπάρχουν στο κείμενο, αλλά αν προστεθούν, όλα κολλάνε (πολύ συχνά, οι use cases λένε «το σύστημα» κι αυτό κάνει τα πράματα ασαφή)

Πού βρίσκουμε το κείμενο:
- Στην καταγραφή των απαιτήσεων
- Στις uses cases

19

Sequence Diagrams for Use Cases

- Ideally: create a sequence diagram per use case ("use case realization" at the modeling level)
- Start with the domain classes (+ any other class) obtained by the parsing of the requirements text
- Arrows are the methods (see next for principles on method placement at classes)
- Add controllers / collection holders / … if needed
  - Can have a "manager" class play the maestro of the use case, or
  - Can pass the state of the use case from object to object
- If the diagram looks as if it cannot translate to code SIMPLY, restart!

- Remember this will NOT be successful in the first place, but it **will be iterated a few time & updated at design time**
- There is a time & effort cost with this process. **Use wisely.**

20

Use cases for methods

- Κάθε use case, αν έχει γραφεί σωστά, έχει προτάσεις της κατηγορίας

  `<id> The <actor / (sub)system> <performs function>`

- Πολύ συχνά το ρήμα της παραπάνω πρότασης είναι και μια μέθοδος (ενδεχομένως με παράμετρο το αντικείμενο της πρότασης). Το ζητούμενο είναι να αποφασίσουμε ποιο αντικείμενο κάνει τη ζητούμενη ενέργεια και να βάλουμε τη σχετική μέθοδο στην εν λόγω κλάση, ιδίως εκεί που έχουμε ως υποκείμενο "the system".

21

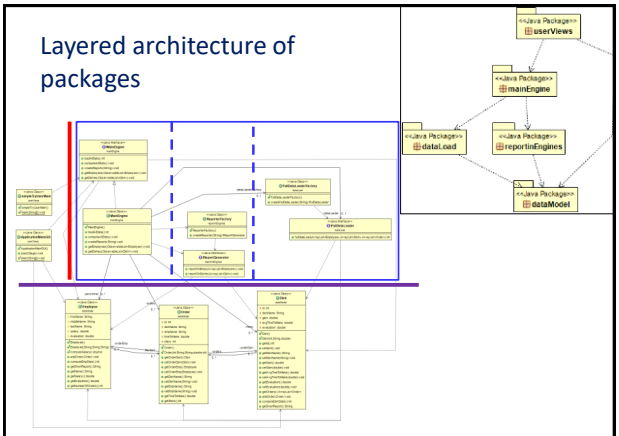
Where to assign methods?

- Where to assign methods?
  - PRINCIPLE OF LOCALITY: place methods as close as possible to their data (i.e., domain classes)
    - Equivalently: each object, should know+set its own status + compute properties that clearly pertain to it (e.g., a method getFinalPrice () that would best look sth like

      `getFinalPrice (){return amt+getTax()-getDiscount();}`

      should be placed as close as possible to these 3 attributes
  - FAÇADE PRINCIPLE: use cases (and major sub-use cases) pertain to business logic "Manager" (a.k.a "Controller") classes that export them to boundary classes
  - WRAPPER PRINCIPLE: for starters, use boundary classes for the user interface + external subsystems (later, move on to design Interfaces as contracts among your system and external actors)

Layered architecture of packages



Κριτήρια για καλές κλάσεις ανάλυσης

- Το όνομα πρέπει να είναι μια σύντομη και ξεκάθαρη περιγραφή της αντίστοιχης οντότητας του πραγματικού κόσμου
- Οι αρμοδιότητες = οι υπηρεσίες = οι δημόσιες μέθοδοι που προσφέρει μια κλάση στις άλλες κλάσεις πρέπει να είναι λίγες και συνεκτικές (βλ. παρακάτω)
- Οι συνεργαζόμενες κλάσεις μιας κλάσης δεν πρέπει να είναι πολλές

Θεμελιώδεις ιδιότητες κλάσεων

- Cohesion: ο βαθμός στον οποίο οι μέθοδοι μιας κλάσης εργάζονται για τον ίδιο σκοπό
- Coupling: ο βαθμός αλληλεξάρτησης με άλλες κλάσεις = ο αριθμός των κλάσεων προς τις οποίες η υπό αναφορά κλάση έχει συσχετίσεις

25

Οδηγίες

- Υπεραπλουστεύοντας, μια χονδρική εκτίμηση:
  - Συνήθως, 3-5 αρμοδιότητες (δλδ., μέθοδοι) ανά κλάση, οι οποίες συνεργάζονται στον ίδιο σκοπό είναι μια καλή ανάθεση
  - Καμιά κλάση δε στέκει μόνη της: για κάθε κλάση μικρός αριθμός συνεργαζόμενων κλάσεων
- Αποφύγετε τις ακραίες λύσεις
  - Extremity #1: God Classes = classes with a huge number of responsibilities (i.e., methods) -- esp., dangerous with façade, manager classes
  - Extremity #2: Oversimplification = huge number of classes with a very small (e..g, 1) number of responsibilities (i.e., methods)
- Αποφύγετε τα δέντρα κληρονομικότητας (και ιδίως τα βαθιά): γενικά, αποφύγετε / καθυστερήστε να εισάγετε ιεραρχίες όσο πιο πολύ γίνεται – κάντε το μόνο αν είναι απαραίτητο και ελέγξτε με το κριτήριο ISA

26

Other hints and tips

- Encapsulation principle: show/do only what's needed!
  - Suppress get() & set() to the absolutely minimum that is needed to get the job done. At the analysis phase you can even skip them.
  - Similarly, for constructors
- Data holding:
  - if you follow a top-down methodology, you can start with a MainEngine class as the holder of object collections.
  - If there is too much complexity in managing a particular collection, consider moving this functionality to a dedicated data holder class.
  - Too often, lookups to objects (e.g., TaxPayer by ΑΦΜ, car by plateNumber, …) will be required from the collection: try to minimize the tendency of overloading the design with all the details, esp., at the early phases
- The **patience** principle: way too often, it is necessary to check out all the use cases to decide the best allocation of collection management, object creation, and method placement.
  - **Sometimes, it is better to delay decisions & see the degree of "reuse" of object collaborations in different use cases to decide difficult cases**

27

Τελικές οδηγίες επί της διαδικασίας

- Προσθέστε όσες κλάσεις χρειάζεστε για αρχή, αρκεί να βγαίνει η υλοποίηση της use case.
- Μετά ελέγξτε αν η σχεδίαση που προέκυψε είναι προβληματική και επαναλάβετε τη διαδικασία μέχρι να συγκλίνει σε μια σχετικώς σταθερή δομή.
  - Ελέγχουμε αν έχουν διαφύγει όροι από το κείμενο των απαιτήσεων ή τα use case specifications
- Η παραπάνω διαδικασία είναι εξαιρετικά χρήσιμη καθώς μπορεί να καταλήξει
  - στην προσθήκη νέων κλάσεων ή την αλλαγή άλλων
  - στην αναθεώρηση / διόρθωση της use case
  - στην ανάγκη να επικοινωνήσουμε με τους χρήστες για να ξεκαθαριστούν οι αρχικές απαιτήσεις

Εναλλακτική μέθοδος εντοπισμού κλάσεων: η μέθοδος CRC

- Class Responsibilities Collaborations
- Η μέθοδος βασίζεται στο brainstorming γύρω από τις CRC Cards (β. δίπλα)
- Τα CRC cards ζωγραφίζονται είτε σε:
  - ασπροπίνακα (για σας: μεγάλο χαρτί)
  - σε μεγάλα sticky notes
  - συνδυασμό των παραπάνω

Class Name:	
Responsibilities	Collaborations

Φάση 1: brainstorming

- Εξηγούμε σε όλους ότι πρόκειται για brainstorming και
  - ΟΛΕΣ οι ιδέες είναι καλές
  - ΔΕΝ υπάρχει αντιπαράθεση επί των προτάσεων
- Εντοπίζουμε «πράγματα» στο χώρο του προβλήματος ως κλάσεις, με τις σχετικές αρμοδιότητες και γεμίζουμε το σχετικό sticky note
- Βρίσκουμε συνεργασίες σε κάθε νέα κλάση και
  - Είτε μετακινούμε τα sticky notes ώστε να είναι κοντά
  - Είτε ζωγραφίζουμε τις σχετικές γραμμές αν δουλεύουμε μόνο με πίνακα

Φάση 2: Ανάλυση

- Με βάση τα προαναφερθέντα κριτήρια, αποτιμούμε αν οι κλάσεις μας είναι «καλές κλάσεις»
- Πιθανώς κάποιο card που διαφαίνεται να είναι «μέρος» άλλου + να έχει λίγες αρμοδιότητες γίνεται πεδίο
  - If in doubt: keep it a class!

31

Εμπλουτισμός, εξειδίκευση και βελτίωση των αρχικών κλάσεων της ανάλυσης (ο οποίες αναπαριστούν τις βασικές οντότητες του προβλήματος) με κλάσεις που αναπαριστούν τα κομμάτια του λογισμικού που δίνουν λύση στο πρόβλημα

ΣΧΕΔΊΑΣΗ ΣΥΣΤΉΜΑΤΟΣ

32

Σχεδίαση

- Η ανάλυση είναι μια πρώτη, σχετικά χονδροειδής, απόπειρα να ορίσουμε τις κλάσεις του συστήματος σε σχέση με τη λειτουργικότητα που θέλουμε να υλοποιηθεί.
- Η σχεδίαση σκοπό έχει να εισάγει την απαραίτητη λεπτομέρεια στο μοντέλο μας – γι' αυτό και η έμφαση είναι πιο πολύ στα interfaces & susbsystems
- Η ανάλυση λέει τι κάνει το σύστημα και η σχεδίαση πώς το κάνει

33

From problem analysis to system design

- From a high level model that describes WHAT the system is expected to do (problem space as expressed at system req's) via an analysis model that includes
  - Analysis packages and classes
  - Sequence/activity diagrams
- We move to a detailed specification of HOW the system will be implemented (solution space) via a design model that includes
  - Subsystems and classes and interfaces @ design level
  - Use case realizations @ design level
  - Deployment diagrams

34

Σχεδίαση

- Σε μεγάλα έργα, με μεγάλα μοντέλα, η σχεδίαση μπορεί να είναι μια μεγάλη διαδικασία που να πολλαπλασιάσει το μέγεθος του μοντέλου σημαντικά
- Σε «έργα» του μεγέθους που αντιμετωπίζετε στις σπουδές σας (και στο παρόν μάθημα), η σχεδίαση πιο πολύ στόχο έχει να ξεκαθαρίσει το μοντέλο σας, να το συγκεκριμενοποιήσει και να φτιάξει τη λεγόμενη αρχιτεκτονική του συστήματος (system architecture)

35

Η σχεδίαση…

- … προσθέτει λεπτομέρειες υλοποίησης στις κλάσεις ανάλυσης, …
- … συχνά «σπάει» κλάσεις της ανάλυσης σε νέες, πιο συνεκτικές κλάσεις, …
- … προσθέτει κώδικα για τη διασύνδεση των κεντρικών κλάσεων με τα κομμάτια του front-end, back-end, network, … (middleware software and existing, trusted software libraries)

Η οργάνωση των κλάσεων όταν η πολυπλοκότητα είναι μεγάλη γίνεται με υποσυστήματα, και η κόλλα μεταξύ τους είναι interfaces (βλ. τις σχετικές ενότητες παρακάτω)

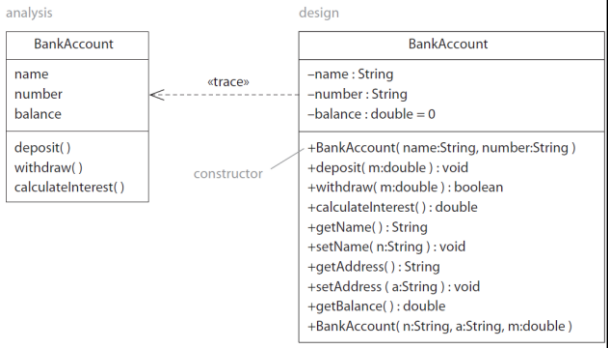
36

Κλάσεις της σχεδίασης

- Οι κλάσεις πλέον πρέπει να έχουν πλήρη στοιχεία για πεδία και μεθόδους
- Το δημόσιο κομμάτι (public interface) μιας κλάσης πλέον συνιστά ένα συμβόλαιο (contract) για το τι μπορεί να προσφέρει η κλάση στους clients της
  - Άρα ο developer της, εγγυάται στους άλλους developers πώς μπορούν να χρησιμοποιούν την κλάση του

37

Διαφορά κλάσης στην ανάλυση και τη σχεδίαση (από Arlow & Neustadt)



Ιδιότητες μιας καλής κλάσης σχεδίασης: a checklist

- The public interface of a class is **exactly** what its clients expect
- A method performs a simple, **atomic**, unique service (Every adjective counts)
- We have achieved **high cohesion** and **low coupling** for modules

39

Ιδιότητες μιας καλής κλάσης σχεδίασης

- Το συμβόλαιο = σύνολο δημόσιων μεθόδων της κλάσης να είναι σαφές, αποδεκτό από όλους και το ελάχιστο δυνατό (exactly what the class' clients expect)
  - ΠΡΟΣΟΧΗ: ποτέ δεν δίνουμε παραπάνω εγγυήσεις στο public interface μιας κλάσης απ' όσες χρειάζεται!!
  - Αλλιώς ελλοχεύει ο κίνδυνος της κακής χρήσης και της περιττής συντήρησης

40

Ιδιότητες μιας καλής κλάσης σχεδίασης

- Οι μέθοδοι είναι όσο το δυνατόν πιο απλές = κάθε μία υλοποιεί μια (1) δουλειά και μόνο (a simple, **atomic**, unique service)
  - Αποφύγετε τις μεθόδους που κάνουν πολλές δουλειές
  - Αποφύγετε τις πολλές εναλλακτικές υλοποιήσεις για την ίδια δουλειά
  - Ότι περιττό προστίθεται, μπορεί να χρησιμοποιηθεί κακώς και σίγουρα επιφέρει ανεπιθύμητη συντήρηση

41

Ιδιότητες μιας καλής κλάσης σχεδίασης

- Υψηλή **συνεκτικότητα** (cohesion)
  - Όλα τα στοιχεία της κλάσης συνεργάζονται για την επίτευξη κατά βάση μίας δουλειάς …
  - … συνήθως αυτό προκύπτει κρατώντας χαμηλά των αριθμό των αρμοδιοτήτων και των πεδίων
- Χαμηλός βαθμός συσχέτισης (coupling)
  - Η συσχέτιση με άλλες κλάσεις πλην των απαραιτήτων απαγορεύεται δια ροπάλου
  - Όσο πιο πολλές συσχετίσεις, τόσο πιο επώδυνη η συντήρηση

42

Παν μέτρον άριστον

- Έχοντας πει τα προηγούμενα …
  - Δε σημαίνει ότι θα καταλήξουμε να έχουμε συμβόλαια με μία μέθοδο μόνο, μόνο και μόνο για να μη θεωρηθεί ότι πλατειάζουμε…
  - Δε σημαίνει ότι αν χρειαστεί να συσχετίσουμε δύο κλάσεις (είτε λόγω μιας σχέσης part-whole είτε λόγω μιας συνεργασίας) δε θα το κάνουμε για να μην υπάρχει coupling…

43

Use case realizations at design level

- Εκτός από κλάσεις, προφανώς μπορούν να αναθεωρηθούν / επεκταθούν / … και άλλα design artifacts της διαδικασίας ανάλυσης:
  - Add implementation details to the fundamental, analysis-level interaction diagrams
  - Zoom-out at system level and model interactions between subsystems (rather than between objects) via interfaces as glue => produce the **system architecture** (**see next!**)

44

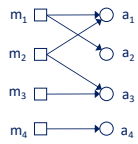
Πώς μετράμε την εσωτερική συνεκτικότητα μιας κλάσης, και για την ακρίβεια, πώς αποτιμούμε το αν οι μέθοδοι μιας κλάσης εξυπηρετούν ακριβώς μία λειτουργικότητα

COHESION METRICS

Cohesion

- **Cohesion** is a measure of the extent to which the methods and the structure of a class are oriented towards serving a single purpose
  - A cohesive module has all its methods and attributes oriented towards serving the same functionality
  - Hard to determine it in an automated, measurable way
  - We approximate the "relatedness" of the classes methods via the patterns of how methods access attributes
  - We typically assess **Lack of** Cohesion Of Methods (LCOM)
    - Chidamber S. R. and Kemerer C. F. A Metrics Suite for Object Oriented Design. IEEE Transactions on Software Engineering, vol. 20, no. 6, pp. 476 - 493, June 1994.
    - Henderson-Sellers, B. Object-oriented metrics : measures of complexity. Prentice-Hall, 1996.
- A high Lack of Cohesion value indicates that possibly the class serves more than one roles and needs to be split
- Huge discussion of the research community over cohesion at both the semantic and the metrics level (out of scope of this course)

Assume the following class



	m1	m2	m3	m4
m1				
m2				
m3				
m4				

When we discuss cohesion and LCOM*, we typically do it by observing **patterns on how methods access attributes**.

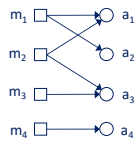
We use a bipartite graph, where
- Square nodes are methods
- Circular nodes are attributes
- An edge (m,a) exists if a method uses an attribute

Here, we have an example of a class with 4 methods and 4 attributes

Frequently (but not always), we assess a quantity over all **pairs of methods** (lower triangular part of the matrix)

Whether to exclude getters and setters from the graph is open to debate!

LCOM1



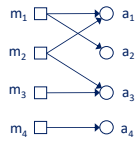
	m1	m2	m3	m4
m1				
m2	X			
m3	!	X		
m4	!	!	!	

Lack of Cohesion of Metrics #1

Chidamber & Kemerer, 1994

- Number of pairs of methods without common attribute references
- Ideally, a super cohesive module has LCOM1 = 0 && a super non-cohesive, the #cells of the lower triangle.
- Here, out of 6 pairs, there are LCOM1 = 4 w/o common accesses

LCOM2



	m1	m2	m3	m4
m1				
m2	X			
m3	!	X		
m4	!	!	!	

Lack of Cohesion of Metrics #2

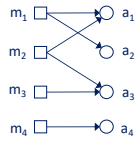
Chidamber & Kemerer, 1994

- Split the pairs-of-methods in two sets
  - P: pairs that do NOT share attributes
  - Q: pairs that share attributes

$$LCOM2 = \begin{cases} P - Q, \text{ if } P - Q \geq 0 \\ 0, \text{ otherwise} \end{cases}$$

- Ideally, a cohesive module has LCOM2 = 0 && a super non-cohesive, LCOM2=P, the #cells of the lower triangle (Q=0).
- Here, LCOM2 = 4-2 = 2

LCOM5



	m1	m2	m3	m4
a1	X	X		
a2	X			
a3		X	X	
a4				X

No more pairs of methods here!

Lack of Cohesion of Metrics #5

Henderson-Sellers, 1996

- Use the attributes, instead!!
- Assuming m methods, α attributes, and *μ(A)* the methods accessing A

$$LCOM5 = \frac{\frac{1}{a}\sum_{j=1}^{a}\mu(A_j) - m}{1 - m}$$

- LCOM5 = 0 whenever all methods access all attributes – super cohesive
- LCOM5 = 1, super non-coh., whenever every method accesses a single attribute
- {¼(2+1+2+1) – 4}/(-3) = (-10/4)/-3 = 5/6

Also known as "Analysis Packages"

A nice way to structure system parts;

As close as possible to subsystem design @ analysis phase

PACKAGES

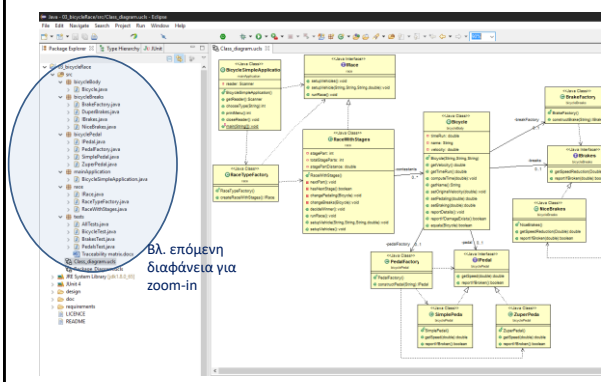
Packages

- Τα πακέτα κώδικα (packages) παρέχουν ένα τρόπο να ομαδοποιούμε «κατασκευές» (κώδικα, διαγράμματα, …) που έχουν μεγάλη λογική συνάφεια
- Ένα πακέτο σε επίπεδο ανάλυσης συνήθως περιέχει:
  - Use cases
  - Analysis classes & class diagrams
  - Use case realizations (typically: sequence diagrams)

Packages

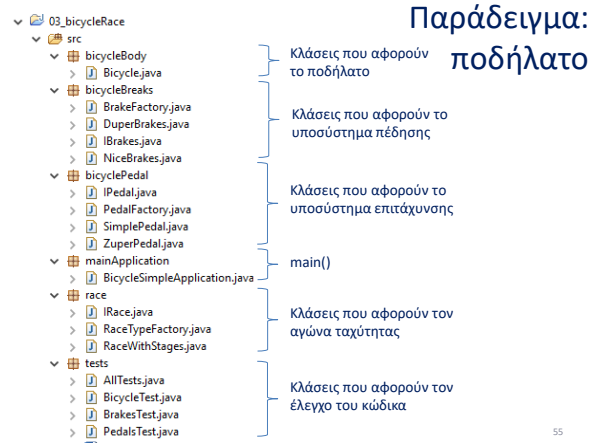
- Τα πακέτα κώδικα (packages) είναι ένας μηχανισμός που μας επιτρέπει να ομαδοποιούμε «κατασκευές» (κώδικα, διαγράμματα, …) που έχουν μεγάλη λογική συνάφεια
- Έτσι μπορούμε να δομούμε ένα μεγάλο μοντέλο σε επί μέρους τμήματα, ή επί μέρους μοντέλα, με το καθένα να έχει μια σχετική αυτοτέλεια και να θέτει **ένα εσωτερικό «σύνορο»** εντός του ευρύτερου υποσυστήματος

Παράδειγμα: ποδήλατο



Βλ. επόμενη διαφάνεια για zoom-in

Παράδειγμα: ποδήλατο

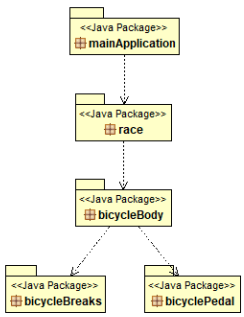


Μερικές ιδιότητες

- Κάθε κατασκευαστικό στοιχείο, όπως και κάθε μοντέλο, ανήκει σε ακριβώς ένα πακέτο
- Κάθε πακέτο ορίζει και ένα ενιαίο + διακριτό χώρο ονομάτων (**namespace**)
- Έχουμε **ορατότητα** (visibility) στα στοιχεία ενός πακέτου (π.χ., σκεφτείτε το με κλάσεις)
  - Public elements (+): visible to other packages
  - Private elements (-): invisible to other packages

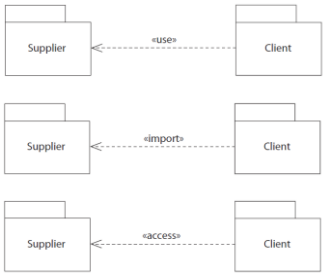
Design guideline: κάντε δημόσιο μόνο ότι χρειάζεται και τίποτε άλλο

Package Diagram



- **Icon**: a folder icon
- **Name**: meaningful characterization of the subsystem
- Dependency: όταν ένα στοιχείο ενός "client" package χρησιμοποιεί ένα στοιχείο ενός "supplier" package

Package dependencies (από Arlow & Neustadt)



«use»: An element in the client package **uses** a public element in the supplier package in some way – the client depends on the supplier
If a package dependency is shown without a stereotype, then «use» should be assumed

«**import**»: Public elements of the supplier namespace are added as **public** elements to the client namespace

«**access**»: Public elements of the supplier namespace are added as **private** elements to the client namespace

The «use» dependency means that there are dependencies between elements in the packages, rather than between the packages themselves.
Both «import» and «access» merge client and supplier namespaces => client elements to use unqualified names to access supplier elements. Differences between the two:
- «import»: merged supplier elements become public in the client; import is transitive
- «access»: merged supplier elements become private in the client; access is not transitive

58

Nesting and Inheritance

- **Εμφώλευση**: ένα πακέτο μπορεί να περιέχει ένα άλλο πακέτο
  - Η αναδρομή μπορεί να συνεχίζεται θεωρητικά σε όσο βάθος θέλουμε – πρακτικά, όχι παραπάνω από 2 -3 επίπεδα
  - Ένα εμφωλευμένο πακέτο «βλέπει» όλα τα δημόσια στοιχεία του περικλείοντος πακέτου, αλλά όχι το αντίστροφο, εκτός κι αν υπάρχει ρητά σχέση εξαγωγής
- **Κληρονομικότητα**: ένα πακέτο μπορεί να επεκτείνει ένα άλλο
  - **Αποφύγετέ την, χάριν απλότητας!**

59

Αρχιτεκτονική Ανάλυση (!!)

- Ο στόχος της αρχιτεκτονικής ανάλυσης είναι **να οργανώσει τις κλάσεις ανάλυσης σε cohesive πακέτα** …
- … τα οποία με τη σειρά τους, τα οργανώνει σε επίπεδα
- **Βασικός στόχος**: ελάχιστη δυνατή εξάρτηση μεταξύ πακέτων (**minimize package coupling**)

60

Minimize package coupling

- Μια από τις πιο βασικές αρχές στη σχεδίαση πακέτων
- Όσο λιγότερη αλληλεξάρτηση μεταξύ πακέτων, τόση λιγότερη συντήρηση!
- Πώς επιτυγχάνεται:
  - **Minimize dependencies between analysis packages!**
  - **Minimize public parts of packages** (in an attempt to minimize unnecessary usage)

61

Πώς βρίσκουμε πακέτα ανάλυσης

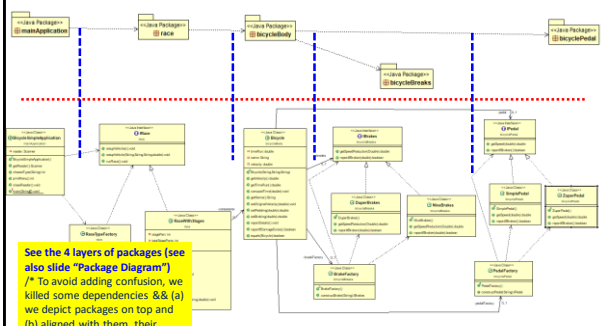
- Ένα πακέτο ανάλυσης πρέπει να περιλαμβάνει ένα σύνολο κατασκευών (ιδίως κλάσεων) που να έχουν στενές εννοιολογικές συσχετίσεις μεταξύ τους
- Οι κλάσεις σχετίζονται μεταξύ τους με την εξής φθίνουσα σειρά στη ένταση αλληλεξάρτησης:
  - Inheritance hierarchies
  - Composition (strong form of aggregation)

  } Practically, a must
  - Aggregation
  - Dependencies

  } Any other cohesive cluster of classes is typically indicated by these two

62

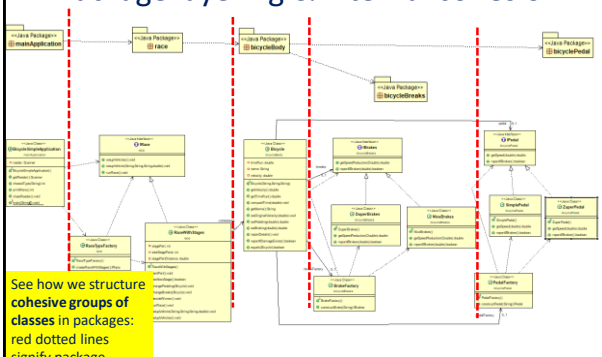
Package layering & internal cohesion



See the 4 layers of packages (see also slide "Package Diagram")
/* To avoid adding confusion, we killed some dependencies && (a) we depict packages on top and (b) aligned with them, their classes. Red dotted line visually separates classes from packages */

63

Package layering & internal cohesion



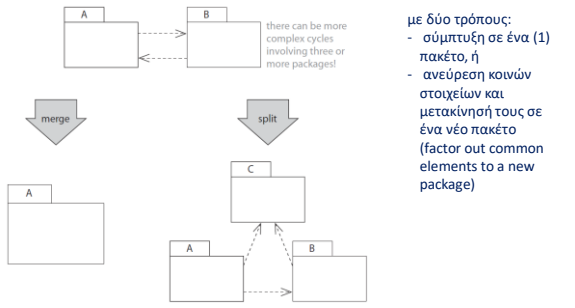
See how we structure **cohesive groups of classes** in packages: red dotted lines signify package borders

Πώς βρίσκουμε πακέτα ανάλυσης

- **Keep it simple**: avoid generalization (practically always) and nesting (at least at first)
- **Keep coupling low and cohesion within packages high!!!**
- Avoid use cases as a hint for packages: typically, use cases span across many packages!
- Avoid cyclic dependencies (IMHO: at all costs)

- When done: reconsider (possibly there is still room for merging/splitting packages, moving classes, …)

Πώς σπάμε κυκλικές εξαρτήσεις



με δύο τρόπους:
- σύμπτυξη σε ένα (1) πακέτο, ή
- ανεύρεση κοινών στοιχείων και μετακίνησή τους σε ένα νέο πακέτο (factor out common elements to a new package)

(σχήματα από Arlow & Neustadt)

Analysis packages: design guidelines and a checklist of good design

- Minimize dependencies between analysis packages
- Minimize publicity per package (publish exactly the elements that are used by client packages)
- Maximize cohesion within packages
- Avoid cycles in the dependencies between packages!!

http://en.wikipedia.org/wiki/Diaeresis_(philosophy)

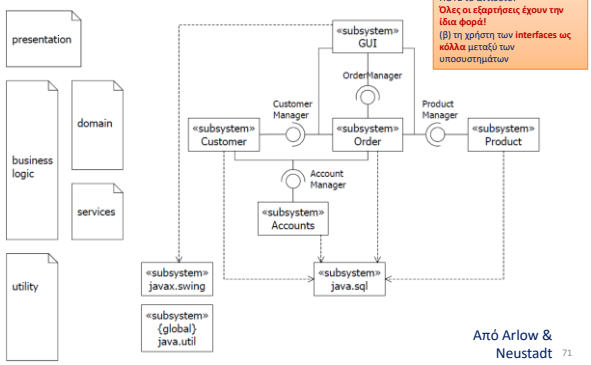
ΥΠΟΣΥΣΤΉΜΑΤΑ

Υποσυστήματα

- Ένα υποσύστημα είναι μια «λογικού επιπέδου» σχεδιαστική κατασκευή (δλδ., σε επίπεδο σχεδίασης και όχι εκτελέσιμου κώδικα) που δρα ως ένα λογικά **ενιαίο συστατικό ενός σύνθετου συστήματος**
  - Με άλλα λόγια, μπορούμε να σκεφτόμαστε ένα υποσύστημα, που στην πράξη περιέχει interfaces και κλάσεις που τα υλοποιούν από πλευράς κώδικα, ως ένα, ατομικό, ενιαίο σχεδιαστικό μπλοκ που παρέχει μια διακριτή λειτουργικότητα, που μπορεί να θεωρηθεί «ανεξάρτητη» από τις άλλες λειτουργίες του συστήματος
  - Τα υποσυστήματα αναπαριστώνται γραφικά ως stereotyped «**subsystem**»

Υποσυστήματα

- Από την πλευρά της όλης σχεδιαστικής διαδικασίας, τα υποσυστήματα έχουν κεντρικό ρόλο στη διαδικασία σχεδίασης μεγάλων συστημάτων, στο βαθμό που επιτρέπουν να σπάσουμε ένα δύσκολο πρόβλημα ανάπτυξης σε μικρότερα υποπροβλήματα (που μπορούν να ανατεθούν εν παραλλήλω σε διαφορετικές ομάδες ανάπτυξης).
- Οι λεπτομέρειες υλοποίησης των υποσυστημάτων κρύβονται πίσω από interfaces – τα υποσυστήματα επικοινωνούν μεταξύ τους με **interfaces** που λειτουργούν ως «**συμβόλαια**»



Από Arlow & Neustadt

Revisit slides of **Unit 4 on UML!**

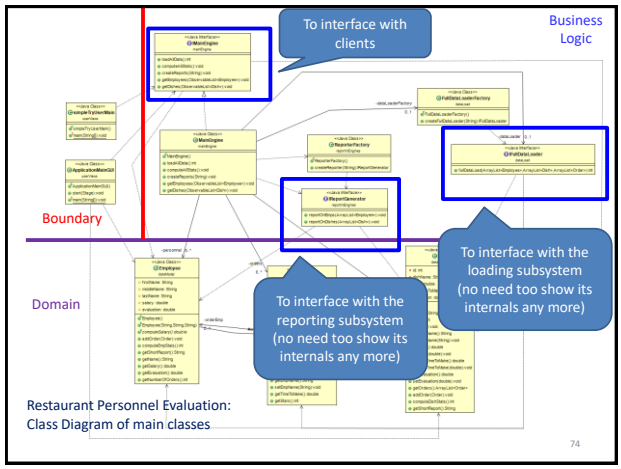
Interfaces are an extremely useful mechanism for maintainable code; the mastery of interfaces is a must for a competent developer

INTERFACES

Interfaces: what-it-is & an example

- Πρακτικά, ένα interface ορίζει ένα σύνολο δημόσιων αφηρημένων μεθόδων τις οποίες, οι κλάσεις που υλοποιούν το interface υποχρεούνται να υλοποιήσουν και να παρέχουν
  - μπορείτε να το σκέφτεστε ως συμβόλαιο, με τον interpreter/compiler στο ρόλο του δικαστή/συμβολαιογράφου που εντοπίζει παραβιάσεις του συμβολαίου
  - Έτσι βοηθά στο να διαχωρίζουμε υποσυστήματα, και να τα συνδέουμε μέσω «συμβολαίων» λειτουργικότητας
  - Επίσης, αν οι clients ενός υποσυστήματος ξέρουν μόνο το interface και όχι την concrete class, είναι πολύ εύκολο να συντηρηθεί το εσωτερικό του υποσυστήματος χωρίις να το μάθουν οι clients

73



Restaurant Personnel Evaluation:
Class Diagram of main classes

74

When to add an interface to the design?

- **Whenever you have multiple classes that are eligible/required to play the same role. E.g.,**
  - … multiple algorithms for the same task
  - … you have an association where evolution can occur at one end (esp., if this is going to be handled in a polymorphic way)
- For each set of operations which is repeatedly reused/reusable, esp., if implemented at different suppliers
- **Whenever you want to define boundaries between subsystems!**

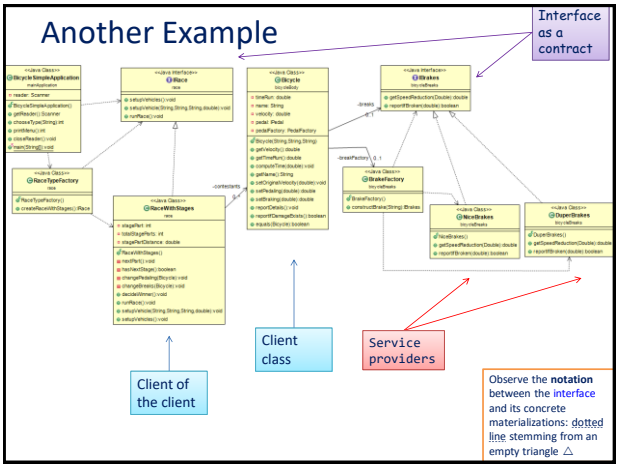
75

Προσέξτε πώς τα interfaces είναι …

- … οι «ραφές» (seams) μεταξύ των υποσυστημάτων = αντιπρόσωποι των υποσυστημάτων, σε σχέση με το τι «υπόσχεται» το υποσύστημα ότι μπορεί να κάνει
- … οι απομονωτές των αλλαγών που θα γίνουν εσωτερικά στο υποσύστημα/package και δε θα επηρεάσουν κανέναν άλλο (αφού κανείς εκτός package δεν ξέρει τι έχει μέσα)
- Εισάγουμε interfaces, λοιπόν
  - Όταν έχουμε αποφασίσει τα υποσυστήματα, ώστε να λειτουργήσουν ως δημόσιοι αντιπρόσωποι των υποσυστημάτων
  - Σε όσα σημεία εκτιμούμε ότι θα έχουμε συντήρηση, ώστε να απομονώσουμε τους clients από τις concrete classes που θα συντηρηθούν και να μη διαδοθεί η επίδραση των αλλαγών περεταίρω.

76

Another Example



Μια γρήγορη ματιά από ένα εξαιρετικά μεγάλο θέμα,
η οποία αφορά το πώς αντιμετωπίζουμε εμείς το διαχωρισμό υποσυστημάτων

ΑΡΧΙΤΕΚΤΟΝΙΚΉ ΑΝΤΙΚΕΙΜΕΝΟΣΤΡΕΦΟΎΣ ΛΟΓΙΣΜΙΚΟΎ

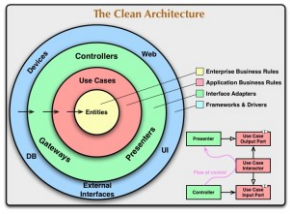
78

Βασικές ερωτήσεις

- Πώς να πρέπει να οργανώσουμε τον κώδικα?
- Υπάρχουν βασικές ιδέες γύρω από την οργάνωση του κώδικα?

Shamelessly "borrowed" figures from their original sites – links show where the figures comes from (many thanks to the authors)

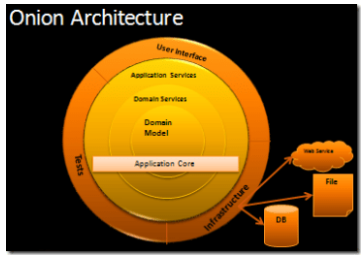
Clean Architecture



https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

- Robert C. Martin's idea of structuring code
- The concentric circles signify different types of classes and interfaces
- ALL DEPENDIENCIES POINT INWARDS!!!
- *"Nothing in an inner circle can know anything at all about something in an outer circle. "*

Onion Architecture

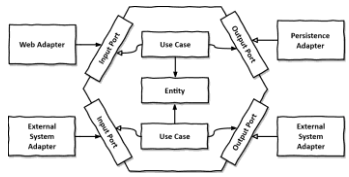


https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/

- Jeffrey Palermo's suggestion, partly inspiring Uncle Bob's clean architecture
- *"The object model is in the center with supporting business logic around it. The direction of coupling is toward the center. … any outer layer can directly call any inner layer. "*

Hexagonal Architecture

- All ports are Interfaces = contracts of behaviors (the arrow with the triangle should be dotted = implementation not ISA)
- All Adapters refer to the adapter design pattern: left hand side adapters customize our system to its clients (e.g., a Web of a GUI front-end) via input ports; right hand side, they hide service providers that we use to get our job done (e.g., a database, http page retriever, …) behind an interface = output port
- Internally, Use Cases correspond to our Business Logic classes (in Homberg's idea, it's more like micro-service structure, one class per use case) and Entities to our domain classes



https://reflectoring.io/book/

https://alistair.cockburn.us/hexagonal-architecture/

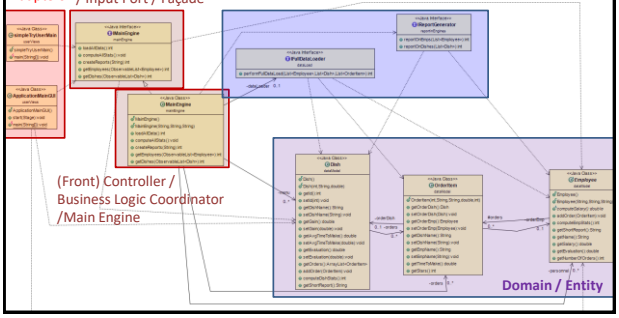
- Alistair Cockburn's original idea that is probably the foundation of all this
- Highly recommended book by Tom Hombergs at https://reflectoring.io/book/

Restaurant Evaluation v02

Boundary / Input Adapters

Service Contract / Input Port / Façade

Business Logic Subsystems / Services / Dedicated Engines / hexagonal "Use Cases" (with their internal "port" = interface as a contract + implementations not shown here)



(Front) Controller / Business Logic Coordinator /Main Engine

Domain / Entity

Restaurant Evaluation v02

In this example, we use **a single entrance port**

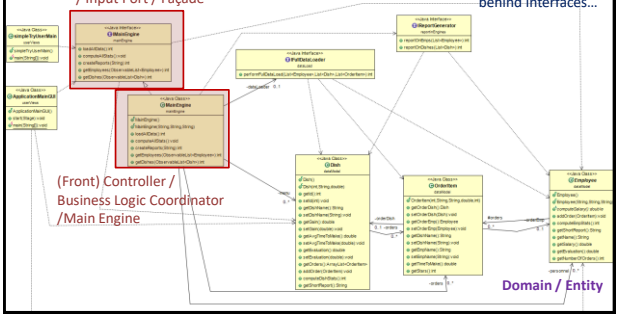
… this is a "door" for front-ends + in general clients of our system…

Service Contract / Input Port / Façade

… and acts as a façade: shows all the functionality of the system, in a single contract.

A **single "central" Main Engine** implementing the contract, receives incoming requests…

… and delegates parts of their servicing to internal services = subsystems, hidden behind interfaces…



(Front) Controller / Business Logic Coordinator /Main Engine

Domain / Entity

Restaurant Evaluation v02

Could it be more "distributed" as an architecture? Yes!

We could have many ports and services behind them!

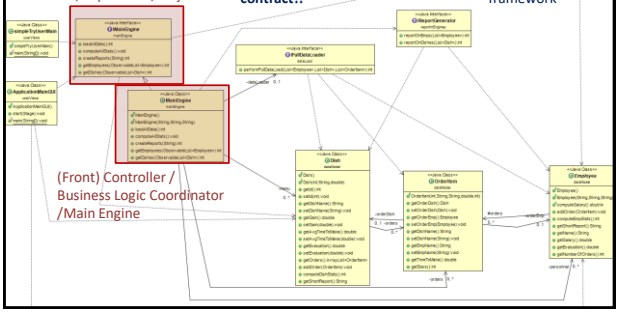
But you need to keep the main idea: **hide the (sub)system behind the contract!!**

What is **absent from the diagram? Object creation**, i.e., the object creation factories!!

In frameworks like e.g. Spring Boot, you can have automatic object creation via the framework

Service Contract / Input Port / Façade

(Front) Controller / Business Logic Coordinator /Main Engine

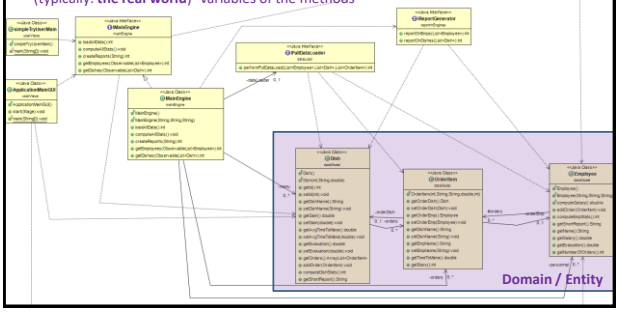


Restaurant Evaluation v02

Domain Classes = Entities are the classes creating objects that represent the actual entities of the domain (typically: **the real world**)

Innermost circle of the onion => Everyone else knows them and uses them as input parameters / return types / internal variables of the methods

All dependencies point towards domain classes; no outgoing dependencies from them to others



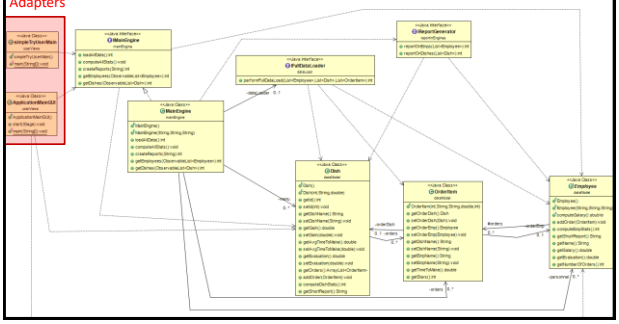
Restaurant Evaluation v02

Boundary / Input Adapters

Anyone using our system is a client.
Here: a text and a GUI client

They know ONLY:
- Input port contract
- Its factory (not shown here)
- Entities

They DO NOT:
- Include business logic, only "view" operations
- Act as target of a dependency (nobody depends upon them)



Restaurant Evaluation v02

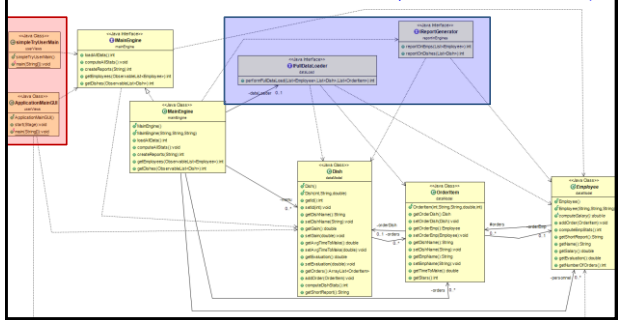
Internal subsystems handling all the business logic!

If a controller exists, typically it delegates work to them.

Typically, a package per service.

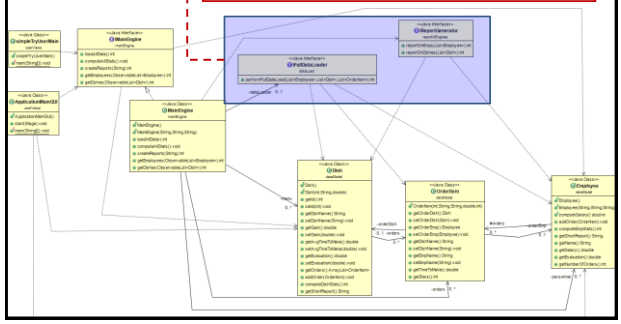
Aka "Services". Could come with their own input port, too

Business Logic Subsystems / Services / Dedicated Engines / hexagonal "Use Cases" (with their internal "port" = interface as a contract + implementations not shown here)



Restaurant Evaluation v02

Missing here: output ports and adapters! The Business Logic Service implementations would use them (i.e., depend upon them) to receive services from external systems -- e.g., sensor data, database support, http transfer services, ...



Summary: types of cl and interfaces

- **Service Contracts** = **Input interfaces** = **input ports**: contracts of what kind of services our system offers. Hide the entire subsystem behind them
- **Services** = **business logic classes** (frequently, behind a single **controller**). Implement the business logic of the (sub)system. **All the use cases land there as methods!!!** You need a **factory** to new() them.
- **Domain Classes** = Entities = classes with the objects of the domain to be used for carrying out the services
- **Output Ports and Adapters**: Boundary Interfaces and their implementation classes to hide external systems that provide services for us. You also need a **factory** to new() them.
- **System Clients** = **Input Adapters** = **Boundary classes** that use our (sub)system's services – e.g., to allow humans to interact with it, via a front-end interface

ΣΗΜΕΙΩΣΕΙΣ

Ακολουθεί ένα παράδειγμα για το πώς

- δοθείσης μιας περιγραφής ενός προβλήματος που έχουμε καταγράψει σε φυσική γλώσσα,
- αρχικά εξάγονται κάποια use cases, και στη συνέχεια,
- εξάγουμε τις κλάσεις ανάλυσης και κάποια sequence diagrams.

Εβδομαδιαία Αξιολόγηση στο Εστιατόριο του Μαστρο-Έκτορα

Κάθε βδομάδα, ο μαστρο-Έκτορας, ιδιοκτήτης εστιατορίου, αξιολογεί τους σεφ και την επίδοση του εστιατορίου. Στη διάρκεια της εβδομάδας, για κάθε πιάτο μιας παραγγελίας, καταγράφεται ποιος μάγειρας το ανέλαβε, σε πόση ώρα το ολοκλήρωσε (σε λεπτά) και τι αξιολόγηση πήρε από τον πελάτη (κλίμακα: 1 – 5 αστεράκια). Έτσι, στο τέλος της εβδομάδας, υπάρχει ένα αρχείο που περιγράφει αυτές τις πληροφορίες. (*Εμείς έχουμε το αρχείο δοθέν*). Για να κάνει την αξιολόγηση ο μαστρο-Έκτορας, θέλει να φορτώνει στο σύστημα που θα φτιάξουμε τρία αρχεία:

- Ένα αρχείο με την πληροφορία για κάθε μάγειρα (όνομα, επίθετο, ρόλος), με το ρόλο να είναι: chef de cuisine, sous chef, chef
- Ένα αρχείο με τα πιάτα που προσφέρει το εστιατόριο (όνομα, κόστος, τιμή)
- Ένα αρχείο με τη διεκπεραίωση του κάθε πιάτου μιας παραγγελίας, όπως προαναφέρθηκε. Επιπλέον των παραπάνω στοιχείων, στο αρχείο με τις παραγγελίες, κάθε εγγραφή έχει κι ένα μοναδικό αύξοντα κωδικό για να ξεχωρίζουν οι παραγγελίες μεταξύ τους.

Κάθε ένα από τα αρχεία αυτά έχει το εξής format: κάθε γραμμή είναι μια εγγραφή και τα στοιχεία της εγγραφής χωρίζονται με ένα διαχωριστικό χαρακτήρα (delimiter – typically a \t or |).

Το σύστημά μας, μόλις φορτωθούν τα αρχεία, *πρέπει να αναπαριστά τα στοιχεία τους ως αντικείμενα για να μπορούμε να τα επεξεργαστούμε. Η φόρτωση γίνεται ως εξής: για κάθε γραμμή ενός αρχείου, διαχωρίζονται τα στοιχεία του και με βάση αυτά φτιάχνεται και ένα αντικείμενο της κατάλληλης κλάσης που καταγράφεται από το σύστημά μας. Αφού όλα τα στοιχεία αναπαρασταθούν ως αντικείμενα,* το σύστημα πρέπει να αξιολογήσει πιάτα και σεφ. Για τα πιάτα, για κάθε πιάτο υπολογίζεται ο μέσος όρος αστεριών που πήρε και ο μέσος χρόνος κατασκευής του. Για κάθε σεφ, υπολογίζεται ο μέσος όρος αστεριών που πήρε και υπολογίζεται ο μισθός ως εξής:

- chef de cuisine: fixed amount + 2% * total profit (price – cost) if total average of stars for the entire set of orders > 4
- sous chef : fixed amount + average #stars * 100
- chef : average #stars * 400

Μετά, το σύστημα πρέπει να μπορεί να φτιάξει τις εξής αναφορές:

- Αναφορά με όλα τα στοιχεία του προσωπικού (συμπ. αξιολόγησης και μισθού)
- Αναφορά με όλα τα στοιχεία του πιάτων (συμπ. αξιολόγησης, χρόνου και κέρδους)

Οι αναφορές πρέπει να μπορούν να γραφτούν σε αρχείο κειμένου και html. Το σύστημα ως διαπροσωπεία έχει μόνο διαγνωστικά μηνύματα στο τέλος της κάθε φάσης, καθώς και πού γράφονται τα αρχεία των αναφορών.

Ο μαστρο Έκτορας, αρκεί να εκκινήσει την εφαρμογή της αξιολόγησης, και το σύστημα φορτώνει τα αρχεία, γίνονται οι αξιολογήσεις και παράγονται οι αναφορές.

Απαιτείται υποχρεωτικά η κατασκευή μιας κεντρικής κλάσης που διεκπεραιώνει όλα τα use cases και την επικοινωνία με τη διαπροσωπεία.

Επεκτάσεις: Διαδραστικό σύστημα επιλογής αναφοράς και αρχείων για φόρτωμα. Γραφική διαπροσωπεία για τα reports. Διαδραστική διαπροσωπεία για την συμπλήρωση των αρχείων στη διάρκεια της εβδομάδας.

Εβδομαδιαία Αξιολόγηση στο Εστιατόριο του Μαστρο-Έκτορα

Κάθε βδομάδα, ο μαστρο-Έκτορας, ιδιοκτήτης εστιατορίου, αξιολογεί τους σεφ και την επίδοση του εστιατορίου. Στη διάρκεια της εβδομάδας, για κάθε πιάτο μιας

παραγγελίας, καταγράφεται ποιος μάγειρας το ανέλαβε, σε πόση ώρα το ολοκλήρωσε (σε λεπτά) και τι αξιολόγηση πήρε από τον πελάτη (κλίμακα: 1 – 5 αστεράκια).

Έτσι, στο τέλος της εβδομάδας, υπάρχει ένα αρχείο που περιγράφει αυτές τις πληροφορίες. (*Εμείς έχουμε το αρχείο δοθέν*). Για να κάνει την αξιολόγηση ο μαστρο-Έκτορας, θέλει να φορτώνει στο σύστημα που θα φτιάξουμε τρία αρχεία:

- Ένα αρχείο με την πληροφορία για κάθε **μάγειρα** (όνομα, επίθετο, ρόλος), με το ρόλο να είναι: chef de cuisine, sous chef, chef
- Ένα αρχείο με τα **πιάτα** που προσφέρει το εστιατόριο (όνομα, κόστος, τιμή)
- Ένα αρχείο με τη διεκπεραίωση του ~~κάθε πιάτου μιας~~ **παραγγελίας**, όπως προαναφέρθηκε. Επιπλέον των παραπάνω στοιχείων, στο αρχείο με τις παραγγελίες, κάθε εγγραφή έχει κι ένα μοναδικό αύξοντα κωδικό για να ξεχωρίζουν οι παραγγελίες μεταξύ τους.

Κάθε ένα από τα αρχεία αυτά έχει το εξής format: κάθε γραμμή είναι μια εγγραφή και τα στοιχεία της εγγραφής χωρίζονται με ένα διαχωριστικό χαρακτήρα (delimiter – typically a \t or |).

Το σύστημά μας,

1. μόλις φορτωθούν τα αρχεία,
2. *πρέπει να αναπαριστά τα στοιχεία τους ως αντικείμενα*
3. *για να μπορούμε να τα επεξεργαστούμε.*

*Η **φόρτωση** γίνεται ως εξής:*

για κάθε γραμμή ενός αρχείου,

διαχωρίζονται τα στοιχεία του και

*με βάση αυτά **τα στοιχεία** φτιάχνεται και ένα αντικείμενο της κατάλληλης κλάσης που καταγράφεται από το σύστημά μας.*

Αφού όλα τα στοιχεία αναπαρασταθούν ως αντικείμενα, το σύστημα πρέπει να αξιολογήσει πιάτα και σεφ.

1. Για **να αξιολογηθούν** τα πιάτα:
    a. για κάθε πιάτο υπολογίζεται
        i. ο μέσος όρος αστεριών που πήρε και
        ii. ο μέσος χρόνος κατασκευής του.
2. Για **να αξιολογήσει** κάθε σεφ,
    a. υπολογίζεται ο μέσος όρος αστεριών που πήρε και
    b. υπολογίζεται ο μισθός ως εξής:

        - chef de cuisine: fixed amount + 2% * total profit (price – cost) if total average of stars for the entire set of orders > 4
        - sous chef : fixed amount + average #stars * 100
        - chef : average #stars * 400

Μετά, το σύστημα πρέπει να μπορεί να φτιάξει τις εξής αναφορές:

- Αναφορά με όλα τα στοιχεία του προσωπικού (συμπ. αξιολόγησης και μισθού)
- Αναφορά με όλα τα στοιχεία του πιάτων (συμπ. αξιολόγησης, χρόνου και κέρδους)

Οι αναφορές πρέπει να μπορούν να γραφτούν σε αρχείο κειμένου και html. Το σύστημα ως διαπροσωπεία έχει μόνο διαγνωστικά μηνύματα στο τέλος της κάθε φάσης, καθώς και πού γράφονται τα αρχεία των αναφορών.

Ο μαστρο Έκτορας, αρκεί να εκκινήσει την εφαρμογή της αξιολόγησης, και το σύστημα φορτώνει τα αρχεία, γίνονται οι αξιολογήσεις και παράγονται οι αναφορές.

Απαιτείται υποχρεωτικά η κατασκευή μιας κεντρικής κλάσης που διεκπεραιώνει όλα τα use cases και την επικοινωνία με τη διαπροσωπεία.

Επεκτάσεις: Διαδραστικό σύστημα επιλογής αναφοράς και αρχείων για φόρτωμα. Γραφική διαπροσωπεία για τα reports. Διαδραστική διαπροσωπεία για την συμπλήρωση των αρχείων στη διάρκεια της εβδομάδας.

USE CASES

EX.6

1. ΑξιολογησεΕπιδοσηΕστιατοριου

Description and Goal

Η use case «ΑξιολόγησεΕπίδοσηΕστιατορίου» υλοποιεί τη βασική λειτουργία του συστήματος. Δοθέντων των αρχείων εισόδου από την λειτουργία του εστιατορίου, παράγει τις αναφορές αξιολόγησης για πιάτα και προσωπικό.

Σημειογραφία:

Actors: μπλε

Σχόλια: πράσινα italics

Υποψήφιες Μέθοδοι: ρήμα + context e.g., computeChefSalary()

Actors (esp. primary actor)

Ιδιοκτήτης Εστιατορίου (βασικός actor)
[*θα μπορούσε να δεχθεί κανείς και* Αρχεία Εισόδου]

Preconditions

Ύπαρξη των αρχείων εισόδου στη σωστή format (εναλλακτικά: τίποτε εδώ, αν στην use case χειριζόμασταν την περίπτωση απουσίας / κακού input – *τα preconditions είναι προαιρετικά*)

Basic Flow

1. Η UC ξεκινά όταν ο ιδιοκτήτης του εστιατορίου ξεκινά την αξιολόγηση
2. Το σύστημα φορτώνει τις εγγραφές από τα αρχεία μαγείρων, πιάτων, αξιολογήσεων
3. Το σύστημα αξιολογεί τα πιάτα.
   3.1. Για κάθε πιάτο
      3.1.1 Το σύστημα υπολογίζει το μέσο όρο αστεριών
      3.1.2 Το σύστημα υπολογίζει το μέσο χρόνο κατασκευής του.
4. Το σύστημα αξιολογεί τους σεφ.
   4.1. Για κάθε σεφ
      4.1.1 Το σύστημα υπολογίζει το μέσο όρο αστεριών
      4.1.2 Το σύστημα υπολογίζει το μισθό του.
5. Το σύστημα παράγει αναφορές *//Το 5. θα μπορούσε και να φτάνει, αλλά, αν θέλετε πάει και:*
   5.1 Το σύστημα παράγει αναφορά με όλα τα στοιχεία του προσωπικού
   5.2 Το σύστημα παράγει αναφορά με όλα τα στοιχεία του πιάτων

Extensions / Variations

Καμία
[εδώ θα μπορούσε να χειριστεί κανείς το exception από κάποιο λάθος ή απουσία στο input αν ήταν σημαντικό να γίνει έτσι]

Post conditions

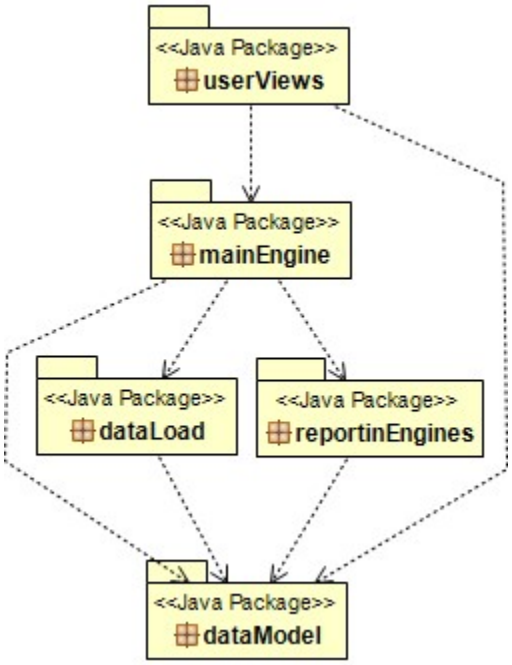
Οι αναφορές αξιολόγησης πιάτων και σεφ έχουν παραχθεί και η τοποθεσία τους αναφέρεται στον ιδιοκτήτη του εστιατορίου

Special Requirements, Issues, Risks and other Comments

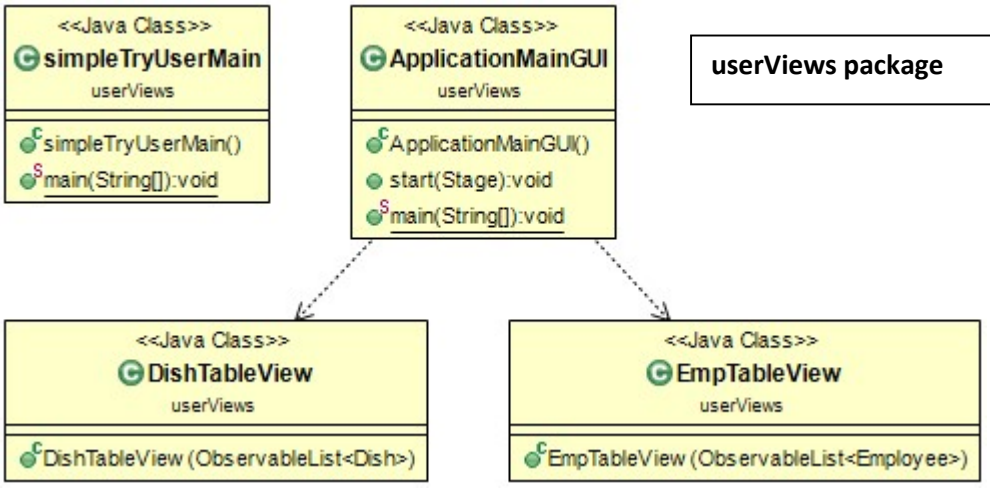
[Μπορείτε και χωρίς κάποιο σχόλιο εδώ.]

Δεν αντιμετωπίζεται η περίπτωση σφάλματος στα αρχεία εισόδου.

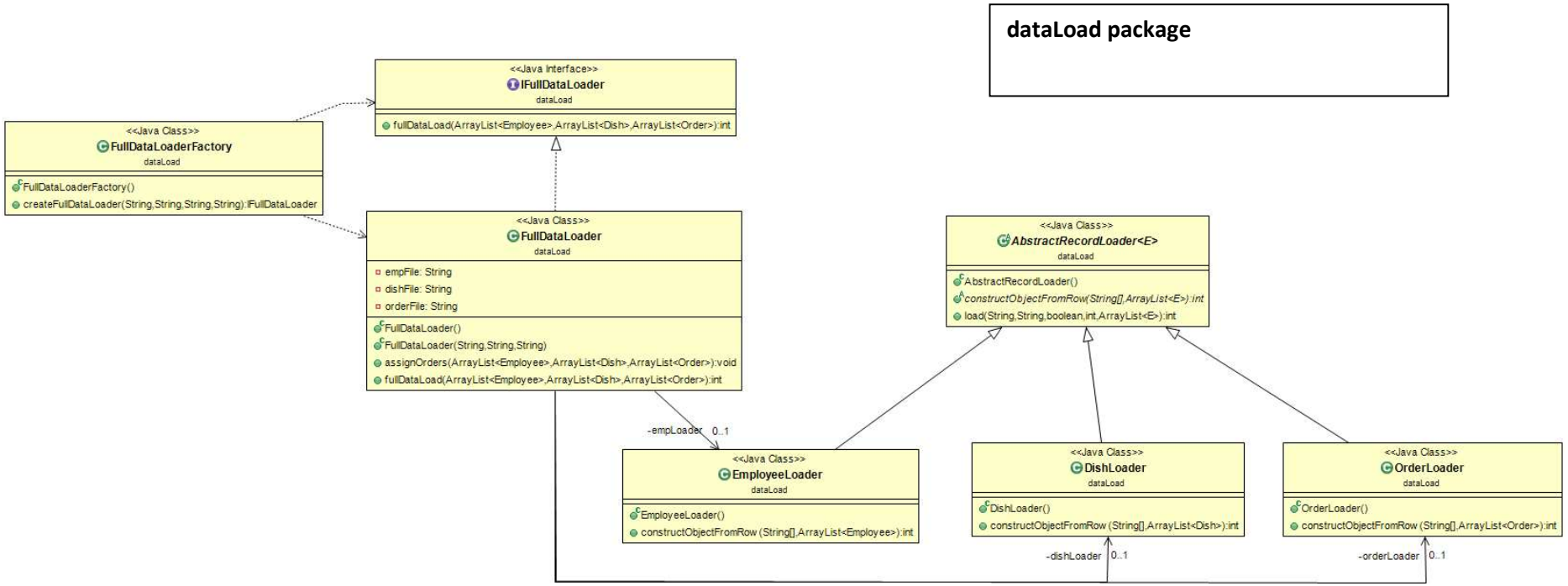
CLASS DIAGRAMS

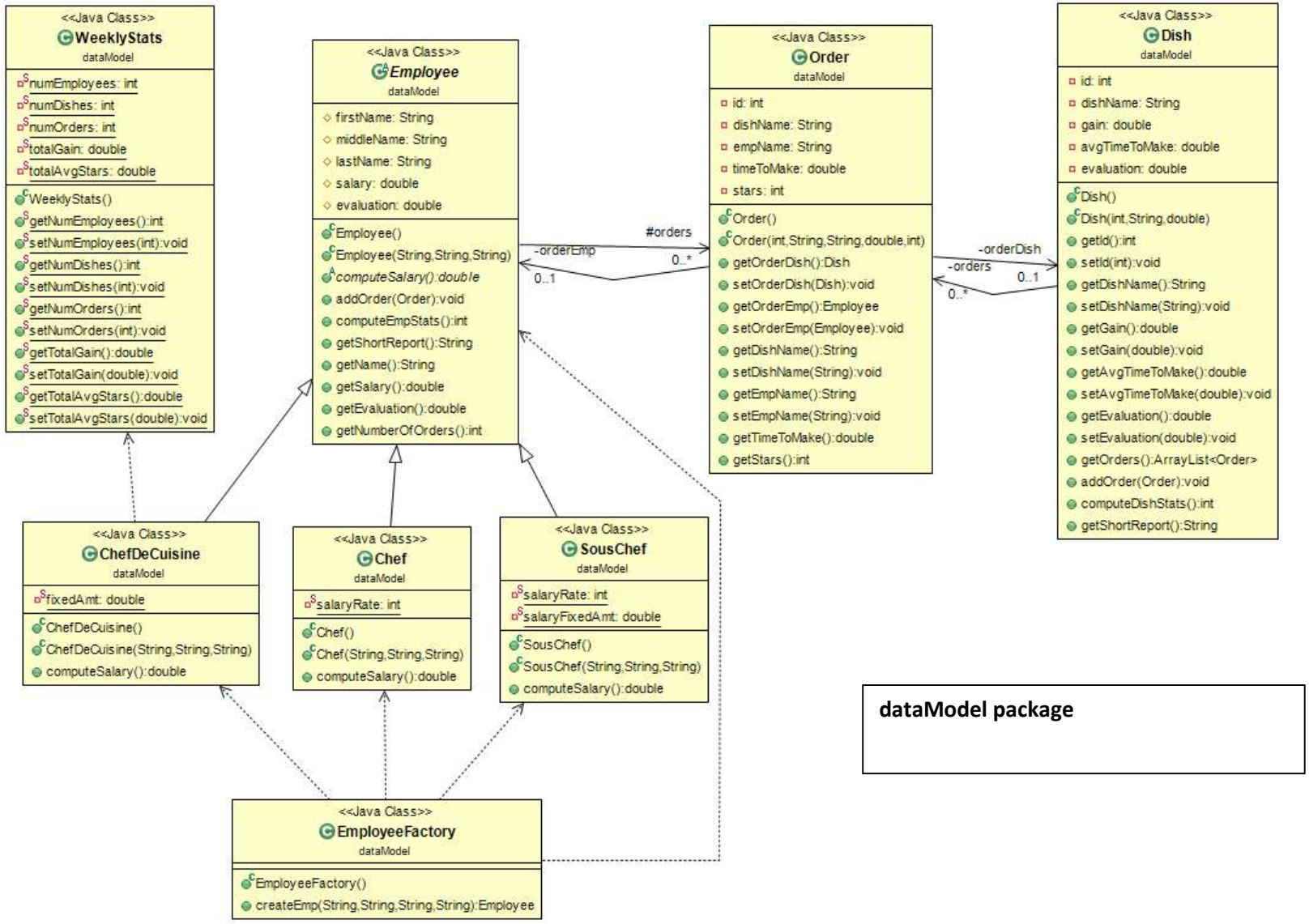


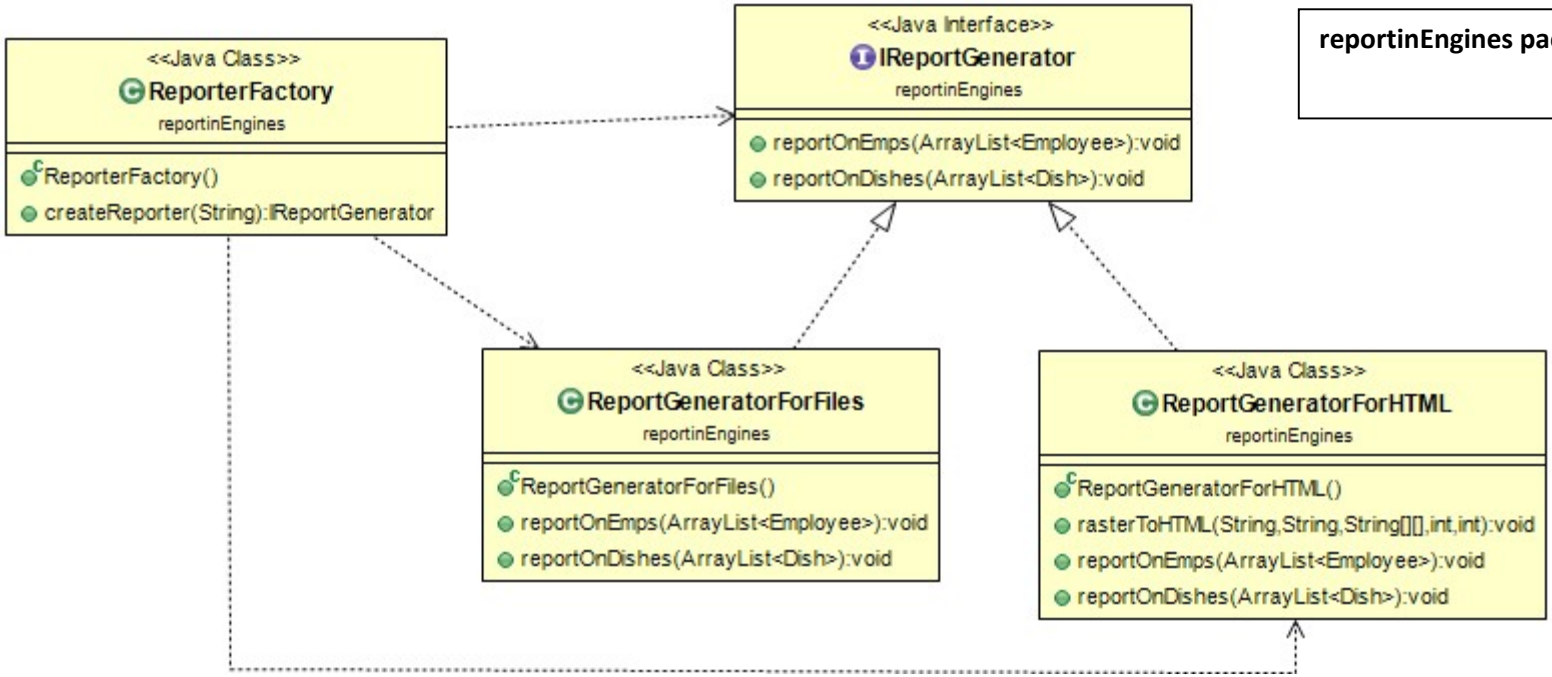
Package diagram



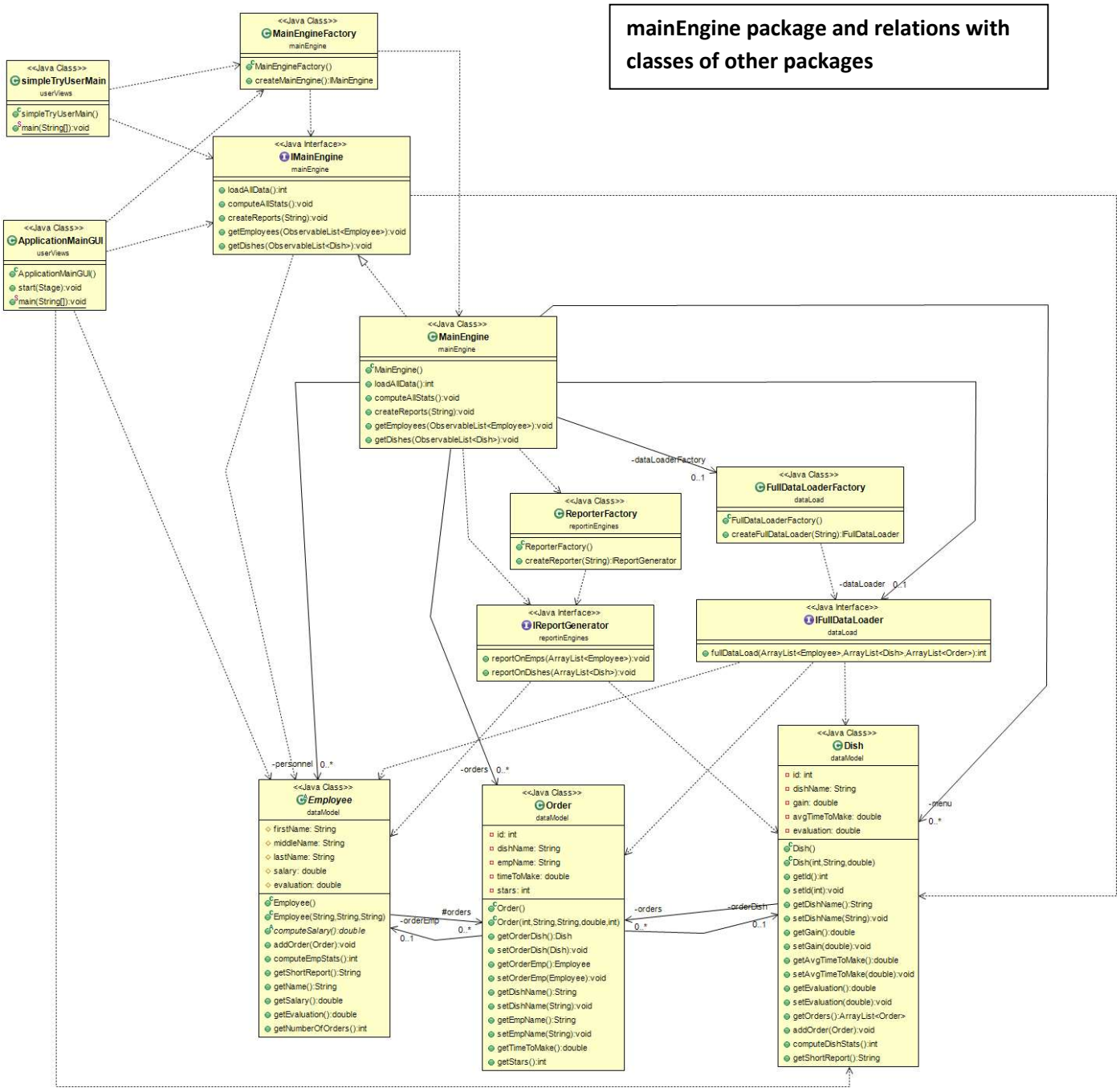
userViews package


dataLoad package
<<Java Interface>>
IFullDataLoader
dataLoad
fullDataLoad(ArrayList<Employee>,ArrayList<Dish>,ArrayList<Order>):int
<<Java Class>>
FullDataLoaderFactory
dataLoad
FullDataLoaderFactory()
createFullDataLoader(String,String,String,String):IFullDataLoader
<<Java Class>>
FullDataLoader
dataLoad
empFile: String
dishFile: String
orderFile: String
FullDataLoader()
FullDataLoader(String,String,String)
assignOrders(ArrayList<Employee>,ArrayList<Dish>,ArrayList<Order>):void
fullDataLoad(ArrayList<Employee>,ArrayList<Dish>,ArrayList<Order>):int
<<Java Class>>
AbstractRecordLoader<E>
dataLoad
AbstractRecordLoader()
constructObjectFromRow(String[],ArrayList<E>):int
load(String,String,boolean,int,ArrayList<E>):int
-empLoader 0..1
<<Java Class>>
EmployeeLoader
dataLoad
EmployeeLoader()
constructObjectFromRow (String[],ArrayList<Employee>):int
<<Java Class>>
DishLoader
dataLoad
DishLoader()
constructObjectFromRow (String[],ArrayList<Dish>):int
-dishLoader 0..1
<<Java Class>>
OrderLoader
dataLoad
OrderLoader()
constructObjectFromRow (String[],ArrayList<Order>):int
-orderLoader 0..1


<<Java Class>>
WeeklyStats
dataModel
numEmployees: int
numDishes: int
numOrders: int
totalGain: double
totalAvgStars: double
WeeklyStats()
getNumEmployees():int
setNumEmployees(int):void
getNumDishes():int
setNumDishes(int):void
getNumOrders():int
setNumOrders(int):void
getTotalGain():double
setTotalGain(double):void
getTotalAvgStars():double
setTotalAvgStars(double):void
<<Java Class>>
Employee
dataModel
firstName: String
middleName: String
lastName: String
salary: double
evaluation: double
Employee()
Employee(String,String,String)
computeSalary():double
addOrder(Order):void
computeEmpStats():int
getShortReport():String
getName():String
getSalary():double
getEvaluation():double
getNumberOfOrders():int
#orders
-orderEmp
0..*
0..1
<<Java Class>>
Order
dataModel
id: int
dishName: String
empName: String
timeToMake: double
stars: int
Order()
Order(int,String,String,double,int)
getOrderDish():Dish
setOrderDish(Dish):void
getOrderEmp():Employee
setOrderEmp(Employee):void
getDishName():String
setDishName(String):void
getEmpName():String
setEmpName(String):void
getTimeToMake():double
getStars():int
-orderDish
-orders
0..1
0..*
<<Java Class>>
Dish
dataModel
id: int
dishName: String
gain: double
avgTimeToMake: double
evaluation: double
Dish()
Dish(int,String,double)
getId():int
setId(int):void
getDishName():String
setDishName(String):void
getGain():double
setGain(double):void
getAvgTimeToMake():double
setAvgTimeToMake(double):void
getEvaluation():double
setEvaluation(double):void
getOrders():ArrayList<Order>
addOrder(Order):void
computeDishStats():int
getShortReport():String
<<Java Class>>
ChefDeCuisine
dataModel
fixedAmt: double
ChefDeCuisine()
ChefDeCuisine(String,String,String)
computeSalary():double
<<Java Class>>
Chef
dataModel
salaryRate: int
Chef()
Chef(String,String,String)
computeSalary():double
<<Java Class>>
SousChef
dataModel
salaryRate: int
salaryFixedAmt: double
SousChef()
SousChef(String,String,String)
computeSalary():double
<<Java Class>>
EmployeeFactory
dataModel
EmployeeFactory()
createEmp(String,String,String,String):Employee
dataModel package


reportinEngines package
<<Java Interface>>
IReportGenerator
reportinEngines
reportOnEmps(ArrayList<Employee>):void
reportOnDishes(ArrayList<Dish>):void
<<Java Class>>
ReporterFactory
reportinEngines
ReporterFactory()
createReporter(String):IReportGenerator
<<Java Class>>
ReportGeneratorForFiles
reportinEngines
ReportGeneratorForFiles()
reportOnEmps(ArrayList<Employee>):void
reportOnDishes(ArrayList<Dish>):void
<<Java Class>>
ReportGeneratorForHTML
reportinEngines
ReportGeneratorForHTML()
rasterToHTML(String,String,String[][],int,int):void
reportOnEmps(ArrayList<Employee>):void
reportOnDishes(ArrayList<Dish>):void

mainEngine package and relations with classes of other packages

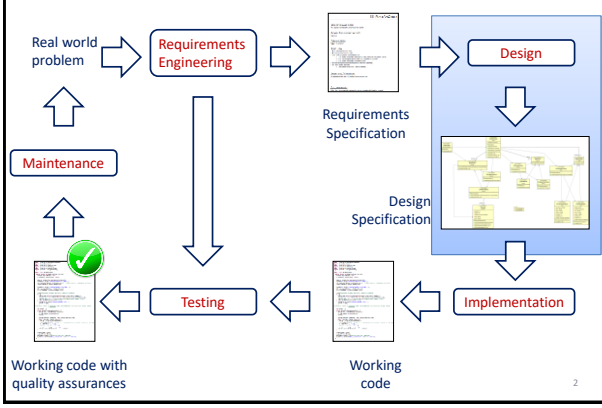


OO Design Principles

Ανάπτυξη Λογισμικού (Software Development)

www.cs.uoi.gr/~pvassil/courses/sw_dev/

ΜΥΥ301/ΠΛΥ 308



Αντικείμενο & εκπ. στόχοι της ενότητας

- Το αντικείμενο της ενότητας αφορά σε βέλτιστες πρακτικές και αρχές σχεδίασης αντικειμενοστρεφούς λογισμικού (Principles of OO Design)
- Ολοκληρώνοντας την ενότητα θα είστε εις θέση
  - Να γνωρίζετε τις βασικές αρχές του OOP
  - Να μπορείτε να τις ανακαλέσετε και να τις εφαρμόσετε στις δικές σας σχεδιάσεις
  - Να μπορείτε να αντιληφθείτε πιθανούς λόγους που οδήγησαν σε σχεδιάσεις που σας δίδονται
  - Να έχετε μια πρώτη εικόνα του τι πρέπει και μπορείτε να διερευνήσετε περισσότερο στο αντικείμενο

Αρχές καλής σχεδίασης

- Υπάρχουν βέλτιστες τεχνικές, όπως προέκυψαν μετά από πολλά (ανθρωπο)χρόνια ανάπτυξης που σκοπό έχουν να πιστοποιήσουν την καλή σχεδίαση ενός συστήματος
  - Μετάφραση: την οργάνωση των βασικών δομών και συστατικών του και –σε πολύ μεγάλο βαθμό- των αλληλοεξαρτήσεών τους

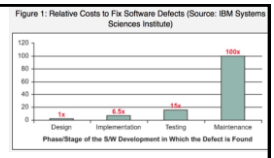
Προβλήματα σχεδίασης: μη συντηρησιμότητα

- Δυσκαμψία (Rigidity): Το σύστημα είναι δύσκολο να τροποποιηθεί διότι κάθε αλλαγή οδηγεί σε πληθώρα αλλαγών σε άλλα τμήματα του συστήματος
- Έλλειψη ρευστότητας (Viscosity): Η πραγματοποίηση τροποποιήσεων με λάθος τρόπο είναι ευκολότερη από την πραγματοποίησή τους με τον ορθό τρόπο.
- Ευθραυστότητα (Fragility): Οι αλλαγές που πραγματοποιούνται στο λογισμικό προκαλούν σφάλματα σε διάφορα σημεία.

Προβλήματα σχεδίασης: πολυπλοκότητα && δυσκολία κατανόησης

- Ακινησία (Immobility): Υπάρχει δυσκολία διαχωρισμού του συστήματος σε συστατικά τα οποία μπορούν να επαναχρησιμοποιηθούν σε άλλες εφαρμογές.
- Περιττή Επανάληψη (Needless Repetition): Η σχεδίαση περιλαμβάνει επαναλαμβανόμενες δομές που θα μπορούσαν να ενοποιηθούν υπό μία κοινή αφαίρεση.
- Περιττή Πολυπλοκότητα (Needless Complexity): Το λογισμικό περιλαμβάνει στοιχεία που δεν είναι (ούτε πρόκειται να γίνουν) χρήσιμα.
- Αδιαφάνεια (Opacity): Δυσκολία κατανόησης μιας μονάδας (σε επίπεδο σχεδίου ή κώδικα).

Αν παρατηρήσατε:

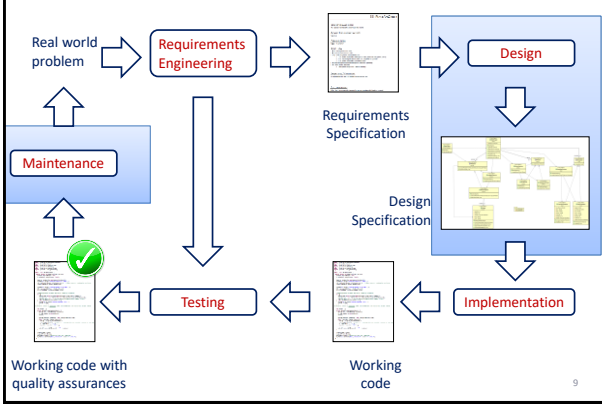


- Η συντήρηση (60%-80% του κόστους)
- Η μη κατανοησιμότητα
- Η πολυπλοκότητα
- … είναι τα βασικά προβλήματά μας (και όχι π.χ., η επίδοση)

Η απλότητα του κώδικα είναι μέγιστη αρετή!

Αρχές ανάπτυξης αντικειμενοστρεφούς λογισμικού

- Για να αντιμετωπισθούν τα προβλήματα συντήρησης και κατανόησης, η επιστημονική κοινότητα της τεχνολογίας λογισμικού έχει συμπυκνώσει την πρακτική γνώση από δεκαετίες επιτυχιών και αποτυχιών σε βέλτιστες πρακτικές και αρχές σχεδίασης αντικειμενοστρεφούς λογισμικού (Principles of OO Design)
- Μπορείτε να αποτείνεστε στο web site Robert Cecil Martin (aka Uncle Bob) ως αξιόπιστο σημείο αναφοράς για υλικό, συζήτηση και παραδείγματα
  http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod



Η αρχή της χαμηλής σύζευξης και η μετρική CBO

LOW COUPLING

Subsystems

- Η συνολική μας σχεδίαση μπορεί να καταλήξει σε ένα πολύ μεγάλο αριθμό από κλάσεις. Η διαχείριση μεγάλου τέτοιου αριθμού κλάσεων είναι δύσκολη =>
- … τις οργανώνουμε σε υποσυστήματα τα οποία σκοπό έχουν να ενσωματώσουν κλάσεις που συνεργάζονται για να πετύχουν ένα υποσύνολο της λειτουργίας του συνολικού συστήματος

Αρχή της χαμηλής σύζευξης

- Πάντα θέλουμε τα επί μέρους συστατικά (modules) του συστήματός μας να έχουν όσο το δυνατόν μικρότερη σύζευξη
- Συστατικά: κλάσεις, packages, subsystems
- Σύζευξη (coupling) ενός module: ο βαθμός αλληλεξάρτησης με άλλα modules

Σύζευξη (coupling)

- Πρακτικά, η σύζευξη αναφέρεται στο πόσο ευπαθές είναι ένα module από την πιθανότητα αλλαγής σε ένα άλλο module
- Όσο πιο στενά συσχετιζόμενο είναι ένα module με άλλα, τόσο πιο μεγάλο coupling έχει, και τόσο πιο ευπαθές είναι όταν τα άλλα αλλάζουν =>

  Always try to minimize coupling!

Πώς το πετυχαίνουμε?

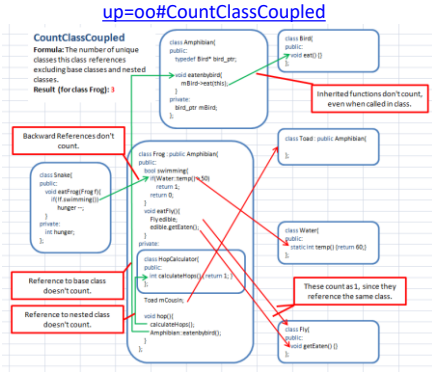
- Θα δούμε στη συνέχεια βασικές τεχνικές που σκοπό έχουν ακριβώς τη μειωμένη σύζευξη των συστατικών ενός συστήματος.
- Οι δύο βασικές μέθοδοι:
  - Ενθυλάκωση
    - Διότι κρύβει τις τεχνικές ανθυπολεπτομέρειες των στοιχείων και παρέχει διαπροσωπείες (που αλλάζουν πιο δύσκολα)
  - Abstract coupling
    - Διότι η εξάρτηση γίνεται σε σχέση με μια αφαιρετική δομή (interface or abstract class) που αλλάζει δύσκολα, χωρίς να εμπλέκονται οι concrete materializations της

Coupling Between Objects (CBO)

- Η μετρική CBO, για μια κλάση Α, μετρά τον αριθμό των κλάσεων από τις οποίες η Α εξαρτάται (πρακτικά: ως προς τη συντήρησή της). Η κλάση Α εξαρτάται από την κλάση Β, όταν η κλάση Α:
  - Έχει κάποιο πεδίο τύπου Β
  - Έχει μια μέθοδο που έχει ως παράμετρο ένα αντικείμενο της κλάσης Β
  - Έχει μια μέθοδο, η οποία στον κώδικά της περιέχει μια μεταβλητή που είναι τύπου Β

  //πρακτικά όλα αυτά έχουν αξία όταν καλούν κάποια μέθοδο της Β, αλλιώς why bother?
- ΔΕΝ μετράμε:
  - Την ίδια κλάση παραπάνω από μία φορές (αφού η CBO μετρά αριθμό κλάσεων από τις οποίες η Α εξαρτάται και όχι εξαρτήσεων)
  - Τις ιεραρχίες
    - αν η κλάση Α κληρονομεί από μια άλλη MamaA, η MamaA δεν προσμετρείται στην CBO, ούτε και οι εξαρτήσεις της
    - αν η κλάση Β κληρονομεί από μια άλλη MamaB, ούτε η mamaB περιλαμβάνεται στο μέτρημα

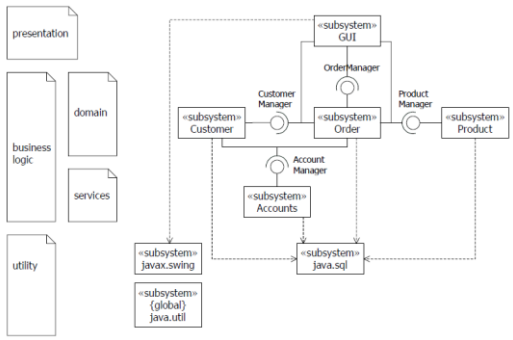
Παράδειγμα (C++) από
http://www.scitools.com/documents/metricsList.php?metricGroup=oo#CountClassCoupled



UML Interfaces, Components, Subsystems

TO PROBE FURTHER

από Arlow – Neustadt (2nd v.): δείτε πώς τα interfaces λειτουργούν ως κόλλα ανάμεσα στα υποσυστήματα



Information hiding – law of Demeter

ENCAPSULATION - RELATED

19

The encapsulation principle

- Internal state must be alterable only via the public interface of an object
- Με άλλα λόγια, δεν βγάζουμε ποτέ την κατάσταση του αντικειμένου σε δημόσια πρόσβαση, αλλά παρέχουμε μεθόδους για τη δουλειά αυτή
- .. με σκοπό να προλάβουμε την αλλαγή της κατάστασης του αντικειμένου με λάθος τρόπο …

20

Ενθυλάκωση

- Αυτό έχει να κάνει με το να μην αφήσουμε τον client κώδικα να αλλάξει το αντικείμενο
  - με λάθος τιμές (π.χ., αρνητικές ηλικίες, λάθος ημερομηνίες, …)
  - με ασυνέπεια μεταξύ αντικειμένων (π.χ., πολλές φορές αλλάζοντας ένα αντικείμενο πρέπει να αλλάξουμε και κάποια άλλα)
  - …
- Όλες οι δημόσιες μέθοδοι που αλλάζουν ένα αντικείμενο πρέπει να έχουν ελέγχους ώστε η κατάσταση του αντικειμένου να είναι συνεπής μετά την αλλαγή
  - Μετάφραση: Η κατάσταση των αντικειμένων πρέπει να σέβεται το σύνολο των λογικών κανόνων/περιορισμών ορθότητας που θέτει ο σχεδιαστής (και είναι γνωστοί ως class invariants)
  - Ναι, αυτό καλά κάνει και σας θυμίζει pre/post conditions…

21

Παράδειγμα

Έστω η κλάση `TimeStamp` που έχει ως δημόσια πεδία

```
int hour; int minute; int second;
```

Έστω και ο client κώδικας

```
TimeStamp t = new TimeStamp(23,59,30);
t.hour++;
```

Θα προκύψει ένα μη-έγκυρο αντικείμενο με ώρα 24:59:30

Τι θα έπρεπε να έχει γίνει? Μέσα στην κλάση μια μέθοδος

```
public void incrementHour(){
   hour++;
   if (hour == 24) hour = 0;
}
```

από το βιβλίο «Αντ. Σχεδίαση» του Α. Χ"γεωργίου

22

Ποτέ public πεδία!

Πρόσβαση μόνο μέσω δημόσιων μεθόδων

Internal state is radioactive

23

Ο νόμος της Δήμητρας

- Ένας όχι απολύτως αποδεκτός «νόμος», που σχετίζεται με την αρχή της ενθυλάκωσης είναι ο νόμος της Δήμητρας («Demeter's law»)
- Εισήχθη στο Παν. Northwestern σε ένα έργο με όνομα «Demeter Project» το 1987 – εξ ού και το όνομα του νόμου
- Στόχο έχει να μειώσει το εύρος της εξάρτησης των κλάσεων

- http://www.ccs.neu.edu/home/lieber/LoD.html
- http://c2.com/cgi/wiki?LawOfDemeter

24

Ο νόμος της Δήμητρας

- Ο νόμος λέει ότι μια μέθοδος m ενός αντικειμένου O μπορεί να καλέσει μόνο τις εξής μεθόδους:
  - Τις μεθόδου του O
  - Τις μεθόδους των αντικειμένων που περνούν ως παράμετροι στη μέθοδο m
  - Τις μεθόδους όποιων αντικειμένων δημιουργούνται μέσα στην m
  - Τις μεθόδους πεδίων της O
  - ~~Τις μεθόδους όποιας global variable είναι στο scope του O~~ (Ποτέ global variables!)
- Από τα παραπάνω, συνάγεται ότι ένα αντικείμενο δεν πρέπει να προσπελάζει μεθόδους από αντικείμενα που προκύπτουν ως αποτελέσματα επιστροφής μεθόδων

  Talk only to your friends; don't talk to strangers
- (Κακή) περίληψη του νόμου: "use only one dot". Π.χ., το a.b.Method() συνιστά μια καθαρή παραβίαση του νόμου

25

Πρακτικές επιπτώσεις

- Αν ένας πελάτης έχει μια παραγγελία και η παραγγελία έχει ένα σύνολο από OrderItems, ο πελάτης για να αλλάξει το status ενός από αυτά πρέπει να το κάνει ΜΟΝΟ μέσω της παραγγελίας =>
- Η παραγγελία πρέπει να έχει μια δημόσια μέθοδο που να αλλάζει το status ενός OrderItem

26

What not to do in the client

```
//Bad Client code
Order o = getOrder();
OrderItem oi = o.getItem(3);
Oi.setStatus("delivered");
```

```
//Bad Client code, equivalent to the above
getOrder().getItem(3).setStatus("delivered");
```

```
//άρα δεν είναι ο σιδηρόδρομος των "." το θέμα…
```

27

Γιατί?

- Διότι ξαφνικά ο client εξαρτάται ΚΑΙ από την Order ΚΑΙ από το OrderItem …
- … αλλαγή σε οποιοδήποτε από τα δύο συνεπάγεται και έλεγχο του client για πιθανές επιπτώσεις
- Client.coupling = 2

Πώς να το κάνουμε

```
Class Order
…
public void setOrderItemStatus(int orderItemId, String
   newStatus){
   OrderItem oi = getItem(orderItemId);
   if (oi !=null)
       oi.setStatus(newStatus);
}
```

```
//Client code with coupling = 1
Order o = getOrder();
o.setOrderItemStatus(3, “delivered”);
```


Και τι πληρώνουμε?

- Επιπλέον μεθόδους για κάθε κομμάτι μιας κλάσης το οποίο θέλουμε ο client κώδικας να διαχειριστεί
- Αυτό μπορεί να γίνει ένας εξαιρετικά μεγάλος αριθμός μεθόδων σε πολύπλοκες εσωτερικές δομήσεις αντικειμένων και αποτελεί και το βασικό λόγο αντιρρήσεων εναντίον του νόμου

  http://c2.com/cgi/wiki?LawOfDemeterIsTooRestrictive

Πότε φαίνεται ότι ο νόμος μπορεί να σπάει εύλογα

- Σε value objects, δηλ. απλά αντικείμενα που επί της ουσίας η τιμή τους τα χαρακτηρίζει (π.χ., συμβολοσειρές, ημερομηνίες, …)
- Σε συλλογές και lambda streams
  - Also in this category: System.out.print*
- Σε Factories, δηλαδή σε κλάσεις που εισάγονται τεχνητώς για να ενσωματώσουν τις βαριάντες κατασκευής αντικειμένων σε ένα (1) σημείο του κώδικα

```
Customer john = new Customer("John", 15);
Customer sarah = new Customer("Sarah", 200);
Customer charles = new Customer("Charles",150);
Customer mary = new Customer("Mary", 1);

List<Customer> customers = Arrays.asList(john,
sarah, charles, mary);

List<Customer> charlesWithMoreThan100Points =
customers
  .stream()
  .filter(c -> c.getPoints() > 100 &&
          c.getName().startsWith("Charles"))
  .collect(Collectors.toList());
```

https://www.baeldung.com/java-stream-filter-lambda

31

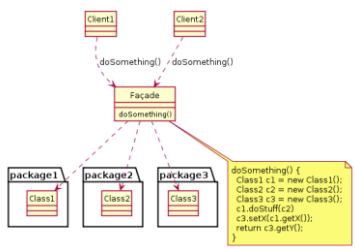
Façade pattern

Visitor pattern – mainly related to dependency inversion

TO PROBE FURTHER

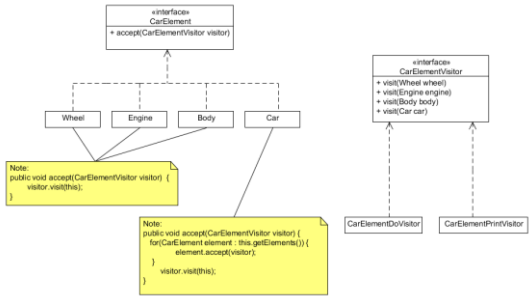
32

Façade Pattern (see http://en.wikipedia.org/wiki/Facade_pattern)



33

Visitor pattern
(http://en.wikipedia.org/wiki/Visitor_pattern)



34

Abstract coupling (the Strategy pattern)

OPEN-CLOSED PRINCIPLE (OCP)

35

Μια βασική αρχή σχεδίασης …

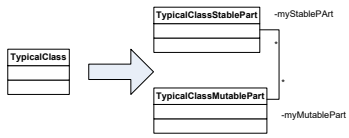
- Όταν σχεδιάζουμε, έχουμε στο πίσω μέρος του νου μας μια βασική αρχή που διέπει (από την εποχή του Πλάτωνα) τα αντικείμενα

Αντικείμενο = σταθερό μέρος + μεταβλητό μέρος

- Από πλευράς συντήρησης, αν ξέρουμε / έχουμε σχεδιάσει πώς μπορεί να εξελιχθεί μια κλάση, μπορούμε να εκμεταλλευθούμε το σταθερό κομμάτι για να αποφύγουμε τη διάδοση της αλλαγής

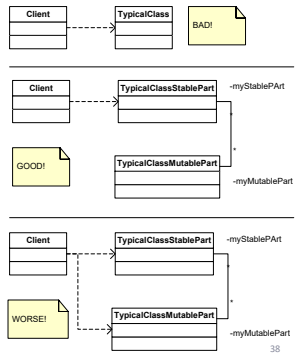
36

Η βασική ιδέα

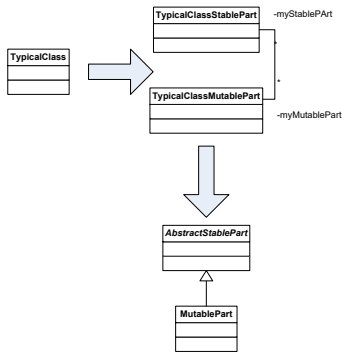


Και γιατί είναι καλή ιδέα αυτή?

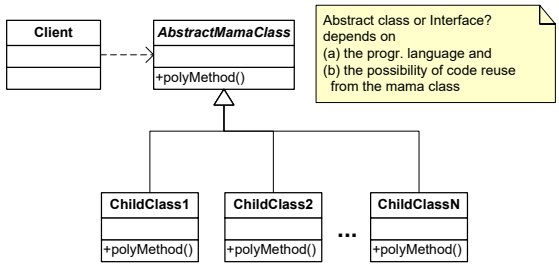
- Διότι μπορούμε (όσο γίνεται) να βάλουμε τους clients να εξαρτώνται ΜΟΝΟ από το σταθερό μέρος, ώστε όταν αλλάζει το μεταβλητό μέρος, αυτοί να μένουν ανεπηρέαστοι στην αλλαγή



Ακόμα καλύτερα …



The Strategy Pattern



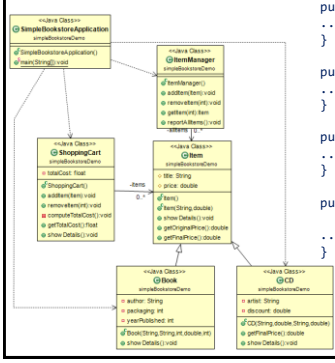
40

Γνωστό και ως Abstract Coupling

- Ο client κώδικας εξαρτάται ΜΟΝΟ από την αφηρημένη μητρική κλάση
- Η αφηρημένη κλάση επιβάλει στις υλοποιήσεις της (κλάσεις-παιδιά) να υλοποιήσουν την (τις) virtual method(s) που χρησιμοποιεί ο client
- Έτσι, ο client είναι ΑΝΕΞΑΡΤΗΤΟΣ από το ποιες υποκλάσεις υπάρχουν στην πράξη και χρειάζεται να ξέρει μόνο την αφηρημένη κλάση

41

Θυμάστε το παράδειγμα από το online bookstore?



```
package simpleBookstoreDemo;

public class Item {
...
}

public class Book extends Item {
...
}

public class CD extends Item {
...
}

public class ItemManager {
     private ArrayList<Item> allItems;
...
}
```

42

Abstract class

For you: can / should we use an **interface** instead?

```
package bookstoreAcceptable;

public abstract class Item {
   public Item(){title="";price=-1.0;id=-1}
   public Item(String aTitle, double aPrice,int id){title =
   aTitle; price=aPrice;this.id =id;}
   public abstract void showDetails();
   public abstract String getDetails();
   public abstract double getFinalPrice();
   public double getOriginalPrice(){return price;}
   public String getTitle(){return title;}
   public int getId(){return id;}

   public abstract double getFinalPrice();

   protected String title;
   protected double price;
   protected int id;
}
```


One implementation

```
package bookstoreAcceptable;

public class Book extends Item {
      private String author;

…
      @Override
      public double getFinalPrice() {
            return price;
      }

}
```


Another implementation

```
public class CD extends Item {
       private String artist;

…
       @Override                                  //implem. abstract
       public double getFinalPrice() {
          return (price - discount);
       }

       public void showDetails() {   //override mama class' method
          super.showDetails();
          System.out.println("by " + artist);
          System.out.println("Price f.: "+ getFinalPrice()+ "\n");
       }
}
```

Abstract coupling

```
public class ItemManager {
  private ArrayList<Item> allItems;
…
}
```

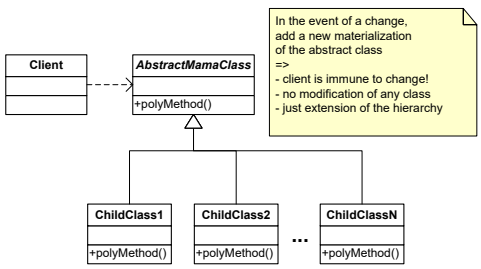
The “client” class ItemManager uses ONLY the abstract class and is completely agnostic to any subclasses the abstract class has.

=> Zero maintenance cost when new subclasses appear

Αρχή της ανοικτής – κλειστής σχεδίασης

- The Open-Closed Principle (B. Meyer, 1998)
- A module should be:
  - Open for extension
  - Closed for modification
- Βασικός στόχος: οι όποιες αλλαγές, να γίνονται με επέκταση και όχι με τροποποίηση του υπάρχοντος κώδικα (source, object or executable)!

Και πώς γίνεται αυτό? Με abstract coupling,φυσικά!!



Even better: use an interface, unless the abstraction has reusable state or behavior!!

Τι κερδίζουμε

- Μια αλλαγή στην ιεραρχία, δεν αλλάζει τον client (rigidity: large number of impacted modules for a single change)
- Για κάθε αντικείμενο που χρησιμοποιεί ο client κώδικας, δεν χρειάζεται να εξετάσουμε σε ποια κλάση ανήκει (fragility: must ensure that all such “switch” checks correspond to the appropriate classes)

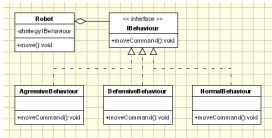
Template Method and Strategy Patterns
Factory – related patterns: see the next subsection

TO PROBE FURTHER

http://www.oodesign.com/strategy-pattern.html vs template-method-pattern.html

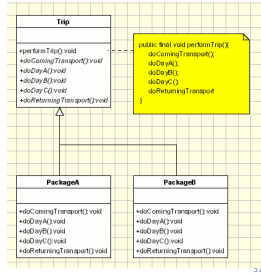
Strategy: behavior is abstracted && each time we “new” as we are supposed to

Also: can be interface-based only



Template method: when some steps are reusable and some have to be overridden.

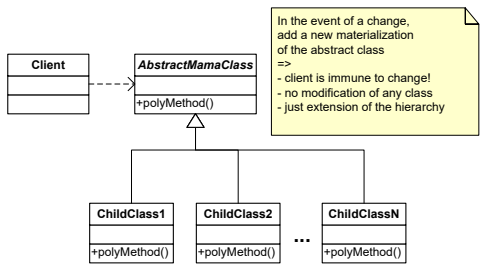
Also: needs abstract class, as there is reusable code



The simplest possible factory

FACTORIES

abstract coupling



Πού «πονά» το abstract coupling

- Ενώ το abstract coupling έχει το καλό ότι ενώνει μια κλάση «χρήστη» ΜΟΝΟ με ένα γονικό interface (ή abstract class) πάσχει στο ότι ΔΕΝ ΜΠΟΡΕΙ ΝΑ ΕΦΑΡΜΟΣΤΕΙ ΣΤΗΝ ΚΑΤΑΣΚΕΥΗ ΤΩΝ ΑΝΤΙΚΕΙΜΕΝΩΝ!!
- Ενώ στη χρήση δηλ., μας αρκεί μια μεταβλητή τύπου AbstractMamaClass, όταν κάνουμε new πρέπει να αναφερθούμε σε μια concrete κλάση

Πού «πονά» το abstract coupling

- … και μάλιστα, κάθε φορά πρέπει να ξέρουμε ποια κλάση θα επιλέξουμε για το αντικείμενο τύπου AbstractMamaClass που θα φτιάξουμε
- … με αποτέλεσμα, τελικά, ο client code να έχει coupling με (πιθανώς) ΟΛΕΣ τις υποκλάσεις της AbstractMamaClass
- … άρα, μόλις χάσαμε το πλεονέκτημα που κερδίσαμε με το abstract coupling. Πώς θα το αποκτήσουμε πίσω?

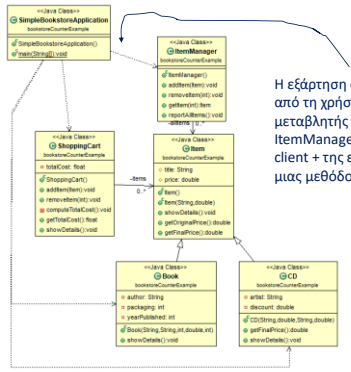
Πού προκύπτει η hard-coded σχέση με ΟΛΕΣ τις υποκλάσεις

```
public class SimpleBookstoreApplication {

    public static void main(String args[]){
        ItemManager amazon = new ItemManager();
        Book bookRef;
        bookRef = new Book("Discours de la methode", "Rene Descartes", 1637,
50.00, 0,0); amazon.addItem(bookRef);
        …
        CD cdRef;
        cdRef = new CD("Piece of Mind", 10.0,"Iron Maiden",4.0,1);
        amazon.addItem(cdRef);
        …
        amazon.reportAllItems();
    }
}
```


Φάρμακο: Factory

- Η λύση είναι να εισάγουμε μια κλάση, η οποία να επωμίζεται το κόστος της κατασκευής των αντικειμένων.
- Ονομάζουμε μια τέτοια κλάση “Factory”
- Η κλάση αυτή έχει τις εξής ιδιότητες:
  - Μαζί με την αφαιρετική μαμά κλάση, είναι οι μόνες που ξέρει ο client
  - Έχει μία ή περισσότερες μεθόδους για να κατασκευάζει αντικείμενα από τις concrete subclasses
  - Ξέρει ΟΛΕΣ τις υποκλάσεις

Πώς ήταν ο κώδικας αρχικά

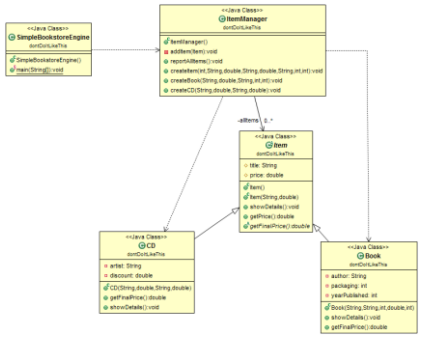


Η εξάρτηση αυτή είναι από τη χρήση μιας μεταβλητής τύπου ItemManager από τον client + της εκτέλεσης μιας μεθόδου γι' αυτήν

Οι εξαρτήσεις αυτές είναι μόνο από το new μέσα στον client

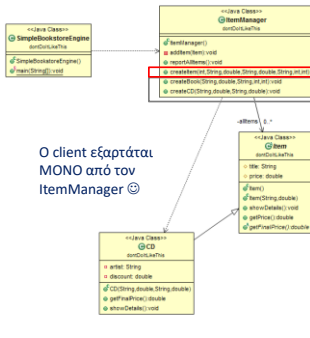
58

Αν έχετε manager, μία λύση είναι να βάλετε εκεί το factory



59

Κέρδη && ζημίες



Ο ItemManager απέκτησε μία μέθοδο γέννησης για κάθε υποκλάση ☹

ΠΡΟΣΟΧΗ: μπορούσαμε να βάλουμε ΜΟΝΟ ΜΙΑ (1) μέθοδο γέννησης (CreateItem) που να έχει όλα τα πεδία απ' όλες τις υποκλάσεις + ένα αναγνωριστικό για το ποιας κλάση θέλουμε να φτιάξουμε αντικείμενο. ΑΠΟΦΥΓΕΤΕ ΤΟ!!☹

Ο client εξαρτάται ΜΟΝΟ από τον ItemManager ☺

60

Όχι εξάρτηση από τις υποκλάσεις

```
public class SimpleBookstoreEngine {
    public static void main(String args[]){
        ItemManager amazon = new ItemManager();
        //Book bookRef;
        amazon.createBook("Discours de la methode", 50.00, "Rene Descartes", 1637,  0);


        //CD cdRef;
        amazon.createCD("Piece of Mind", 10.0,"Iron Maiden",4.0);
        amazon.reportAllItems();
    }
}

/* what you should NOT do:
 * amazon.createItem(1,"Discours de la methode", 50.00, "",0, "Rene Descartes", 1637,  0);
*/
```

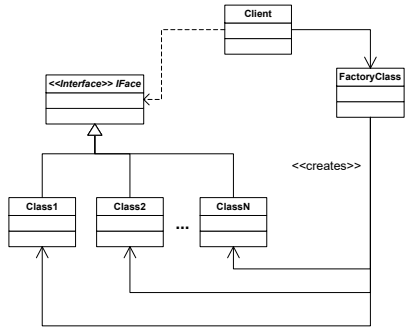

Κέρδη && ζημίες

- Είναι μεγάλο πλεονέκτημα ο client να είναι ελεύθερος από τις υποκλάσεις
- Μπορούμε να ζήσουμε με τη λύση αυτή (για το τίμημα μιας μεθόδου ανά υποκλάση) αν όποιος φτιάχνει το manager φτιάχνει και την ιεραρχία των υλοποιήσεων της Item
- Still, we can do better than that

Πριν προχωρήσουμε…

- Αποφύγετε τη μία και μόνη μέθοδο που θα διαλέγει τι είδους αντικείμενο θα κατασκευαστεί
- Είναι πολύ δύσκολο να ξέρεις ποιο πεδίο είναι τι
- Είναι πολύ επικίνδυνη στη συντήρηση
- Είναι υποχρεωτικό να συντηρείται σε κάθε αλλαγή σε υποκλάση
- Είναι στριφνή στον έλεγχο
- …

Το τυπικό factory: έχει μια dedicated class για τη γέννηση των αντικειμένων

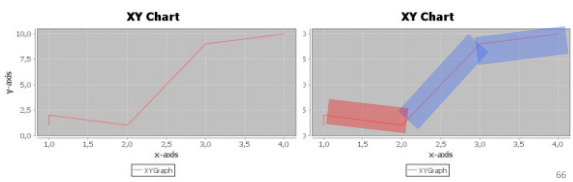


Πώς γεννιούνται τα αντικείμενα?

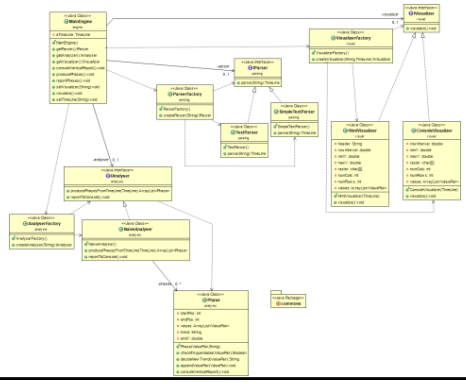
- Αν οι υποκλάσεις έχουν διαφορετική δομή στους constructors τους, τότε η πιο απλή λύση είναι να έχετε από μία μέθοδο για κάθε υποκλάση
- Αν οι constructors είναι ίδιοι, τότε μπορείτε να έχετε μία (1) μέθοδο για όλους, με μία παράμετρο, η οποία να καθορίζει ποιας υποκλάσης το αντικείμενο γεννιέται
  - Parameterized factory

Ένα παράδειγμα

- Έχουμε μια μηχανή data analytics που θέλει να εξάγει στατιστικά συμπεράσματα
- Εδώ: μια μηχανή που παίρνει για είσοδο ένα timeseries και θέλει να το σπάσει σε φάσεις
- (εδώ μια φάση καθόδου και μια ανόδου)



Architecture of the project



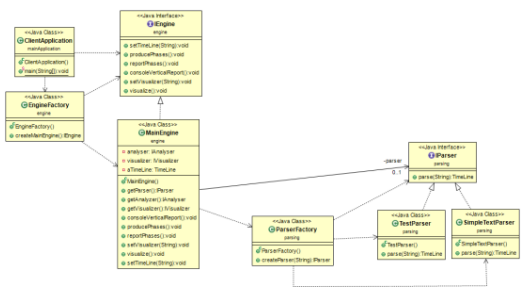
67

Ας δούμε ένα πακέτο μόνο: parsing

- Για να πάρουμε δεδομένα, συνήθως χρησιμοποιούμε ένα parser ο οποίος επεξεργάζεται ένα αρχείο εισόδου με τιμές και τις μετατρέπει σε ένα επιθυμητό αντικείμενο (ή συλλογή αντικειμένων)
- Εδώ, ας υποθέσουμε ότι έχουμε ένα parser for simple ASCII texts κι άλλον ένα που δίνει τιμές απ' ευθείας στη μηχανή (γρήγορο hack to test the rest of the implementation)

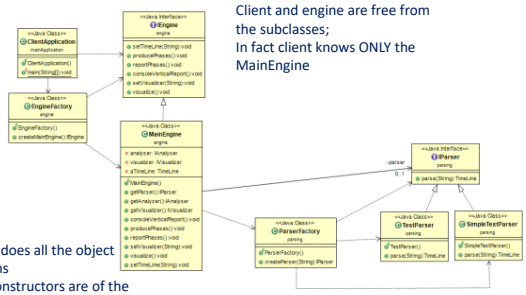
68

Ένα παράδειγμα factory



69

Ένα παράδειγμα factory



Client and engine are free from the subclasses;
In fact client knows ONLY the MainEngine

Factory does all the object creations
Since constructors are of the same signature, we can pick via a single (1) factory method

Parameterized Factory: Simple as that

```
package parsing;

public class ParserFactory {
    public IParser createParser(String concreteClassName){
        if (concreteClassName.equals("TestParser")){
                return new TestParser();
        }
        else if (concreteClassName.equals("SimpleTextParser")){
                return new SimpleTextParser();
        }
        System.out.println("If you got here, you passed a wrong
                argument to ParserFactory");
        return null;
    }
}
```

Όλο το κόλπο είναι ότι η createParser(), ενώ μέσα της κατασκευάζει αντικείμενα από concrete κλάσεις (άρα τα new γίνονται σωστά), συντακτικά επιστρέφει IParser, δλδ., just an interface-abiding object!!!

Αυτό σημαίνει ότι **οι clients** της ParserFactory, που καλούν την createParser() αντιμετωπίζουν το αποτέλεσμά της => **εξαρτώνται και ξέρουν ΜΟΝΟ το IParser** (και τη λειτουργικότητά του) και όχι τις concrete κλάσεις που το υλοποιούν!!!!

Remember to return an element of abstraction in factory methods!

public class ItemFactory

When method signatures are incompatible, use different methods!!

```
//DON'T DO IT!!!
public Item createItem(int aType,
  String aTitle, double aPrice,
  String anArtist, double aDiscount,
  String anAuthor, int aDate, int
  aPackage,int id){
   switch(aType){
    case 1: Book newBook = new
            Book(aTitle, anAuthor,
            aDate,
            aPrice,aPackage,id);
             return newBook;
    case 2: CD newCD = new
            CD(aTitle, aPrice,
            anArtist, aDiscount, id);
             return newCD;
    default:
            System.out.println("Wrong
            type of item ");
              return null;
   }
}
```

```
//DO THIS!!!
public Item createBook(String
   aTitle, double aPrice,
   String anAuthor, int aDate,
   int aPackage,int id){
    Book newBook = new
       Book(aTitle, anAuthor,
       aDate, aPrice,aPackage,
       id);
return newBook;
}

public Item createCD(String
   aTitle, double aPrice,
   String anArtist, double
   aDiscount, int id){
    CD newCD = new CD(aTitle,
       aPrice, anArtist,
       aDiscount, id);
return newCD;
}
```

… and avoid the non-maintainable mega-constructor with the zillion parameters

Κέρδη && ζημίες

- Ο client και το Main Engine είναι ελεύθερα από την ιεραρχία των parsers
- Το coupling τους είναι:
  - 1 για τον client
  - 2 για το engine (no matter how many subclasses, it is always 2, one for the iface and one for the factory)
    - Και χωρίς factory θα ήταν 2, αλλά αυτό θα ήταν ίσο με τον αριθμό των υποκλάσεων => για κάθε νέα υποκλάση θα αύξανε…

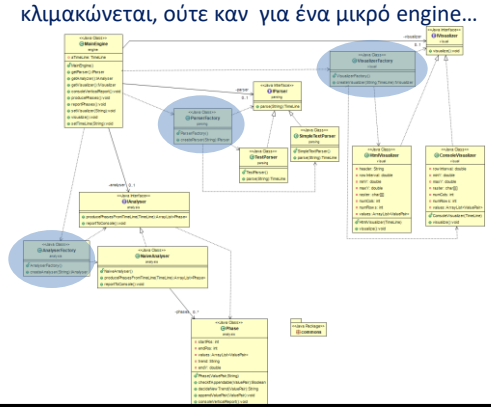
73

Κέρδη && ζημίες

- Για το κέρδος αυτό, πληρώσαμε:
  - Μια επιπλέον κλάση (το factory)
  - Coupling for the factory class = number of subclasses + 1 (for the interface)
- Έχουμε εισάγει δηλ., μια κλάση με όσο μεγαλύτερη εξάρτηση γίνεται από την ιεραρχία
- We are prepared to pay this price, if this is a dedicated, centralized, single point of maintenance
  - Με άλλα λόγια, ξέρουμε ότι όταν θα προκύψει νέα υποκλάση, θα πρέπει να συντηρήσουμε το Factory.
  - Όμως, είναι ένα (1) και μόνο, γνωστό, σημείο συντήρησης – a price worth paying, IMHO

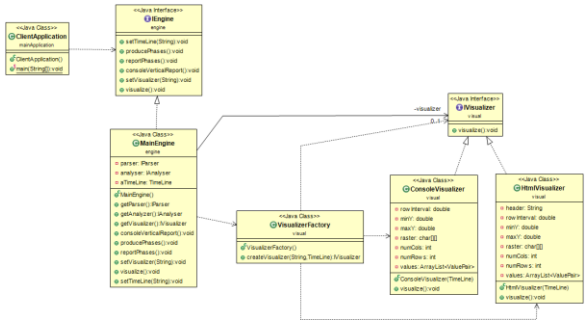
74

Να βάζαμε τις μεθόδους στο engine? Μπα, αυτό δε κλιμακώνεται, ούτε καν για ένα μικρό engine…



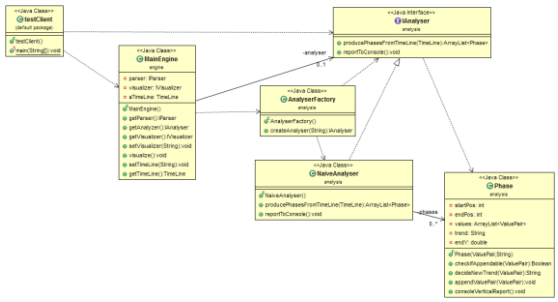
75

Η ίδια συνταγή



76

Για σας: πώς θα βελτιώνατε αυτό?



77

Παρένθεση

- Όπως μόλις είδατε, για ένα μεγάλο διάγραμμα, μπορώ να βγάλω υποσύνολα του
- Η ουσία είναι να καταλαβαίνουμε και να συνεννοούμαστε => μπορούμε να εστιάζουμε σε όποιο υποσύνολο θέλουμε αν αυτό εξυπηρετεί

78

Factory – related patterns: Factory mainly (also factory method, abstract factory)

Builder (for complicated factories)

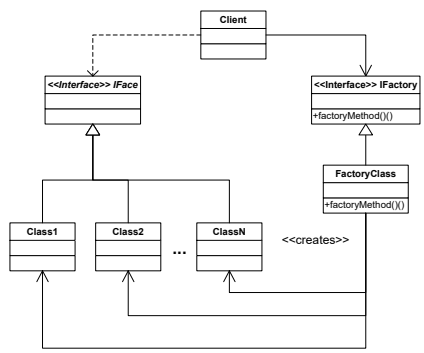
TO PROBE FURTHER

Factory

- http://www.oodesign.com/factory-pattern.html
- http://www.oodesign.com/factory-method-pattern.html
- http://www.oodesign.com/abstract-factory-pattern.html

- http://c2.com/cgi/wiki?AbstractFactory
- http://c2.com/cgi/wiki?FactoryMethodPattern

Factory Method: a variant with client totally free from concrete (i.e., mutable) classes



Builder pattern

- http://www.oodesign.com/builder-pattern.html
- http://en.wikipedia.org/wiki/Builder_pattern

82

Depend on abstractions!

DEPENDENCY INVERSION

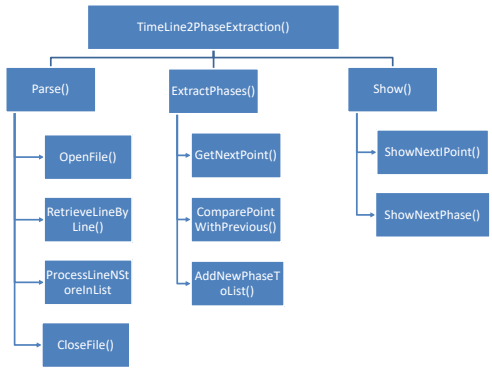
83

Παραδοσιακή Σχεδίαση

- Structured Analysis and Design Technique (SADT), Ross 1977: σπάσε κάθε δουλειά σε υπο-εργασίες (συναρτήσεις) που η μία καλεί τις άλλες μέσα της
- Κάθε σύνθετη δουλειά οργανώνεται σε επί μέρους τμήματα που το ένα καλεί το άλλο και μεταξύ τους επικοινωνούν με δομές δεδομένων

84

Παραδοσιακή Σχεδίαση



85

Παραδοσιακή Σχεδίαση

- Top-Down decomposition
- Calls from high-level to low-level =>
  - dependency of the high level from the low level ☹
- Glue on the basis of actual, implemented instances of data structures
  - Dependency of "everyone" from the implemented glue constructs
- Evolution of the low level or the glue SIGNIFICANTLY affects the high levels, and thus, PROPAGATES EASILY

86

Dependency Inversion Principle (DIP)

- It is high-level modules (classes)
  - … that determine how low-level modules will be implemented
  - … that should instruct how change is to be performed
  - … that should be immune to change from the details (i.e., low level or glue modules)

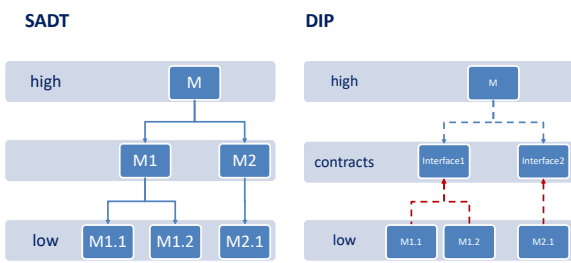
87

Dependency Inversion Principle Definition

- High level modules should not depend upon low level modules
  - BOTH should depend upon abstractions
- Abstractions should not depend upon details
  - Details should depend upon abstractions

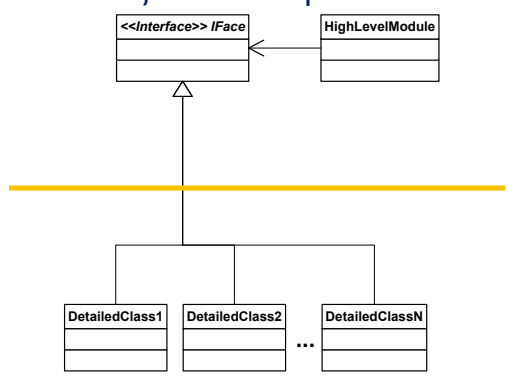
88

Dependency Inversion Principles



89

Πώς υλοποιούμε το DIP



90

Πώς υλοποιούμε το DIP

- The interface is a contract between the high and the low level developers
- Typically, the interface is owned by the high level (orange line separates high and low)
- It's the high level (interface included) that determines what will be implemented in the low level
- HIGH LEVEL SHOULD BE PLANNED TO REMAIN STABLE!!!!

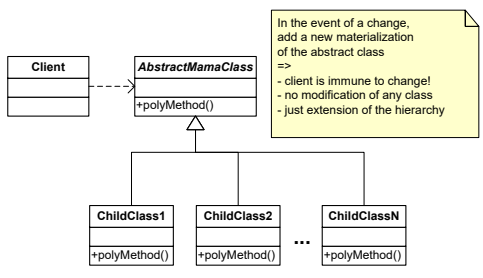
91

Uncle Bob says: Depend upon abstractions ONLY!

- No attributes referencing concrete classes (but only interfaces)
- No inheritance from concrete classes
  - But only from interfaces
  - … it is not our friend …
- No overriding of implemented methods (but only of virtual ones)

92

Is it abstract coupling, still?

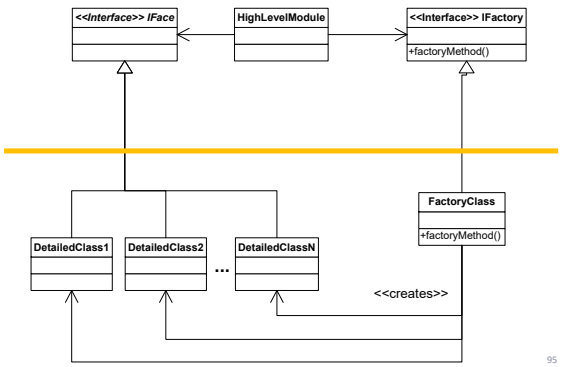


93

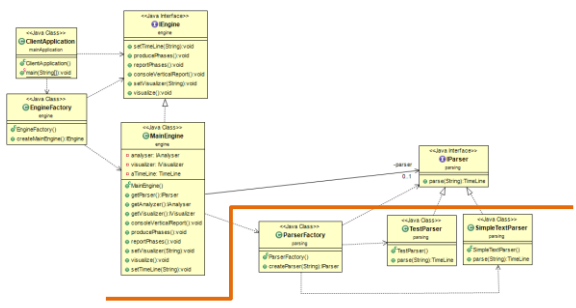
Λύσαμε όλα τα προβλήματα?

- Όχι: όλα είναι καλά εκτός από:
  - Το ρίσκο του να χρειαστεί να αλλάξει το υψηλό επίπεδο (που συμβαίνει, εξ' ου και θα χρειαστεί συντήρηση το λογισμικό)
  - Την εξάρτηση από το χαμηλό επίπεδο, στην κατασκευή αντικειμένων
    - Όπου γίνεται: factories
  - «Μα δεν είναι πολύ αυστηρές αυτές οι οδηγίες?»
    - Είναι – και μπορείτε να τις παραβιάσετε όπου αναμένεται να μην υπάρχει εξέλιξη, ή, αν είστε διατεθειμένοι να πληρώσετε το τίμημα της παραβίασης του OCP όταν συντηρήσετε τα σημεία παραβίασης

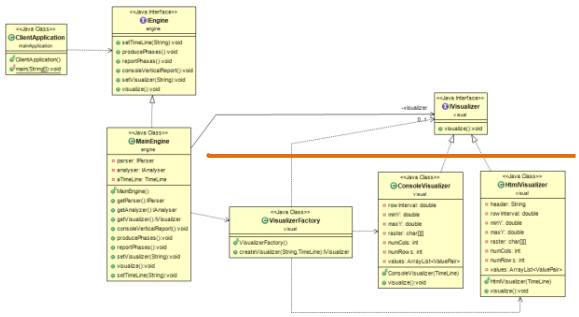
Εργοστασιακό DIP



Ένα παράδειγμα



Η ίδια συνταγή



97

Κρατήστε μόνο τις βασικές αρχές

LISKOV SUBSTITUTION PRINCIPLE

98

Liskov Substitution Principle (LSP)

- Η Barbara Liskov, 1988 έδωσε τις βασικές αρχές «αντικατάστασης» (διάβαζε: χρήσης στη θέση) αντικειμένων της βασικής κλάσης από αντικείμενα των παραγόμενων από αυτήν κλάσεων)
- Σε απόδοση από τον Uncle Bob ο κανόνας λέει:

Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it

99

Πολυμορφισμός

Έστω ότι η μέθοδος ενός client / engine έχει ορισθεί ως

```
Client::myMethod(MamaClass m){
   m.doStuff()
}
/*άρα χρησιμοποιεί ένα αντικείμενο της κλάσης MamaClass το οποίο καλεί τη μέθοδο
   doStuff() */
```

και υπάρχει η κλάση παιδί Child που κάνει override τη doStuff()

και στον κώδικα του client έχω

```
   Child c = new Child();
   myMethod(c);
```

τότε η συνάρτηση που θα κληθεί μέσα στη myMethod είναι η Child::doStuff()

Πολυμορφισμός

- «Μα καλά, το LSP δεν είναι η ουσία του πολυμορφισμού όπως υποστηρίζεται εγγενώς από τις ΟΟ γλώσσες προγραμματισμού?»
  - Αν μείνουμε στην παραπάνω διατύπωση αυστηρά, ναι, εκ γενετής, στις γλώσσες που ξέρουμε, το LSP ισχύει (όμως είναι πιο σύνθετο από αυτό)
- «Τόσο καιρό που συζητάμε το abstract coupling, αυτό δεν κάνουμε?»
  - Η ουσία του abstract coupling στηρίζεται στη βασική ιδιότητα του πολυμορφισμού, να μπορούμε να καλέσουμε τη σωστή συνάρτηση από ένα αντικείμενο της παραγόμενης κλάσης, χωρίς να απασχολείται ο προγραμματιστής με αυτό

Η ουσία του LSP

- Δεν αρκεί η κλάση Child να έχει ίδια δομή με την κλάση Mama
- Δεν αρκεί να έχουν την ίδια υπογραφή για τις μεθόδους
  - ATTN: if they do, and child inherits mama, polymorphic overriding and abstract coupling work just fine
- **Πρέπει επιπλέον, και η συμπεριφορά των μεθόδων, σε σχέση με το τι αναμένει να δει/λάβει/… ο client κώδικας, να είναι αντικαταστάσιμη**

Squares and Rectangles

- Rectangle:
  - Width, height
  - setWidth(double x), setHeight(double x)
- Είναι ένα Square ένα Rectangle?
  - Width, height: OK
    - with the extra constraint that width = height always
  - setWidth(double x), setHeight(double x): OK too
    - ATTN: both functions in their code contain
    - width=x; height=x;

103

What if a client writes

```
public void doTheTrick(Rectangle r){
  r.setHeight(4);
  r.setWidth(5);
  assert((t.getWidth()*t.getHeight()) =  20);
}
```

Με άλλα λόγια, ένα κώδικας που αναμένει να ΜΗΝ υπάρχει το constraint height=width, αλλά το constraint area = 20

104

WWW (what went wrong)

- Η συμπεριφορά του Square ΔΕΝ είναι ίδια με τη συμπεριφορά του Rectangle σε σχέση
  - Με το τι γίνεται εσωτερικά στον κώδικα
  - Με το τι λογικοί περιορισμοί υπάρχουν για τις μεθόδους
  - … και τελικά, με το τι αναμένει ο client code
- … ασχέτως του ότι όλα τα άλλα κριτήρια πληρούνται για να μπορείς να αντικαταστήσεις το Rectangle με ένα Square

105

Συμβολαιογραφείο (Design by Contract)

- Ο B. Meyer (1988) έβαλε τις μεθόδους να έχουν preconditions && postconditions
- You can substitute an object Mm of a mama class with an object Mc of a child class if the method used by Mc:
  - has a weaker precondition than the one of Mm
    - i.e., if your client code would start working with Mm, it will also start working with Mc
  - has a stronger postcondition than the one of Mm
    - i.e., once done, all the constraints that the execution with Mm would respect, will also be respected with Mc

106

...οπότε...

- Αν η postcondition for Rectangle.setWidth() says:
  - width= x and height = old.height
- Και η αντίστοιχη postcondition for Square.setWidth() says:
  - width= x and height = x
- Then, squares cannot substitute rectangles

107

HANDLE WITH EXTREME CARE!!!

SINGLE RESPONSIBILITY PRINCIPLE (SRP)

108

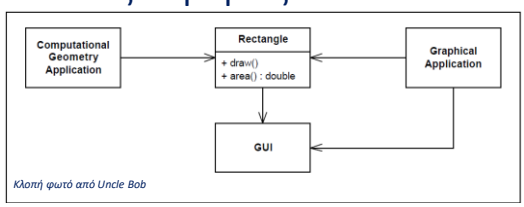
…προτού προχωρήσουμε παρακάτω…

- Το SRP είναι ένα εξαιρετικά απλό και ταυτοχρόνως εξαιρετικά επικίνδυνο principle
- ΜΗΝ μπείτε στον πειρασμό να το εφαρμόσετε παντού άκριτα
- … Κρατήστε την ουσία όμως …

Single Responsibility Principle (SRP)

- A class should have no more than one reason to change
- Tom deMarco, 1979, στο βιβλίο του για Structured Analysis & System Specification
- ΙΜΗΟ άλλο ένα κακό όνομα στο χώρο της πληροφορικής: όπως λέει και η ουσία του νόμου, αυτό που είναι single είναι the reason to change (ενώ ο τίτλος μιλά για responsibility).
- However, this is a fairly good approximation…

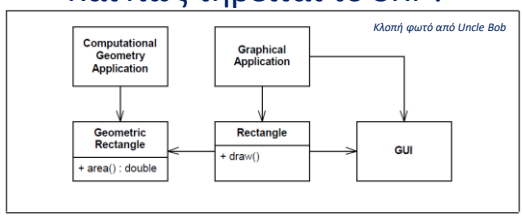
Πώς παραβιάζεται το SRP?



Κλοπή φωτό από Uncle Bob

- Όταν μια κλάση έχει παραπάνω από μία αρμοδιότητες, συνήθως έχει και παραπάνω από ένα λόγο να αλλάξει
- Εδώ έχουμε μια κλάση που κάνει και back-stage δουλειά (υπολογίζει εμβαδόν για μια εφαρμογή υπολογιστικής γεωμετρίας) και front-end δουλειά για το graphical app.

Και πώς τηρείται το SRP?



- Τηρείται όταν κάθε κλάση έχει μόνο ένα λόγο να αλλάξει
- Εδώ: αλλαγές λόγω back-end λειτουργίας στην GeoRectangle και λόγω front-end στην (σκέτο ☹) Rectangle

To probe further …

- Στα μαθήματα της Τεχν. Λογισμικού και Αντικειμενοστρεφούς Σχεδίασης, όταν θα έχετε και πιο πολλά χιλιόμετρα κώδικα στα χέρια σας, θα μάθετε
  - Πιο πολλά για το SRP
  - Καθώς και για cohesion metrics (LCOM 1,2,3)

ΔΕΝ ΞΕΧΝΩ:
responsibility is actually a “reason to change” ☺

Μιας και σπάμε τις κλάσεις, μήπως να σπάσουμε και τα interfaces?

INTERFACE SEGREGATION

Interface Segregation Principle (ISP)

- Clients should not be forced to depend upon interfaces they do not use
  - Robert Martin (Uncle Bob), 2003
  - Μετάφραση: αν μια κλάση βγάζει προς τα έξω ένα σχετικά ευμέγεθες ("fat") interface και οι clients της χρησιμοποιούν μόνο ένα μέρος του, τότε κάθε client εκτίθεται (και άρα χρειάζεται να ελεγχθεί για πιθανή συντήρηση) σε περισσότερες αλλαγές της service provider κλάσης απ' όσες του αναλογούν στ' αλήθεια

115

Interface Segregation Principle (ISP)

- Avoid "fat" interfaces that do everything => THINK SMALL!!
- Many special purpose interfaces are probably better than general-purpose interfaces
  - As they prohibit worn/unwanted usage of interfaces from client programmers
  - As they immunize clients from changes that do not concern them

116

Ιδεατά

- Κάθε client «βλέπει» ένα interface με ακριβώς τις μεθόδους που χρησιμοποιεί και μόνο
- Αυτό σημαίνει ότι μόνο αλλαγές στο interface (ή/και στις κλάσεις που το υλοποιούν) θα περάσουν στον client προς συντήρησή του
- … και ότι ο client developer θα έχει ένα μικρό συνεκτικό σύνολο μεθόδων για να κάνει τη δουλειά του (και όχι ότι να 'ναι)

117

Θυμηθείτε

- Ο σωστός τρόπος εξέλιξης είναι
  - High-level modules own interfaces
  - High-level modules dictate change, due to updates in the user requirements
  - Low-level modules should adapt to the above
  - Interfaces are the gateways between high and low level modules
- Δεν είναι βολικό να έχω πολλούς high-level modules να χτυπούν το ίδιο interface – αν και κάποιες φορές, για λόγους επαναχρησιμοποίησης του κώδικα, δεν μπορώ να το αποφύγω

Παράδειγμα από Uncle Bob

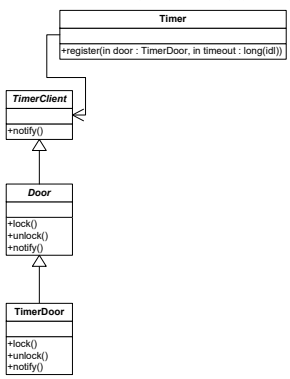
```
public abstract class Door {
   public abstract void Lock();
   public abstract void Unlock();
   public abstract bool IsDoorOpen();
};
public abstract class TimerClient {
   public abstract void Notify();
};
```

Έστω δύο interfaces/abstract classes που χαρακτηρίζουν (α) πόρτες και (β) αντικείμενα με χρονοδιακόπτες. Οι δύο αυτές αφηρημένες κλάσεις έχουν διαφορετικούς clients.

Για παράδειγμα, τα αντικείμενα μιας κλάσης Timers χρησιμοποιούν την TimerClient για να κάνουν register σε ένα timer ο οποίος θα τους ειδοποιεί όταν περάσει η ώρα

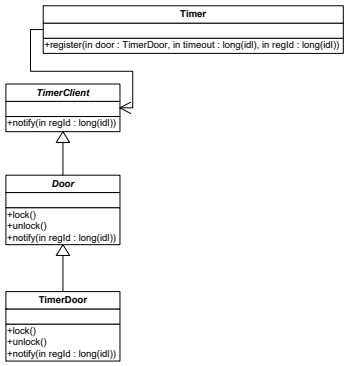
```
public class Timer {
   public void Register(int timeout, TimerClient client){ … }
};
```


Παράδειγμα από Uncle Bob



- Πώς υλοποιούμε μια πόρτα με χρονοδιακόπτη, ώστε (α) να ανοιγοκλείνει σε Door clients και (β) να λέει την ώρα σε Timer clients?
- Θυμηθείτε: η ιεραρχίες ΔΕΝ είναι φίλοι μας!

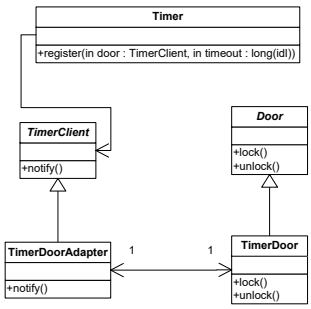
Interface Pollution



Τι γίνεται άμα η πόρτα ανοιγοκλείνει και πρέπει να βάλει πολλά timeout registrations?

«Μολύνουμε» το interface με ένα ακόμα πεδίο στη notify?

Delegation: use composition instead of aggregation & distribute clients



Όσοι clients θέλουν να δουλέψουν με τον Timer βλέπουν μόνο αυτόν
Όσοι θέλουν την Door, μόνο αυτή
Μαζί: μέσω σύνθεσης και όχι κληρονομικότητας!!

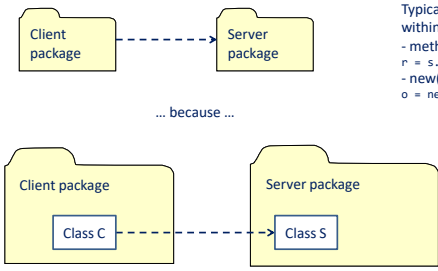
Ηθικό δίδαγμα από ISP

- Clients should not be forced to depend upon interfaces they do not use
- Aggregation and NOT inheritance handles complexity!
- When interfaces become fat, or when you need to combine functionality
  - Split classes and aggregate objects
  - Avoid polluting interfaces
- Δεν πας πειράζει να έχουμε πιο πολλά interfaces, αν αυτό είναι το τίμημα της μειωμένης εξάρτησης
- Πάντα να σκέφτεστε μήπως αντί για ιεραρχίες μπορεί να κάνει τη δουλειά σας η συνάθροιση!

Εκτός από το πώς οργανώνουμε τις κλάσεις και τα interfaces, είναι πολύ σημαντικό το πώς οργανώνονται και τα πακέτα, καθώς αυτό «παράγει», σε μεγάλο βαθμό, την αρχιτεκτονική του συστήματος

PACKAGE PRINCIPLES

Package Dependency



Dependency: whenever changing the public i-face of a service provider class affects a class that uses it, aka "client".
Typically due to calls within a client's code
- method calls
`r = s.getServiceFromS(…)`
- new() calls
`o = new S(…)`

Package principles

- Cohesion within a package
  - Release Reuse Equivalence (REP)
  - **Common Closure Principle (CCP)**
  - Common Reuse Principle (CReP)
- Coupling between packages
  - **Acyclic Dependency Principle (ADP)**
  - **Stable Dependency Principle (SDP)**
  - **Stable Abstraction Principle (SAP)**

Release Reuse Equivalence (REP)

- The granule of reuse is the granule of release
- How: when you release, release entire packages.
  - This makes it easier both for your clients and for the version management system.
- Why?
  - Take if from Uncle Bob: "I reuse code if, and only if, I never need to look at the source code … I expect the code I am reusing to be treated like a product. It is not maintained by me. It is not distributed by me. I am the customer, and the author, or some other entity, is responsible for maintaining it. When the libraries that I am reusing are changed by the author, I need to be notified. Moreover, I may decide to use the old version of the library for a time. …"
- Robert Martin, "Granularity", at
  - http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod
  - Also: https://sites.google.com/site/unclebobconsultingllc/home/articles (search for Granularity)

127

Common Closure Principle (CCP)

- **Place classes that change together in the same package!**
- **Why**? Because **you localize maintenance**! (esp., if packages are "tied" to specific developers)
- **How? Package together classes that …**
  - **… are tightly coupled (structurally and semantically)!**
  - **… you know will change together! (remember OCP?)**
- Not possible always. In fact, it is unavoidable that you will have cross-package dependencies (or else, sth is wrong with your design). However: try to minimize cross-package dependencies!

128

Common Reuse Principle (CReP)

- Do not group together (i.e., in the same package) classes that are not reused together (by other classes)
- Based on the principle that if you depend on one class, you depend on its entire package (changes in the package might implicitly impact the class you depend upon). This suggests that ideally, you use all the classes of a package.
- PV: over-restrictive && too hard to attain @early stages of development!
- CCP is based on maintenance, CReP on reuse. Antagonistic to each other!
- PV: use it only if you know what you are doing!

129

Tensions between packages

- **CCP vs. REP and CReP**: in extremis, CCP would produce a single, large package (it aims at making larger packages) and CReP a large number of very small packages. A balance is needed…
- **CCP tries to make the life of the maintainer easier. So it is guiding package architecture, esp., at the beginning of the project**, when architecture is still fluid.
- **REP+CReP** try to make the life of the client of a package ("reuser") easier. **At later stages of a project's lifecycle**, **once the architecture is stable**, and the subsystems coarsely fixed, and **there are subsystems that can be reused** (esp., by other projects too), they start to influence design decisions more…

130

Package principles

- Cohesion within a package
  - Release Reuse Equivalence (REP)
  - **Common Closure Principle (CCP)**
  - Common Reuse Principle (CReP)
- Coupling between packages
  - **Acyclic Dependency Principle (ADP)**
  - **Stable Dependency Principle (SDP)**
  - **Stable Abstraction Principle (SAP)**

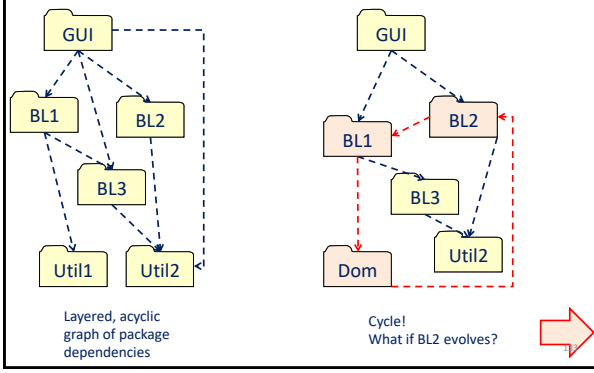
131

Acyclic Dependency Principle (ADP)

- **The dependencies between packages must form no cycles!**
- Packages should form a Directed Acyclic Graph
- **Why?** Even if the packages are cohesive => at maintenance, changes will be focused in few places, still, the entire configuration needs to be tested at the end of the day
- Cyclic dependencies hide (a) cycles in maintenance and testing without observable end of the propagation of the maintenance and (b) too tight coupling among packages
- **How to achieve?** Break cycles at first sight, either (a) by merging packages, or (b)by factoring out the common part that causes the cycle

132

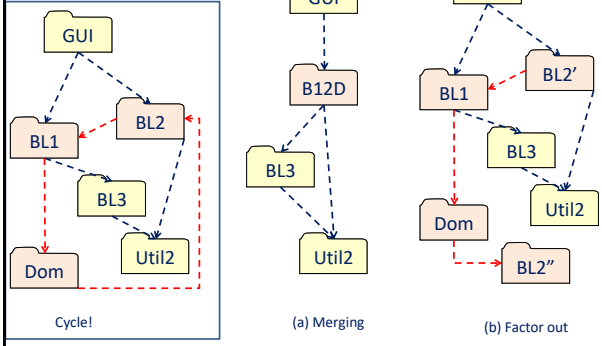
The good, the bad and …



Layered, acyclic graph of package dependencies

Cycle!
What if BL2 evolves?

… the fixes



Cycle!

(a) Merging

(b) Factor out

134

Stable Dependency Principle (SDP)

- **Depend in the direction of stability!**
- Packages likely to change, should have few incoming edges (should not be depended upon a lot)
- Packages that are considered stable can tolerate many incoming dependencies, and hopefully, they depend less upon others
- **The lower you go in the layered diagram, the more stable you are and the less outgoing edges you have**
- Remember: an acyclic package graph implies a layered design with a single direction of the dependencies

135

Stable Dependency Principle

- What is stability?
  - Not rigidity: stability is NOT the improbability of change
  - **Stability of a package assesses the amount of work needed to make a change to the package**
- How to assess stability?
  - Quality of code: complexity, size, clarity, …
  - #incoming edges: the more incoming edges a package has, the more stable it is (a small change propagates to many other packages).
  - Degree of dependency upon others: the more you depend, the more unstable you are

Stable Dependencies Principle

- Ca = afferent coupling = incoming edges from packages that depend upon you (ελκτικός, κεντρομόλος), measures how much others depend on you
- Ce = efferent coupling = outgoing edges to packages upon which you depend (εξερχόμενος, φυγόκεντρος), measures how much you depend upon others
- Instability I = Ce / (Ca+Ce)
- I=1 means nobody depends upon you (fan-In ->0)
- I=0 means you are independent (no outgoing, fan-Out->0)

Stable Dependencies Principle

- Why? Coupled with common closure (cohesive packages) & acyclic graph it guarantees a small propagation of changes, w/o cycles, and smaller chances of propagating outside a package
- We **want** the system to be able to change! We **want** the system to evolve!
  - Lehman's laws insist that w/o (ability to easily) change, a system dies
- How? All that we have learnt in this class…

Stable Abstractions Principle (SAP)

- **Stable packages should be abstract packages**
- The lower in the hierarchy, the less propagation a change should have
- **Use abstract classes and interfaces as the elements of lower level packages upon which higher levels depend!**
- **Use OCP&&DIP to extend them (instead of updating them), whenever evolution is needed**
- **Do not depend upon concrete classes!**
- **Use abstract coupling!**

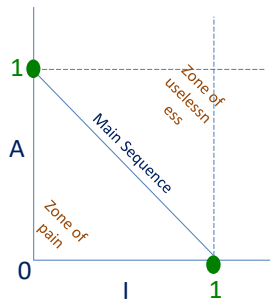
139

Stable Abstractions Principle (SAP)

- … and a metric of abstractedness …
- Nc = #classes+interfaces of a package
- Na = #abstr. classes+interfaces of a package
- A = Na / Nc
- Ideally for a package: either A=0 (super concrete package) or 1 (just a contract)
- Bad: be close to 0.5…
- Remember also instability: I=1 means nobody depends upon you, I=0 means you are independent (SEE NEXT!)

140

Uncle Bob's AI graph: ideally, as *A* decreases, *I* should increase

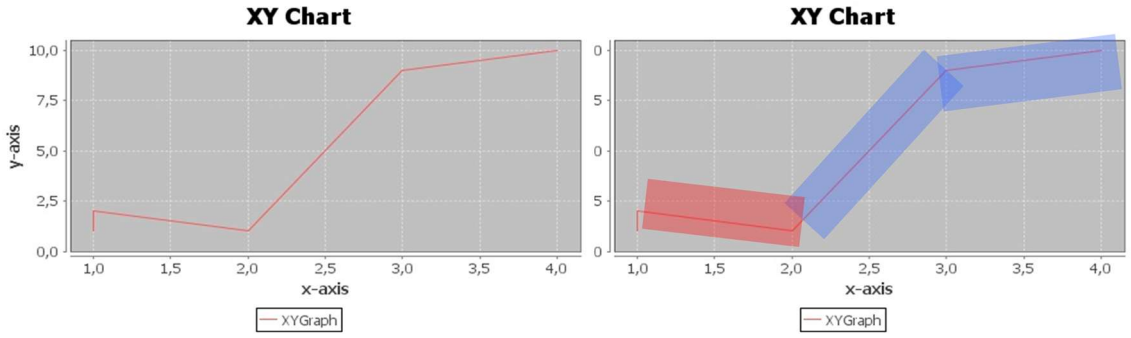


- A=1, I=0: super abstract, super independent ☺
- A=0, I=1: super concrete, nobody depends on it ☺
- Zone of pain: both concrete and very high pct of incoming dependencies ☹
- Zone of uselessness: only abstract and too dependent upon others ☹
- **Live in the line of the main sequence, as close to the two extremes as possible!**

141

ΣΗΜΕΙΩΣΕΙΣ

Έχουμε ένα project στο οποίο πρέπει να φτιάξουμε μια μηχανή η οποία να λαμβάνει ως είσοδο μια χρονοσειρά και να εξάγει φάσεις της χρονοσειράς. Μια χρονοσειρά είναι μια λίστα από σημεία μέτρησης, με κάθε σημείο να είναι ένα ζεύγος (α/α, μέτρηση). Μια φάση είναι ένα σύνολο συνεχόμενων σημείων με ίδια τάση. Στο παρακάτω παράδειγμα, φαίνεται η βασική ιδέα: κάθε σημείο έχει απεικονισθεί σε ένα διδιάστατο διάγραμμα, με τον α/α να απεικονίζεται στον οριζόντιο και τη μέτρηση στον κάθετο άξονα (αριστερό διάγραμμα) και οργανώνεται σε μια κόκκινη, πτωτική φάση και σε μια μπλε, αύξουσα φάση.



Το σύστημά μας θέλουμε να υποστηρίζει τις εξής λειτουργίες:

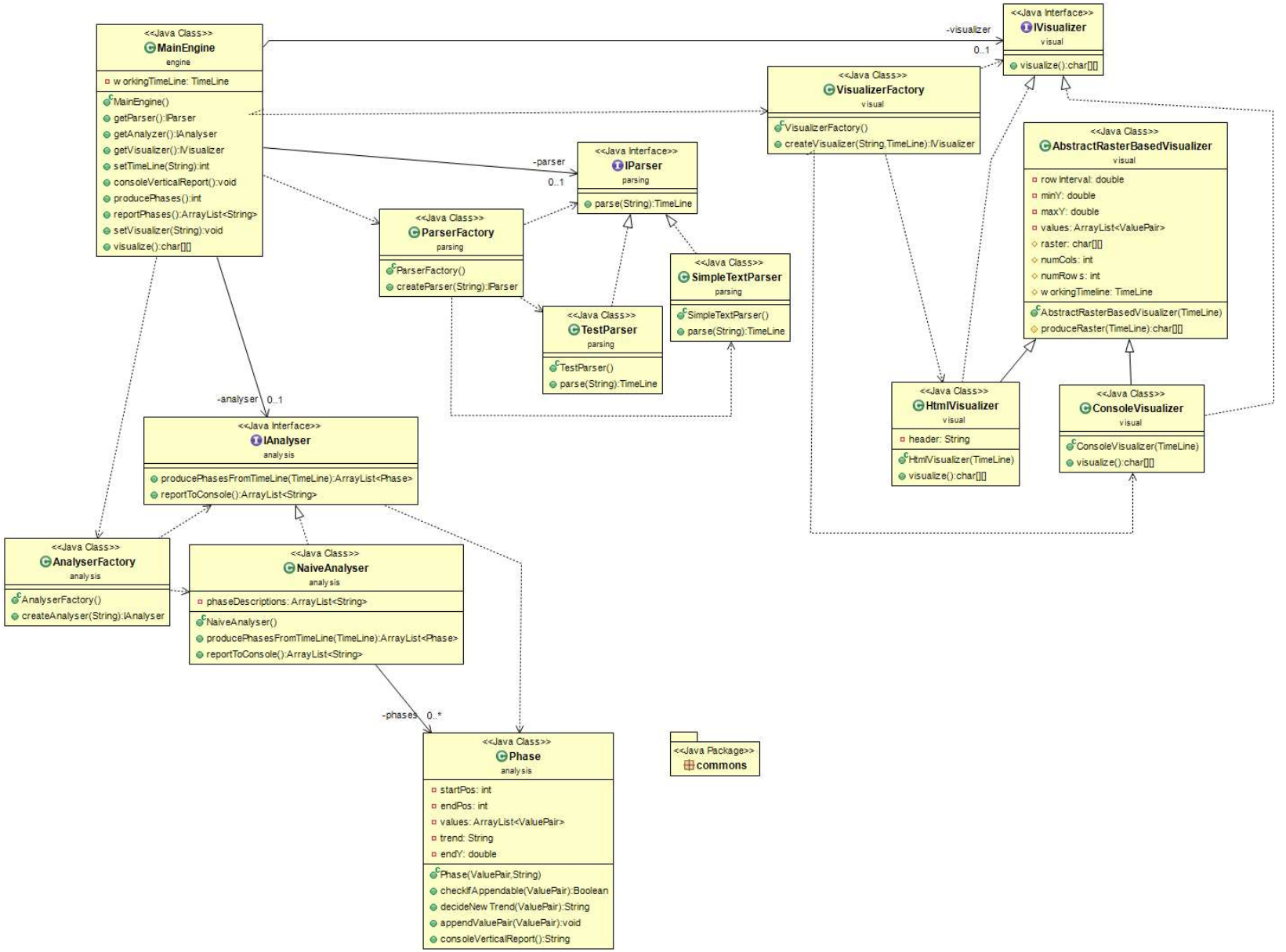
- Φόρτωση δεδομένων με επεκτάσιμο τρόπο (και συγκεκριμένα, εδώ, τουλάχιστον με ένα απλό φόρτωμα από μια μέθοδο, και με επεξεργασία από αρχείο). Στην πρώτη περίπτωση απαιτείται μια απλή μέθοδος που θα κατασκευάζει και θα αναπαριστά μια μικρή χρονοσειρά (κυρίως για να χρησιμοποιηθεί σε περιπτώσεις ελέγχων). Στη δεύτερη περίπτωση, απαιτείται ο χρήστης να προσδιορίσει ένα αρχείο εισόδου από το οποίο τα δεδομένα θα φορτωθούν στο σύστημα και θα αναπαρασταθούν. Το αρχείο εισόδου θεωρούμε ότι έχει μια μέτρηση με 3 στοιχεία: <α/α><χρονόσημο><μέτρηση> σε κάθε γραμμή. Το χρονόσημο και το <α/α> μπορεί να ταυτίζονται αν ο ανθρώπινος χρόνος δεν είναι διαθέσιμος.

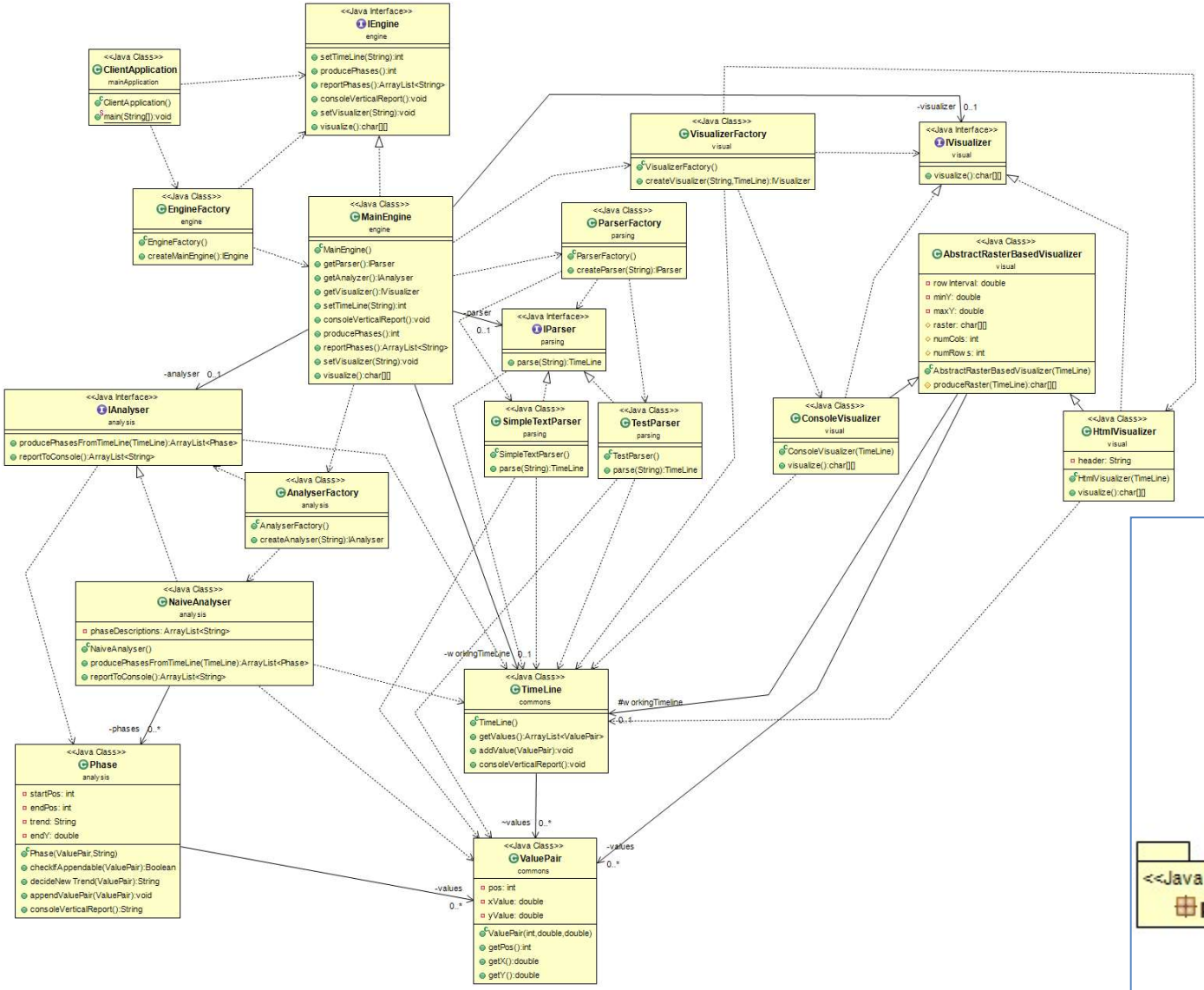
- Ανάλυση της χρονοσειράς σε φάσεις (επίσης με επεκτάσιμο τρόπο – εδώ με ένα απλό αλγόριθμο, αλλά στο μέλλον με αλγορίθμους που, π.χ., αγνοούν μικρά spikes). Ο βασικός αλγόριθμος στηρίζεται στην ιδέα ότι κάθε φάση έχει μια τάση (ανοδική, καθοδική, ή επίπεδη). Ο αλγόριθμος επεξεργάζεται ένα ένα τα σημεία της χρονοσειράς. Ας υποθέσουμε, για να καταλάβουμε τον αλγόριθμο, ότι βρισκόμαστε στη μέση της χρονοσειράς, και η τρέχουσα φάση είναι η Φ. Έστω δε ότι μόλις προσθέσαμε το σημείο στη θέση k στη χρονοσειρά και πάμε να προσθέσουμε το σημείο k+1. Αν η τάση της φάσης Φ διατηρείται με την προσθήκη του νέου σημείου, τότε το σημείο k+1 προστίθεται στην υπάρχουσα φάση. Αν όχι, τότε ξεκινάμε μια νέα φάση. Η διατήρηση της τάσεως είναι απλή: αν η τάση είναι καθοδική και η μέτρηση του σημείου k+1 είναι μικρότερη από την μέτρηση του σημείου k, τότε η τάση διατηρείται.

- Παρουσίαση της χρονοσειράς, επίσης με επεκτάσιμο τρόπο (εδώ: (i) στην κονσόλα και (ii) σε μια html σελίδα). Η παρουσίαση περιλαμβάνει την οπτικοποίηση των 2 αξόνων, X και Y, με τον άξονα X να αντιστοιχεί στο χρόνο και τον Y στην μετρούμενη ποσότητα και την παρουσίαση των σημείων μέτρησης της χρονοσειράς ως σημεία στο διδιάστατο χώρο που προκύπτει από τους 2 άξονες.

- Παρουσίαση των φάσεων που δημιουργούνται από την χρονοσειρά. Για κάθε φάση, καταγράφουμε ένα μήνυμα «Νέα φάση με τάση …» ανάλογα με την τάση, και στη συνέχεια παρατίθενται τα σημεία της φάσεως, το ένα μετά το άλλο, το καθένα στη δικιά του γραμμή.

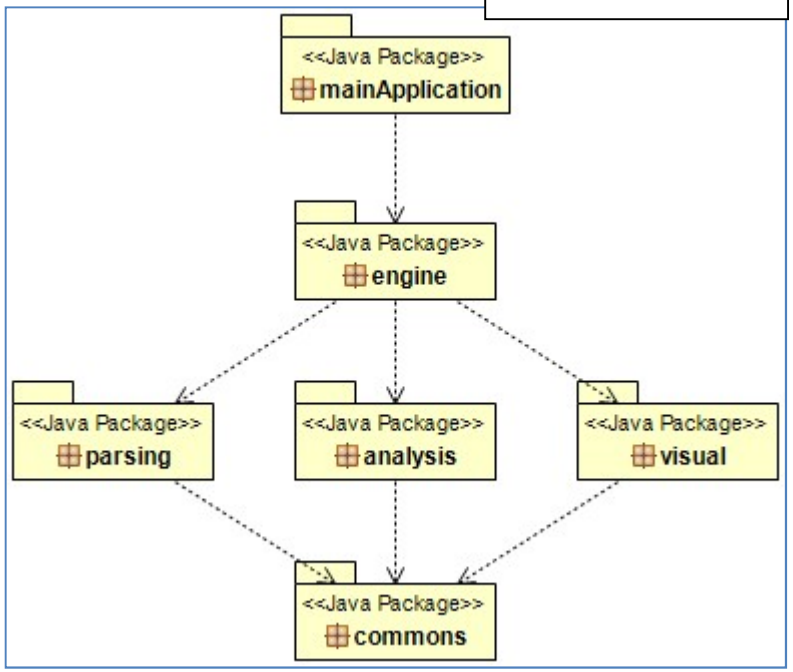
Ένα απλό διάγραμμα με ένα υποσύνολο των κλάσεων για τα βασικά στοιχεία του project

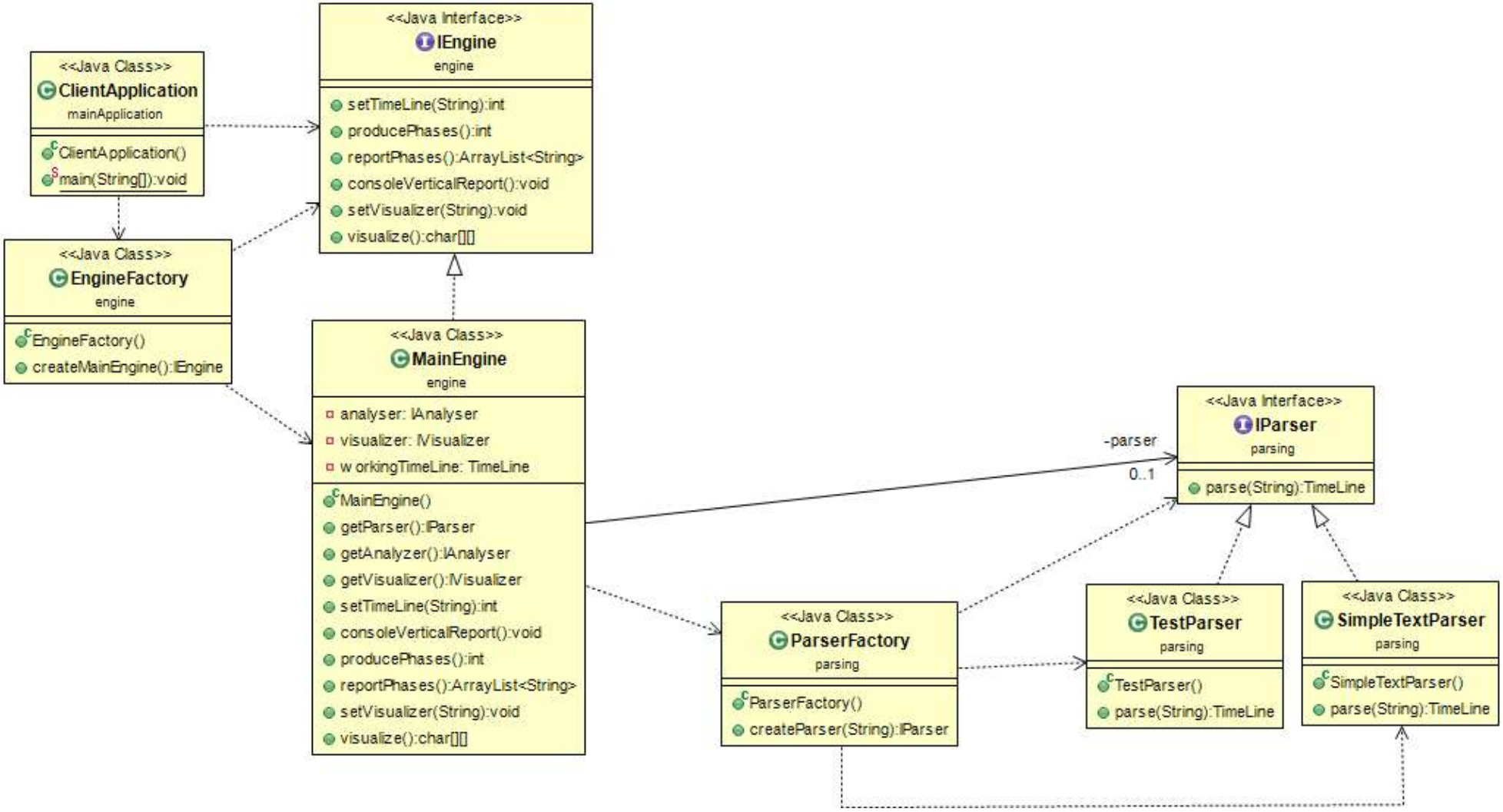




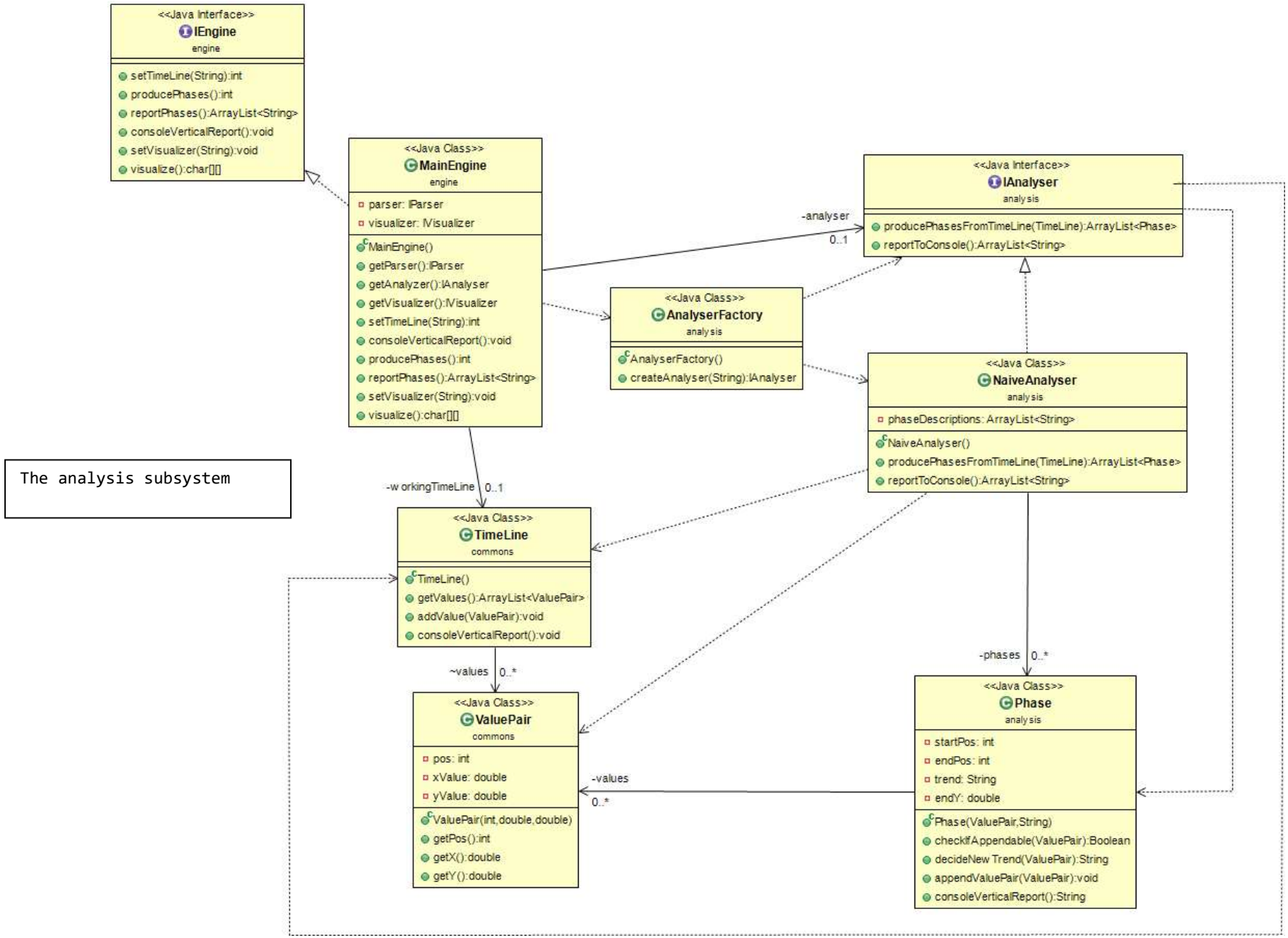
The entire class diagram

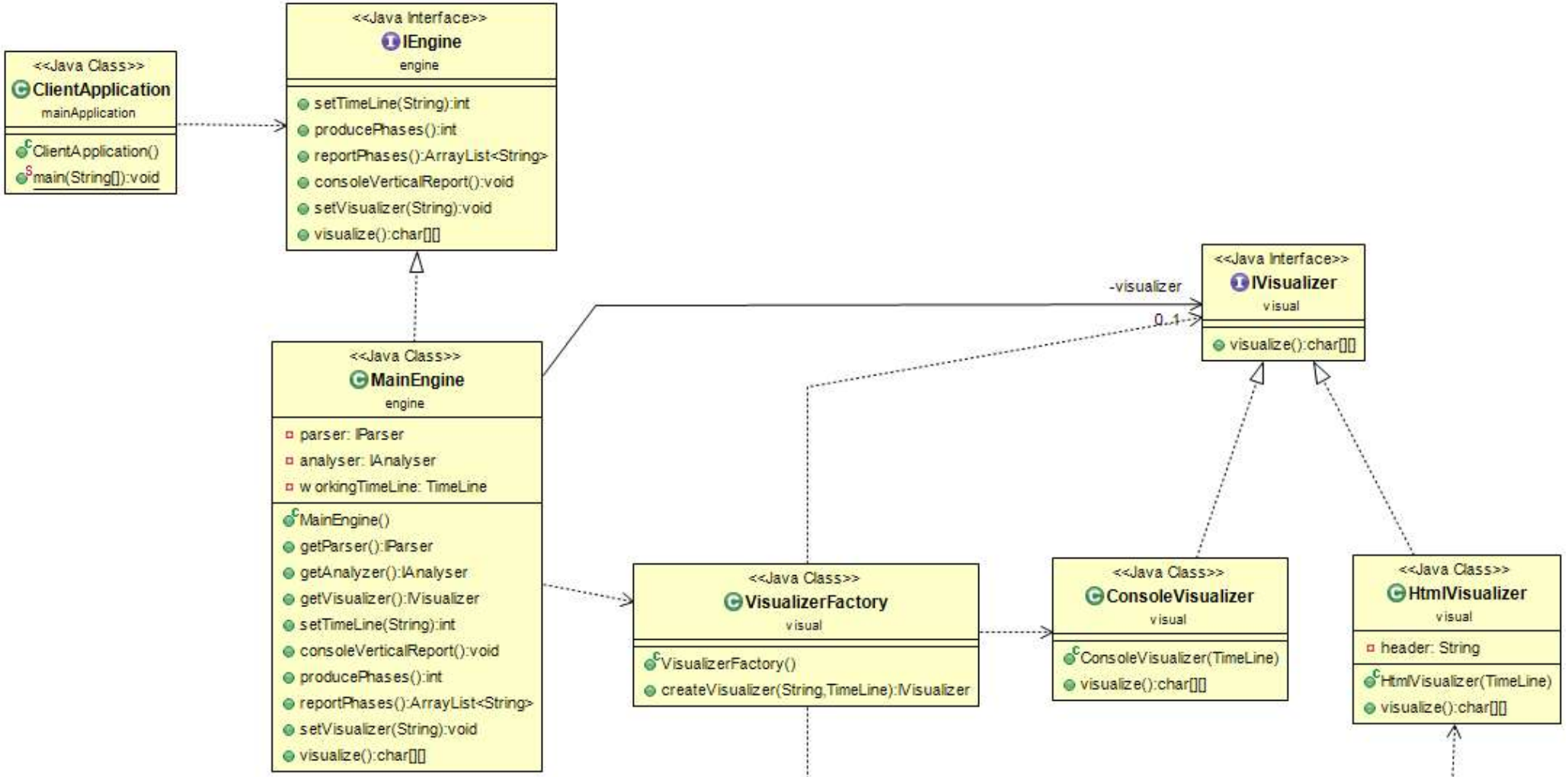
The package diagram





The parsing subsystem

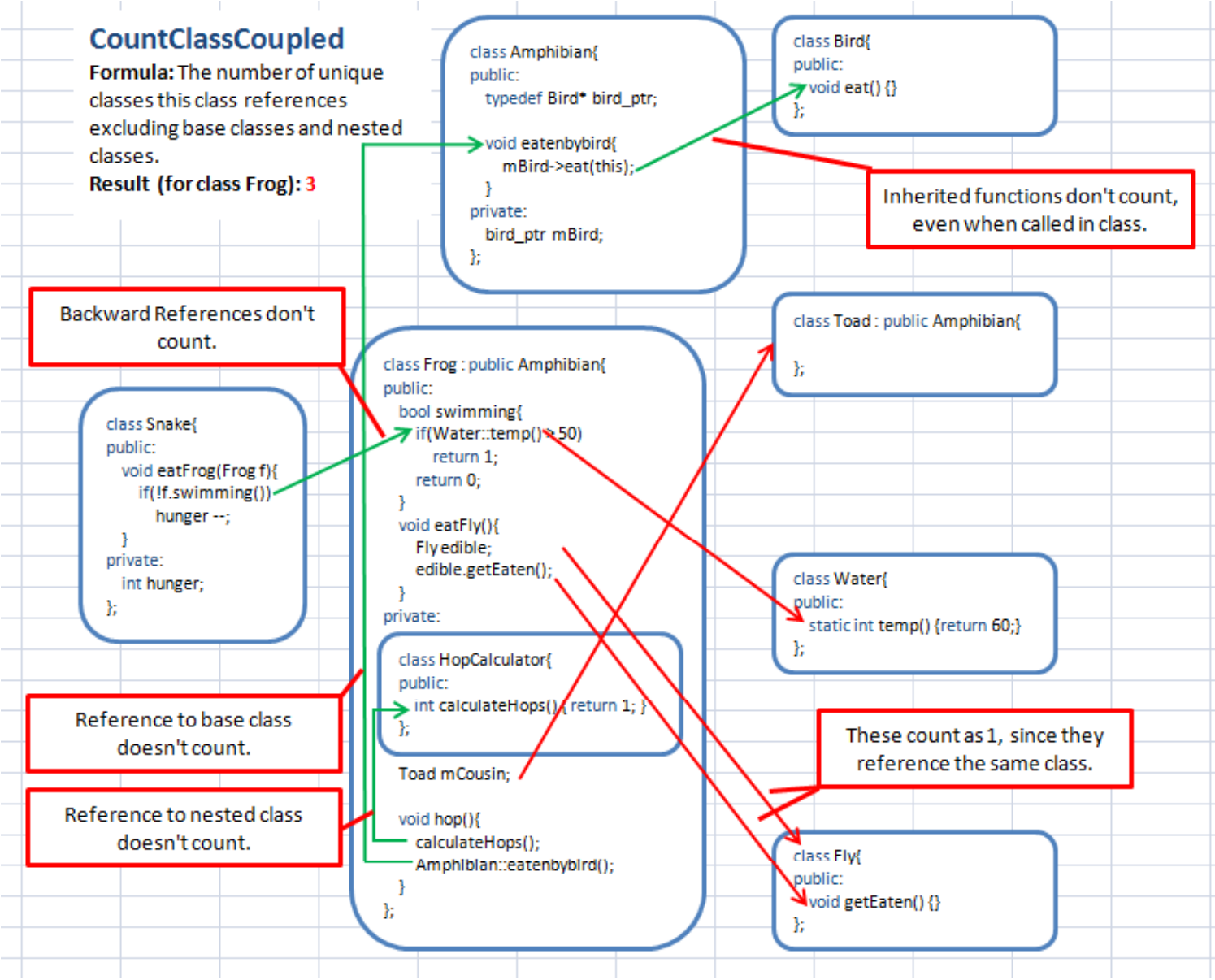

The analysis subsystem
<<Java Interface>>
IEngine
engine
setTimeLine(String):int
producePhases():int
reportPhases():ArrayList<String>
consoleVerticalReport():void
setVisualizer(String):void
visualize():char[][]
<<Java Class>>
MainEngine
engine
parser: IParser
visualizer: IVisualizer
MainEngine()
getParser():IParser
getAnalyzer():IAnalyser
getVisualizer():IVisualizer
setTimeLine(String):int
consoleVerticalReport():void
producePhases():int
reportPhases():ArrayList<String>
setVisualizer(String):void
visualize():char[][]
-analyser
0..1
<<Java Interface>>
IAnalyser
analysis
producePhasesFromTimeLine(TimeLine):ArrayList<Phase>
reportToConsole():ArrayList<String>
<<Java Class>>
AnalyserFactory
analysis
AnalyserFactory()
createAnalyser(String):IAnalyser
<<Java Class>>
NaiveAnalyser
analysis
phaseDescriptions: ArrayList<String>
NaiveAnalyser()
producePhasesFromTimeLine(TimeLine):ArrayList<Phase>
reportToConsole():ArrayList<String>
-workingTimeLine
0..1
<<Java Class>>
TimeLine
commons
TimeLine()
getValues():ArrayList<ValuePair>
addValue(ValuePair):void
consoleVerticalReport():void
~values
0..*
<<Java Class>>
ValuePair
commons
pos: int
xValue: double
yValue: double
ValuePair(int,double,double)
getPos():int
getX():double
getY():double
-values
0..*
-phases
0..*
<<Java Class>>
Phase
analysis
startPos: int
endPos: int
trend: String
endY: double
Phase(ValuePair,String)
checkIfAppendable(ValuePair):Boolean
decideNewTrend(ValuePair):String
appendValuePair(ValuePair):void
consoleVerticalReport():String



The visualization subsystem

(the abstract visualizer class omitted for clarity)

Παράδειγμα (C++) από http://www.scitools.com/documents/metricsList.php?metricGroup=oo#CountClassCoupled



Testing

Ανάπτυξη Λογισμικού (Software Development)

www.cs.uoi.gr/~pvassil/courses/sw_dev/

ΜΥΥ301/ΠΛΥ 308

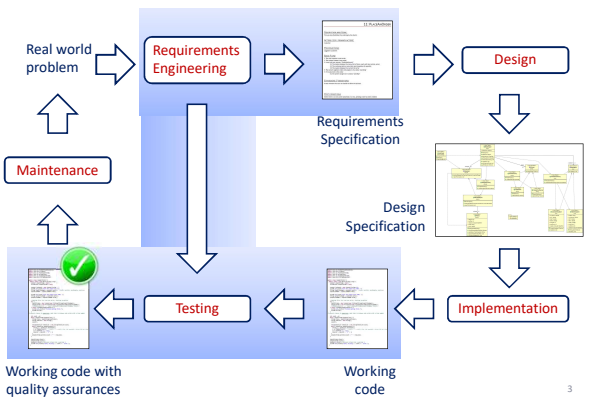
PV notes are based – to a fairly large extent – on B. Laboon's (thank you!) book
" A friendly introduction to software testing"
https://github.com/laboon/software-testing
Any errors should be attributed to PV only.

1

Έλεγχος

- [SWEBOK] Software testing consists of the *dynamic verification* that a program provides *expected behaviors* on a *finite set of test cases, suitably selected from* the usually infinite execution domain.
- Επιβεβαιώνουμε την ορθή λειτουργία του κώδικα δυναμικά (τρέχοντάς τον, δλδ)
- Επιλέγουμε ένα υποσύνολο από test cases για το σκοπό αυτό
- Ο έλεγχος συνίσταται στην αντιπαραβολή του αναμενόμενου αποτελέσματος σε σχέση με το παραγόμενο αποτέλεσμα, για κάθε test case

2



3

Always remember:
Software testing – quality assurance – is all about validating that …

the observed behavior = the expected behavior

BASIC CONCEPTS

4

Software testing is …

- Software testing is a way of providing an estimate of software quality to stakeholders (e.g., customers, users, and managers).
- Software testing is NOT …
  - … finding every single defect.
  - … randomly pressing buttons, hoping that something will break.
  - … hoping that something will break, period.
  - … something you do after all the programming is complete.
  - … something you postpone until users start complaining. (REALLY!!)

5

V&V

- Software testing involves ensuring that the right software was created, with the right internal quality
- Verification is ensuring that you're building the software right.
  - Verification is ensuring that the code respects coding standards, the design choices are reasonable, there are no outright blunders in the code, and the internal quality of the code is acceptable
  - Typically, software houses employ peer developers to perform **code review**, a **static** form of code inspection in order to ensure the code is verified.
- Validation is ensuring that you're building the right software
  - Validation is ensuring that the software does what the user wants.
  - To this end, we **dynamically test** the code in order to find whether the external behavior of the system we have built is exactly what we have agreed with the customer in the requirements
    - To a less extent, we can also check if there are any gaps in the requirements (so that even if the software does meet all the requirements, the user might not be satisfied with the product)

6

Defects

- A defect is an issue that …
  - … either **does not meet the system requirements**
  - … or breaks the functionality of a system
- Observe the role of requirements here:
  - as in all engineering projects, we need to provide guarantees on the behavior of our delivered system…
  - … and thus, we validate that the system provides EXACTLY the specified functionality
- Naturally, crashing code is a clear case of (possibly implicitly expected) requirement violation

7

Defects

- **Software testing – quality assurance – is all about validating** that …

the observed behavior = the expected behavior

8

Nobody's perfect…

- Software development is one of the most intellectually challenging human activities
- Software engineering is the discipline that imposes **principle in the development of software**, such that – much like all engineering principles – we can guarantee that the system that we produce meets **quality assurance** criteria and can be accepted as a deliverable to clients.

9

Nobody's perfect …

- Everybody's code has requirement violations
- … yours, mine, everybody's…
- There are several techniques to validate that at least certain aspects of the system meet the specified criteria, but… testing is never enough.
- However, the most important things to remember are …

10

IMPORTANT SLIDE, EVERY SINGLE WORD MATTERS!

Expand your test suite INCREMENTALLY!!

- **The goal is NOT to test everything!**
- The goal is to **cover the most important points of risk** in your code!
- We organize our test suite **to be expanded incrementally**, in a way that it is easy to expand the test suite one-test-at-a-time & run the tests fast!

DO NOT LET THE FACT THAT YOU CANNOT TEST EVERYTHING INTIMIDATE YOU FROM COVERING AS MANY REQUIREMENT VIOLATONS AS POSSIBLE!!!!

11

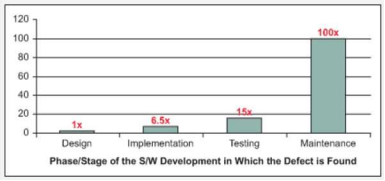
The role of a software tester

- A software tester – Quality Assurance (QA) Engineer – does not blindly test that the software meets the requirements.
- **Testers can be thought of as defenders of the user experience**, even pushing back against other internal stakeholders to develop software which meets the needs of users instead of simply meeting the bottom line.

12

Fix defects as early as possible!!

Figure 1: Relative Costs to Fix Software Defects (Source: IBM Systems Sciences Institute)



13

We start with a very simple scenario on testing, before dealing with the problem at a larger context

A FIRST EXAMPLE ON WHITE-BOX TESTING

14

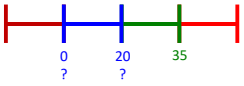
A Reference Example

- Test a new display for a **car tire air pressure sensor**, returning an integer to our software, which is responsible for activating the driver's panel
- We must ensure that our software correctly activates the driver's panel according to the following **requirement rules**:

- If (air pressure reading > 35 PSI)
  - the OVERPRESSURE light should turn on
- If (air pressure reading is in the area of 0 to 20 PSI),
  - the UNDERPRESSURE light should turn on
- If (air pressure reading < 0),
  - the ERROR light should come on

15

A Reference Example

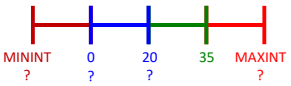


We were told:

```
If (air pressure > 35 PSI)
    the OVERPRESSURE light should
    turn on
If (air pressure reading is in the
area of 0 to 20 PSI),
    the UNDERPRESSURE light should
    turn on
If (air pressure reading < 0),
    the ERROR light should come on
```

16

A Reference Example



What is the min possible value?

Similarly for the max possible

Where does 0 belong?

Similarly for 20 and 35

What happens between 20 and 35? Oeo?

If the pressure changes, the already lit lights turn off?

Questions…

We were told:

```
If (air pressure > 35 PSI)
    the OVERPRESSURE light should
    turn on
If (air pressure reading is in the
area of 0 to 20 PSI),
    the UNDERPRESSURE light should
    turn on
If (air pressure reading < 0),
    the ERROR light should come on
```

Assume we go back to the end-user and refine req's…

17

Requirements refined

Conveniently, the sensor returns an INT, not double…

- If (air pressure in [36 .. MAXINT))
  - the OVERPRESSURE light should turn on, and,
  - all other lights should be off.
- If (air pressure in [0 … 20] PSI),
  - the UNDERPRESSURE light should turn on, and,
  - all other lights should be off.
- No light comes on for PSIs in [21 .. 35] (inclusive) - normal operating conditions
- If (air pressure in (MININT .. -1]),
  - the ERROR light should come on, and,
  - all other lights should be off.

18

Good requirements

- **Complete**. Nothing is missing
  - Here: we had an entire range missing + what happens to other lights
- **Unambiguous**. There are no terms like “stuff”, “this one”, … . Includes **quantified** too.
  - Here: the area around 0 to 20 ?? Should have been EXPLICITLY stated as [0 .. 20]
  - Better say price = 0, than ‘free’.
- **Consistent**. What if
  - UNDERPRESSURE in [0 .. 20]
  - NORMAL in [20.. 30] ?

19

We partition the range of values to equivalence classes

- An **equivalence class** (also called an equivalence partition) is one set of input values that maps to an output value.
- If no overlap of classes: **strict** partition
- If there is no spec for a partition: **undefined**

- **We had 2 cases of undefined behavior in our example:**
  - The range [21 .. 35]
  - What happens with each and every light at each equivalence class

20

Boundary & Interior values

- **Boundary values:** the “last” of one equivalence class and the “first” of a new equivalence class.
- Values which are not boundary values are called **interior values**.
- **Implicit boundary values**: due to the specificities of the system under test or the environment under which the system operates.
  - E.g., MAXINT and MININT are system dependent
  - Max size of a collection
  - Math properties: div by 0, sqrt(-1), …

21

How to test with equivalence classes

- **Ideally, all boundary values…**
- **… and a good sample of interior values**
- In our example, values to test:
  1. Interior values, ERROR: -3, -100
  2. Boundary values, ERROR / UNDERPRESSURE: -1, 0
  3. Interior values, UNDERPRESSURE: 5, 11
  4. Boundary values, UNDERPRESSURE / NORMAL: 20, 21
  5. Interior values, NORMAL: 25, 31
  6. Boundary values, NORMAL / OVERPRESSURE: 35, 36
  7. Interior values, OVERPRESSURE: 40, 95

22

Test cases

- **Base** case: a value in an equiv. class that is not near the boundary
  - **Happy path** - a case where with valid, usual input and no problems occur.
- **Edge** case: a value at the boundary, or, an unexpected case
- **Pathological** case: value completely out of the specified domain (e.g., you expect an int and you get a string or an imaginary number)

23

IMPORTANT SLIDE!

Why important?

- **ALWAYS test the happy path of a use case!**
  - Even if you don't test anything else, this is the minimum testing you need

- Boundaries (=> edge cases) are prone to creating problems because of the if(), <, <=, … typically involved
- Catastrophic cases: things can always go wrong with the input
  - invalid input
  - null pointers

24

For you. Case #1

- Bus ticket prices:
  - For children under 2: free
  - For children older than 2 but under 18: 1$
  - For senior citizens, 65 or older: 1$
  - All else: 2$
- Can you spot problems?

25

HIDDEN Case #1

- Bus ticket prices, assuming age is double:
  - For person.age in [0..2] : 0$
  - For person.age in (2..18) : 1$
  - For person.age in [18, 65): 2$
  - For person.age in [65, INF]: 1$
- All is quantified, complete, consistent, unambigous

26

For you. Case #2

- Pizza discounts at the KΨM
- Undergraduate students: 20% discount
- Grad students: 30%
- TA's: 10%
- TA is orthogonal to grad/ugrad; can also be non-student personnel
- Grad/ugrad are mutually exclusive
- Discount is additive (e.g., ugrad TA gets 20% + 10% = 30% discount)

27

For you. Case #3

- The shopping cart. An online store provides each registered customer with a shopping cart. The customer can
  - Add a number of items to the cart (e.g., 3 t-shirts)
  - Remove a number of items from the cart
- You can always add a new item at a cart, at any integer quantity greater than zero
- If you want to add an item at a cart that already exists, the respective qty must be increased
- If you want to remove a quantity of an item from the cart, (a) the item must exist in your cart, (b) the removed quantity must be <= the qty that already exists
- Before and after every operation, the list of items and their qty's are displayed to the user

28

Always remember:

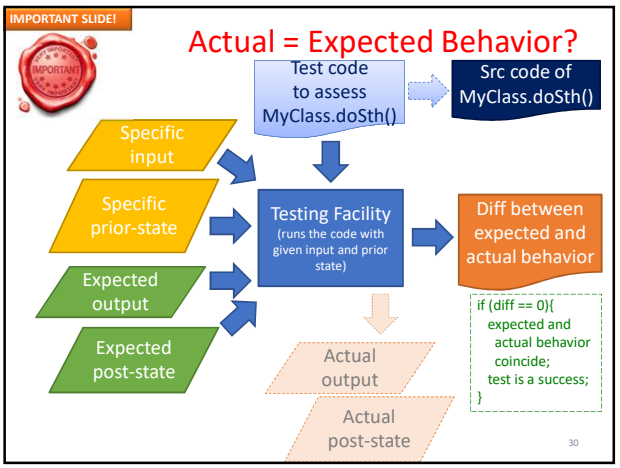
Software testing – quality assurance – is all about validating that …

the observed behavior = the expected behavior

BASIC CONCEPTS RELOADED

29

IMPORTANT SLIDE!



30

Test execution

- When we run a test, there are several potential outcomes. The most typically encountered are:
  - **Test passed**: the test execution produces a pair { output + post-conditions } which is identical to the respective expected one
  - **Test failed**: at least one part of the actual behavior differs from the respective expected one
  - **Test crashed (a.k.a. error)**: the test did not complete normally, typically without producing an actual output.

31

Yes, but how to define which tests? (coming soon)

- **Black-box** testing: tester is agnostic to the internals of the source code, just based on the relationship of input – expected – actual output
- **White-box** testing: tester is based on source code structure, trying to cover the most important execution paths of the source code
  - **Code Review** (executed by peer developers) is NOT white-box testing: code review is a static inspection of the code, to uncover obvious defects, possibly in the style and structure of the code; testing is a dynamic execution of the system, where the actual behavior of the system is compared to the expected one

32

Happy Path Reloaded

- The happy path / happy day scenario is:
  - For system tests, concerning use cases: the primary flow = zero-problem execution of a use case
  - For simple tests, e.g., unit tests of specific methods: the case where input is valid, no problems occur and the execution terminates successfully
- ALWAYS TEST THE HAPPY PATH!

33

PLANNING TESTS: STRUCTURING TEST CASES

34

Test plan

- A **test plan** is a collection of test cases, aimed towards ensure that a certain requirement is met
- A **test case** is a combination of
  - Code to be tested
  - Input values
  - Expected output
  - Pre- and post- conditions
- For every test plan we have several test cases
- We also have a principled, structure way of producing test plans (see next)

35

Elements of a test case

- **Identifier**: An identifier, such as "T1_V1", "16", "DB-7", or "DATABASE-TABLE-DROP-TEST", which **uniquely** identifies the test case.
- **Test Case Description:** a **structured** description of the test case and what it is testing.
- **Preconditions**: Any preconditions for the **state** of the system or world before the test begins.
- **Input Values**: Any **values that we input** directly to the test.
- **Expected Output Values**: Any **values directly generated as output** by the execution of the test case.
- **Postconditions**: Any postconditions of the **state** of the system or world which should hold true after the test has been executed.
- **Method(s) to test**: the method(s) whose output is to be asserted for correctness (if known at the design of the test – else omit)
- Other:
  - Execution steps. Detail execution steps if need
  - Comments.

36

Id
Description
Preconditions
Input
Postconditions
Output

Identifier

- A unique string to uniquely characterize a test case among its peers. In the context of this course we will introduce two possible versions:
- A **short** version, denoting the family of similar test cases to which it belongs + a unique identifier within the family
- A **long** version, appending a short textual description

Identifier

Assume you have two use cases:
- UC**1**: executePurchase
- UC**2**: registerNewUser

Now we want to create a test plan that includes use cases for them.
We will test the happy day case for each of them, plus **V**ariants with invalid input.
We assign the following Identifiers
- SHORT: We rank each family of test cases with a family id that corresponds to the respective use case (first part of the short id) and an increasing integer for the variant (second part of the short id).
- LONG: we append a string with the description of the use case and a description of the variant

Short	Long
T**1**_V0	T1_V0_executePurchase_HappyDay
T**1**_V1	T1_V1_executePurchase_WO_Login
T**1**_V2	T1_V2_executePurchase_WO_Money
T**2**_V0	T2_V0_registerNewUser_HappyDay
T**2**_V1	T2_V1_registerNewUser_invalidInput
…	…

Unless otherwise specified, all references to the use case are expected to use just the short version

This is just a naming scheme adopted for this course. In practice this can vary a lot.

Id
Description
Preconditions
Input
Postconditions
Output

Test case description: OREOS

- A sentence in the format
  - **O**N a *context/pre-existing state* of the system
  - **R**ECEIVING *a certain input/event*,
  - **E**NSURE THAT THE SYSTEM
  - **O**UTPUTS *such and such* an output,
  - **S**UCH THAT its *posterior state is such and such*

precondition
input
output
postcondition

IMPOSE the OREOS rigorous structure to your description!!!

Note that pre- and post- conditions can be empty
Pre-cond: the system is in a particular state
Post-cond: the system MUST result in a particular state

Methods & Test case description

- At the end of the day, **we end up testing methods** (not classes, not packages, … -- it all boils down to testing **methods**)
- Be it simple methods or methods pertaining to entire use cases: try to express the method's essence as an OREOS statement

40

OREOS Examples

Ensure that on-sale items can be added to the cart and will have their price automatically reduced.

T5_V1	
ON	a cart having being assigned to a customer *(absent from the spec. text)*
RECEIVING	on-sale items to be added to the cart
ENSURE	THAT THE SYSTEM
OUTPUTS	an updated cart
SUCH THAT	the cart is filled with the added items with their price automatically reduced

41

OREOS Examples

Ensure that passing a non-numeric string as input will result in the square root function throwing an InvalidNumber exception.

T123_V2	
ON	any context
RECEIVING	a non-numeric string as input
ENSURE	THAT THE SYSTEM
OUTPUTS	throwing an InvalidNumber exception.
SUCH THAT	state is intact

42

OREOS Examples

When the system detects that the internal temperature has reached 150 degrees Fahrenheit, ensure that it displays an error message and shuts down within five seconds.

T45_V99	
ON	any context
RECEIVING	a signal that the internal temperature has reached 150 degrees Fahrenheit
ENSURE	THAT THE SYSTEM
OUTPUTS	displays an error message
SUCH THAT	the system shuts down for 5”

43

OREOS Examples

Ensure that if the operating system switches time zones midway through a computation, that computation will use the original time zone when reporting results.

T6_V0	
ON	The O/S being in the context of a computation
RECEIVING	A signal of time zone switching
ENSURE	THAT THE SYSTEM
OUTPUTS	the results, having used the original time zone
SUCH THAT	state is intact

44

Id
Description
Preconditions
Input
Postconditions
Output

Input and Output

- **Identifier**: An identifier, such as “T1 TABLE-DROP-TEST”, which **uniquely**
- **Test Case Description:** a **structured** hat it is testing.

Once the description is structured, be specific on these!!

- **Preconditions**: Any preconditions for the **state** of the system or world before the test begins.
- **Input Values**: Any **values that we input** directly to the test.
- **Expected Output Values**: Any **values directly generated as output** by the execution of the test case.
- **Postconditions**: Any postconditions of the **state** of the system or world which should hold true after the test has been executed.
- **Method(s) to test**: the method(s) whose output is to be asserted for correctness (if known at the design of the test – else omit)
- Other:
  - Execution steps. Detail execution steps if need
  - Comments.

45

Input and Preconditions

- Think of …
  - Pre-condition: a STATE of the system that has to be present for the test to apply
  - Input: think of it as method input parameters, i.e., values
  - Output: think of it as method return value (although not necessarily so)
  - Post-condition: the STATE after completion
- In our minds: we can think of a state as a set of Boolean conditions that must hold!
- If a test applies anyway, no pre-condition (default value: "any context")
- If there is no effect to the state of the system, no post condition (default value: "state is intact")
- Input != Precondition (similarly for out-/post-)
  - Input is sth given by another s/w module or the user for the purpose of the tested method
  - Pre-cond is a setup of the objects/vrbl's & their states of the runtime

46

Methods to test && how to build both the method and its test code

- **It is methods that we test. Remember: we give input and expected output and assert that the actual output is identical to the expected one!**
- **Checklist on how to structure the source code of the method in a way that it is testable**:
  - Pass all input to a method as method parameters
  - Produce a return value as either the output (if the output is an object) or a description of the output (e.g., if the output is a file, return a string with its path)
- **Checklist on how to build the code of the test** :
  - The pre-existing state of the system should be created via the appropriate new() of the appropriate objects
  - The resulting state of the system would require that the object whose method we test holds the state, or that the affected objects are created via the new() calls and participate in the method's execution

47

States and values

- Input != Precondition (similarly for out-/post-)
  - Input is sth given by another s/w module or the user for the purpose of the tested method
  - Pre-cond is a setup of the objects/vrbl's & their states of the runtime
- We can think of pre- and post- conditions as **states**; we can think of a state as a set of **Boolean conditions** that must hold!
  - In the case of the Happy Path: the call is made only if the pre-condition state holds
  - In the case of Alternative Variants: we might intentionally create offending states, to check whether the code will detect them

48

States and their Boolean expression

State	Boolean correspondence
a cart having being assigned to a customer	Boolean isCartAssignedToCustomer();
The O/S being in the context of a computation	Boolean isOSInComputationContext();

The code of each such method depends on the values of the attributes of one or more collaborating objects!

49

Example of the design of a simple test

//We want to split a time line in phases & invoke an "analyser" to do it

ID	T2_V0_01	HappyDayScenario for NaiveAnalyser
Description	ON	any context
	RECEIVING	Request to analyze a valid timeline into phases
	ENSURE	That the System
	OUTPUTS	a set of phases
	SUCH THAT	state is intact
Pre-cond.		Load input_test.txt, a small file with less than 10 entries, all valid, for o(5) phases, and produce a timeline
Input		the abovementioned produced timeline
Output		A correct #phases, with the correct points inside
Post-cond.		No state properties tested
Method To test		`NaiveAnalyser.producePhasesFromTimeLine (TimeLine)`

50

Test Driven Development (TDD)

- Design the test, before even touching the code of a method!
- Allows you to envision:
  - What will be the return value of the method, as you have to design the assert statement
  - What will the parameters of the method, as you need all the inputs of the method to be present as its parameters
- Even if you do not follow it, it allows you to think of how to design the method in the first place, or refactor it, as soon as your tests prove to be hard to construct
- As always in life, when in trouble:

SEE THE END FIRST!

51

Template structure of a test

Assume we need to test the method **MyClass.getACertainValue()**
We will construct a test class MyClassTester with a method testGetACertainValueHappyDay()
What is the general structure of the **MyClassTester.testGetACertainValueHappyDay()**?

Construct a Boolean method to check the precondition, say **isPreConditionOn(...args...)**,
with all the needed objects as parameters, returning true if the preCond holds
Similarly a Boolean method **isPostCondValid(...args...)**

```
public int testGetACertainValueHappyDay(){
        //frequently, we will need key objects to be member attributes of the MyClassTester class
        //this includes a MyClass refObject = new MyClass(...); s.t. the getACertainValue canbe invoked
        Construct prior system state (ie, the needed objects) && assign any relationships between them
        Construct the input for the method
        Construct the expectedOutput
        Construct the expectedPostState
        if(isPreConditionOn(...args...){
                output = refObj.getACertainValue(...args...);  //the actual invocation
        }
        //Final check: is output == expected output?
        Assert that output == expectedOutput;
        Assert that postState == expectedPostState //(use isPostCondValid(...args...) if need)
}
```

We will see later, how to facilitate this with Junit tests

52

Focus on the important!

DESIGN OF TEST CASES

53

Incremental Testing Method

- Just see the end first here for a moment…
- What we actually want is to validate our code via **system tests**, i.e., tests that in the end, guarantee that the system does what the requirements state
- To this end, it is possible to follow an incremental black-box, acceptance-oriented strategy to develop the appropriate **test cases**

- **The pillars of the approach:**
  - Always test the core use cases of the system!
  - At least, the happy day scenario for each use case should be tested!
  - Build the tests in an **incremental fashion**, one method at a time

54

Repeat until done…

For the next **use-case** you want to implement {
  For each class.**method** you want to implement that is essential to the use case {
    PLAN THE **TESTS BEFORE/WITH** the method; */* happy day at least*/*
    IMPLEMENT TEST;
    CONSTRUCT THE METHOD'S CODE ;
    PASS THE TESTS; */* fix tests too, if tests are wrong */*
  }
  Update traceability matrix; */* see next */*
}

See how easy it is to incrementally add tests and code, if you follow this

Cannot do it for EVERY METHOD; don't let this intimidate you from doing it at least for the core methods!

A known trick: Before coding anything else, implement a failing test, run all tests and see that the new test actually fails: this ensures that the test is actually run ☺

55

Traceability matrix

- The **traceability matrix** is a matrix that says how use cases are related to test cases
  - Assigning id's to both test cases and use cases is always useful
  - A cell with an 'x' means that the respective test case (partially) tests the respective use case
  - The existence of a reusable unit in the src means that its testing verifies (parts of) more than one use cases
  - **You need at least one 'x' per use case!!!**

Test Cases x Use Cases	UC 1	UC 2	UC 3	UC 4	UC 5	UC 6
T1_V0	x					
T1_V1	x	x				
T1_V2	x	x	x			
T2_V0		x			x	
T2_V1		x				x
T2_V2		x		x		
T3_V0			x		x	
T3_V2	x	x	x		x	
…	…	…	…	…	…	…

- *In the course, when I ask for test cases, I need their full definition*
  - *Id*
  - *Description*
  - *Preconditions*
  - *Input*
  - *Postconditions*
  - *Output*
- *… and not just the traceability matrix (will be explicitly requested, too)*

56

Traceability

- The reason for keeping the linkage of use to test cases is very important:
  - We validate that the system returns what has been requested at the requirements …
  - … and ONLY these … ☺
  - **We support maintenance: whenever requirements change (and, believe me, they will), we immediately know which test cases are affected and need maintenance too.**
- @SW Eng. Body of Knowledge (SWEBOK) : "Perhaps the most crucial point in understanding software requirements is that a significant proportion of the requirements *will* change. ... Whatever the cause, it is important to recognize the inevitability of change and take steps to mitigate its effects. Change has to be managed by ensuring that proposed changes go through a defined review and approval process and by applying careful requirements tracing, impact analysis, and software configuration management."

57

Checklist

- **Focus on the important! Importance is determined by**
  - **significance,**
  - **common usage,**
  - **risk**

  … of the tested code!
- Always test at least the happy day (start with it): most significant, useful and common
  - This might require multiple tests for different equivalence classes and boundaries of the happy day scenario
- Once these have been done, try to break the software
  - Order variants by significance and risk; start with the most important and risky
  - **Risk** is commonly related with wrong/absent/… **input** or error-prone complicated **logic**
- **Obligatorily: all test methods in a test class should be independent of run order (the order with which tests are executed should not matter)**

58

Variants of a test case

- Produce **Equivalence Classes** and pick **boundaries** and **interior values** (these can still be happy day tests)
- Practically, you vary prior state and input; later, we will see how easily this can be done with Junit

- Non-happy day: Take care of …
  - **Missing** input
  - **Wrong** input (errors, null pointers, no disk/net/…)
  - **Incorrect logic** of the program
  - … AOB not mentioned here
  - *… see the next section of these slides for more…*

59

Code coverage (by tests)

- Code coverage: what percentage of [unit-of-code] has tests that call into it

/* unit-of-code: method / branch / statement */

- Don't focus so much on coverage but on importance:
  - We cannot test everything;
  - We should at least test the most significant, common, risky parts of the coe

60

Possible test cases for the 150F case

	Test 1	Test 2	Test 3	Test 4	…
ON	-	-	-	-	…
INPUT	149.9	150.0	150.1	"Abcd"	…
EXP. OUTPUT	-	Error	Error	??	…
SUCH THAT	-	Shut	Shut	??	….
		down	down		

These are the test cases!!

This is ~~just~~ a structured (and very helpful) sentence, but NOT a test case

ID1234	
ON	any context
RECEIVING	a signal that the internal temperature has reached 150 degrees Fahrenheit
ENSURE	THAT THE SYSTEM
OUTPUTS	displays an error message
SUCH THAT	the system shuts down for 5"

61

Only some possibilities out of zillion ones

DESIGNING TEST CASE VARIANTS: TRY TO BREAK THE CODE

62

The code can break because of…

- … **missing input (general overview)**
  - User does not fill a form => a parameter in a method call is null / empty string
  - A JSON file does not have a certain field
  - A csv file is missing a column
  - A file is not where it should be
  - A web stream stops fetching data
  - …
- In general, you expect data to be readily available as input to your code (either as parameters, or contents of a data store), and it's not there
- => you can TEST your code on how it behaves when this happens

63

The code can break because of…

- … **missing input @ disk level, because a file …**
  - … is not where it should be
  - … is formatted wrongly (or you assume the wrong format)
  - … does not come with the correct permissions for read/write
  - … is locked by someone else
  - … is empty
  - … is too big for your method to process it
- … **missing input @ network level, because the network …**
  - … is too slow
  - … is disconnected (test the program by switching the network off in your machine)
  - … has bursts of connectivity/disconnection
  - …

64

The code can break because of…

- … input data **errors**
  - Formatting: e.g., you expect tab delimited files and the delimiter is a ';'
  - Out of Range: you expect an int and you get "boo", or you expect a positive double and you get -45.6
  - Too big: such that your arraylist in main memory cannot hold it, or, that it makes searching in it way too slow
  - Corrupted: what if some injection attack takes place?
  - Unparsable: e.g., you expect a correctly formatted HTML file, but some elements are incorrectly typed (e.g., a '>' is missing)
  - Plainly wrong: e.g., Mr. X is reported to be pregnant, or an ID for someone is given as input and it does not exist in the database…
- In general, you expect data to be in a certain format / range / … and they are not…

65

The code can break because of…

- … **null pointers** as input
  - Special case of both the previous categories, super important to check
  - Objects passed as parameters can be null or not
  - If methods on null objects are called, … well, it's a problem… ☺
  - Check before using!!

66

The code can produce errors, too

- Here, the problem is with the **logic** that produces the output
  - Wrong formulas, computations & calculations
  - Wrong if conditions and conditional logic (avoid deep nesting of conditionals, anyway)
  - Some small typo at <, <=, …
  - Wrong iterations (out of bounds, missing positions, …)
  - Special case: wrong by one position (esp., at boundaries, array iterations…)
- Apart from constructing equivalence classes, and testing them all, there are also white-box testing techniques…

It is of uttermost importance (this is the polite way to say OBLIGATORY) to learn how to automate your tests.

YOU NEED TO MASTER THE SKILL OF BUILDING AUTOMATED TESTS

As you will see, automated testing is one of the greatest gifts ever made to developers.

AUTOMATING TESTING: UNIT TESTS, JUNIT, AND MORE

Unit tests

- **In order to test, we 'd better think small**: the fundamental testable unit of OO software is the method
- So, **we need to test methods. And we need to do it automatically. Enter Unit testing**.
- A **unit test** is a test for a small piece of code, e.g., a method, which asserts that the execution of the tested piece of code produces the expected output and post-conditions
- **Unit testing frameworks** are software engines that automate the execution of these small tests (i.e., avoid the problem of creating dozens of small main() calls) and provide facilities to assert that the output holds certain properties
- To the extent that we will ultimately map entire use cases to methods, unit testing frameworks allow us to automate the testing of mission critical parts of the code, in a homogeneous way.

Junit at Eclipse

- In this course, we will use Junit 4 (not 5) within Eclipse, as our automated unit testing framework
- See https://www.vogella.com/tutorials/JUnit/article.html

70

What to test?

- (Ideally) A Thorough To-Do List:
  - Test constructors
  - Test each method
  - Test how the attributes of the class change as the methods are executed

… NOT POSSIBLE …

INSTEAD

- **The goal is NOT to test everything!**
- The goal is to **cover the most important points of risk** in your code!
- We organize our test suite **to be expanded incrementally**, in a way that it is easy to expand the test suite one-test-at-a-time

71

OOP Particularities

- Monitor object state via encapsulation:
  - We need to construct monitoring methods that report an object's private state
- Inheritance and polymorphism:
  - When a method is overridden in a subclass, we have to test it for the specific subclass.
  - For each subclass, we need to test the implemented methods of the mama class (if any, hopefully none), as they might invoke methods that are overridden at the subclass level and thus demonstrate different behavior per subclass

72

What unit tests to make?

- **Always test the happy path of a use case!**
- Then, try to break your code with variants!
  - Test the test: before anything else, intentionally make the test fail (so that you know it is actually executed)
  - Stop testing once all areas of risk have been covered; don't worry: you 're not done testing anyway ☺
- **The goal is NOT to test everything!** The goal is to **cover the most important points of risk** in your code! Avoid spending time to points of low risk (e.g., setters and getters)
- We organize our test suite **to be expanded incrementally**, in a way that it is easy to expand the test suite one-test-at-a-time

73

How to automate tests? JUnit tests@ Eclipse

- Assume you have a class Book with methods Book(…args…), getPrice(), getTitle(), … & you want to automate their testing

1. You create a Junit class BookTest (attn: not as a regular class)
2. **Incrementally, for each test case you want to run for a method of Book, you introduce a test method in class BookTest**
3. You annotate test methods with **annotations**, like e.g., @test. This makes them "executable", much like main() – but not in the regular execution sense, but as executable tests
4. You make the appropriate new() calls to construct preconditions, either in the code of the test methods, or beforehand in a setup() method annotated as @Before
5. Then, you run Run->Run as Junit class (attn: NOT as a typical Run, that executes main()) and a Junit perspective with the results of the tests is shown to you

74

```
public class BookTest {
 private Book bookToTest;

 @Before
 public void setUp() throws Exception {
  //this runs before _each_ test. create a book to test constructor, price and final price
  bookToTest = new Book("Discours de la methode", "Rene Descartes",1637, 50.00,0,1);
 }
@Test
 public final void testGetPrice() {
  //remember: the setUp() method has run already: Book should have been initialized!
  assertEquals("test if Item.getOriginalPrice() works", 50.00, bookToTest.getOriginalPrice(), 3);
 //fail("Not yet implemented");
 //first thing to write, to ensure the test is actually run
 }

@Test
 public final void testBookNoTitle() {
  //See how this test fails. It gets a null title. The constructor should trap this
  //and avoid creating a book without a tile. Unfortunately, it fails...
  bookToTest = new Book(null, "Rene Descartes", 1637, 50.00, 0, 2);
  assertNull("test if constructor prevents creation with null title", bookToTest);
}
```

To test class Book, we introduce class BookTest, with a Book as a member attribute

Annotation @Test tells the Junit engine that this is a test to automatically run (like a main())
@Before runs before each test

Assertions make the diff of expected and actual output. *assertEquals* has (i) a msg, (ii) the expected value, (iii) the method to produce actual output, and (iv) a precision tuning

75

@ annotations

TEST CASES = test methods

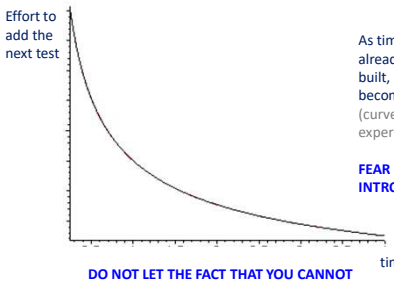
- @Test: signifies a test case
- VERY IMPORTANT: you CANNOT rely on a specific order of @Test test methods being executed => each test case should depend only upon @before and nothing else

SETUP PRE-COND & CLEANUPS

- @Before: a piece of code that runs before each test case
- @BeforeClass: a piece of code that runs once, before any other method of the test class
- @After: cleanup piece of code that runs after every test case (used to cleanup variables, objects and their linkage)

76

Show No Fear: Incremental development of tests is easy – hmm, well, after a while…



As time passes by, and you already have a few tests built, adding a new one becomes easier and easier (curve based on personal experience)

FEAR NOT THE INTRODUCTORY COST!!

DO NOT LET THE FACT THAT YOU CANNOT TEST EVERYTHING INTIMIDATE YOU FROM COVERING AS MANY REQUIREMENT VIOLATONS AS POSSIBLE!!!!

77

Most important: requirements determine the system tests

- **At the end of the day, we want to deliver a system with quality assurance guarantees for its behavior**
- This is why user requirements determine tests

Functional Req's -> Use cases -> Testable Methods

- **System tests are all about validating that user requirements are respected and we deliver exactly what we have agreed with the user!**
- Unit testing is a means to test low-level functionality && individual methods; however, since use cases are ultimately mapped to methods, we can (ab)use it to test entire use cases!

78

… and probes for more …

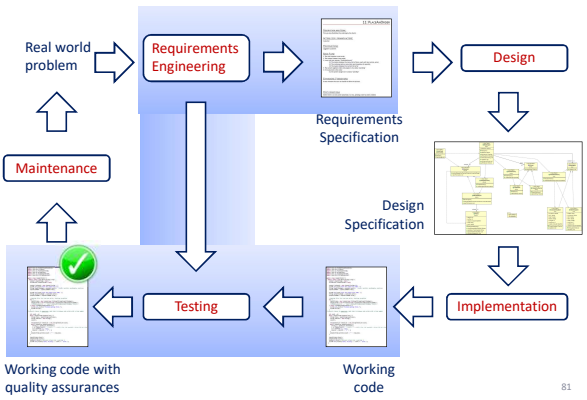
WRAPPING THINGS UP

79

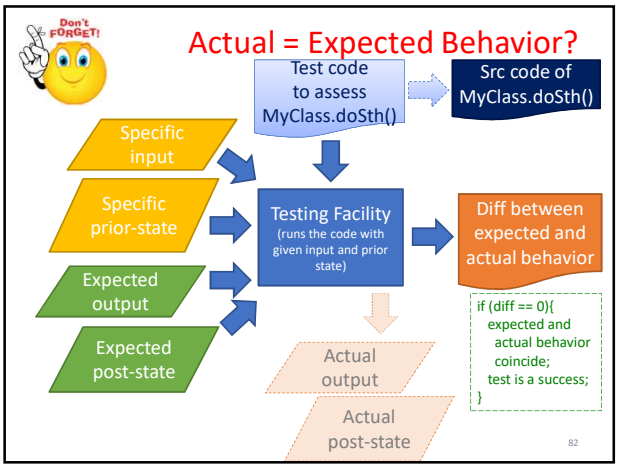
To probe further …

- We have captured a tiny piece of the area of software testing. We have omitted several categories of testing:
  - Regression testing
  - Smoke / exploratory / … testing
  - Integration testing
  - Alpha / beta /field / acceptance testing
  - Installation / usability / load / recovery / security / performance testing
- Try SWEBOK (www.swebok.org) Chapter 4 on Testing for a quick overview

80



81



Expand your test suite INCREMENTALLY!!

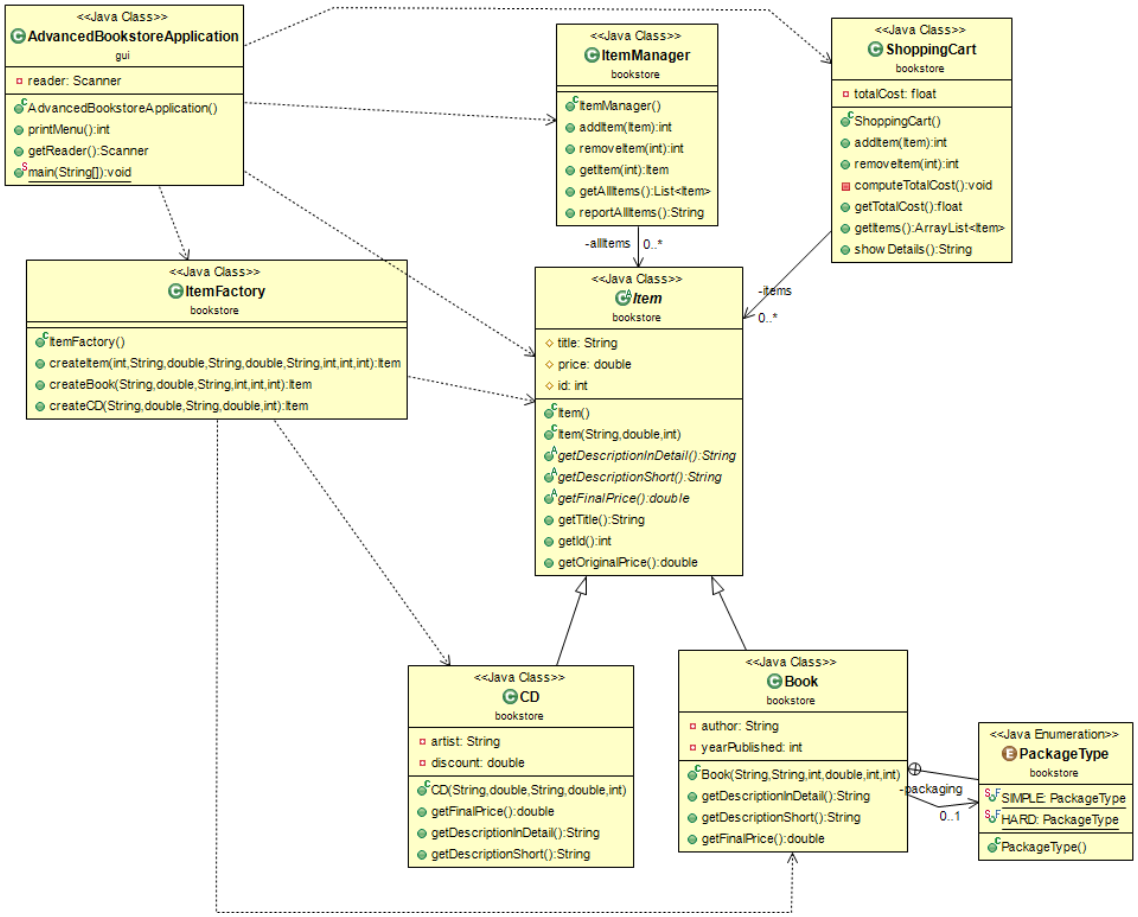
- **The goal is NOT to test everything!**
- The goal is to **cover the most important points of risk** in your code!
- We organize our test suite **to be expanded incrementally**, in a way that it is easy to expand the test suite one-test-at-a-time & run the tests fast!

DO NOT LET THE FACT THAT YOU CANNOT TEST EVERYTHING INTIMIDATE YOU FROM COVERING AS MANY REQUIREMENT VIOLATONS AS POSSIBLE!!!!

83

ΣΗΜΕΙΩΣΕΙΣ

```
package bookstore;
```




```
public abstract class Item {
  protected final String title;
  protected final double price;
  protected final int id;

  public Item(){title="";price=-1.0;id=-1;}
  public Item(String aTitle, double aPrice,int id){
    title = aTitle; price=aPrice;this.id =id;
  }


  public abstract String getDescriptionInDetail();

  public abstract String getDescriptionShort();

  public abstract double getFinalPrice();

  public String getTitle(){return title;}
  public int getId(){return id;}
  public double getOriginalPrice(){return price;}

}
```

```java
package bookstore;

public final class Book extends Item {

  public enum PackageType{
    SIMPLE, HARD;
  }

  private final String author;
  private final PackageType packaging;
  private final int yearPublished;

  /**
   * Book constructor
   *
   * @param aTitle a String with the title; should be obligatory
   * @param anAuthor a String with the author of the book
   * @param aDate an int with the year
   * @param aPrice a double with the original price of the book
   * @param aPackage an int with the id of the respective PackageType value
   * @param id a unique id for the book; should be unique
   *
   * TODO: see class comments on whether this is a good setup or not.
   */
  public Book(String aTitle, String anAuthor, int aDate, double aPrice,
      int aPackage, int id) {

    super(aTitle, aPrice, id); //constructor of Item!!!
    this.author = anAuthor;
    this.yearPublished = aDate;

    int packagePos = 0;
    if (aPackage <= PackageType.values().length - 1) {
      packagePos = aPackage;
    }
    this.packaging = PackageType.values()[packagePos];
  }

  @Override
  public String getDescriptionInDetail() {
    String packString = this.packaging.name().toLowerCase() + " packaging";
    String result = "***Item id: " + id + "***\n" +
        title + "\t\t Price:" + price + "\n" +
        " by " + author + " at " + yearPublished +  "\n" +
        "with " + packString + ".\n\n";
    return result;
  }


  @Override
  public String getDescriptionShort(){
    String packString = this.packaging.name().toLowerCase() + " packaging";
    return (title + "\nby " + author + " at " + yearPublished
        + "\nwith " + packString  + "\n");
  }

  @Override
  public double getFinalPrice() {
    return price;
  }


}//end Book
```

```java
/**
 * Small enum for Packaging Types
 * SIMPLE for 0 and HARD for 1
 * Intentionally we have avoided updating the constructors
 * to avoid making the rest of the project dependent upon a
 * very specific enum.
 * So the constructor takes an int as an argument for the package.
 *
 * The problem: what if I change the order of the enum values later?
 * Everything is gonna be wrong
 * Also: what happens if the constructor takes sth other than
 * an available int?
 *
 * @todo TODO  Can *you* find a way to ensure that the rest of
 * the system is intact & we avoid a dependency? Is it better this way?
 * What price would you pay and what would you gain?
 * (there is no silver bullet)
 * Is it better -in the end- to just use a PackageType
 * in the constructor and let everyone know?
 */
```

```java
package bookstore;

/**
* <h1>CD</h1>
* Class responsible from handling all
* the CD items.
*
* @version 1.0
* @since   2017-07-17
*/
public final class CD extends Item {
  private final String artist;
  private final double discount;

  public CD(String aTitle, double aPrice, String anArtist, double aDiscount, int id) {
    super(aTitle, aPrice, id);
    artist = anArtist;
    discount = aDiscount;
  }

  @Override
  public double getFinalPrice() {
    return (price - discount);
  }

  @Override
  public String getDescriptionInDetail() {
    String result = "***Item id: " + id + "*** \n" +
        title + "\t\t Price:" + price + "\n" +
        "by " + artist + "\n" +
        "final price: " + getFinalPrice()+ "\n\n";
    return result;
  }

  @Override
  public String getDescriptionShort(){
    return (title + "\n by " + artist);
  }
}
```

```
package bookstore;

public class ItemFactory {
  /*
   * Always bad to do sth like the following.
   *
   * Why? Think what happens if subclasses of Item start evolving. What happens?
   *
   * *DO NOT USE FACTORIES LIKE THIS!* Try two different methods, instead!
   */
  @Deprecated
  public Item createItem(int aType, String aTitle, double aPrice, String anArtist,
double aDiscount, String anAuthor, int aDate, int aPackage,int id){

    switch(aType){
    case 1:  Book newBook = new Book(aTitle, anAuthor, aDate, aPrice,aPackage,id);
          return newBook;
    case 2:  CD newCD = new CD(aTitle, aPrice, anArtist, aDiscount, id);
          return newCD;
    default: System.out.println("Wrong type of item for createItem() -- nothing
created");
          return null;
    }
  }

  public Item createBook(String aTitle, double aPrice, String anAuthor, int aDate, int
aPackage,int id){
    Book newBook = new Book(aTitle, anAuthor, aDate, aPrice,aPackage, id);
    return newBook;
  }


public Item createCD(String aTitle, double aPrice, String anArtist, double aDiscount,
int id){
    CD newCD = new CD(aTitle, aPrice, anArtist, aDiscount, id);
    return newCD;
  }
}
```

```java
package bookstore;

import java.util.ArrayList;
import java.util.List;

public final class ItemManager {
  private final List<Item> allItems;

  public ItemManager(){
    allItems = new ArrayList<Item>();
  }

  public int addItem(Item anItem){
    allItems.add(anItem);
    return allItems.size();
  }

  public int removeItem(int index){
    this.allItems.remove(index);
    return allItems.size();
  }

  public Item getItem(int index){
    return this.allItems.get(index);
  }

  public List<Item> getAllItems(){
    return this.allItems;
  }

  public String reportAllItems(){
    String result ="";
    for(Item item: this.allItems) {
      result = result + item.getDescriptionInDetail();
    }
    result = result + "Total number of items: " + this.allItems.size()+ "\n";
    System.out.println(result);
    return result;
  }
}
```

```java
package bookstore;

import java.util.ArrayList;
import java.util.List;

public class ShoppingCart {

  private final List<Item> items;
  private float totalCost;

  public ShoppingCart(){
    this.items = new ArrayList<Item>();
    this.totalCost = 0;
  }

  public int addItem(Item item){
    this.items.add(item);
    return this.items.size();
  }

  public int removeItem(int anId){
    int pos = -1;
    for(int i = 0; i < this.items.size(); i++){
      if (items.get(i).getId() == anId){
        pos = i;
        break;
      }
    }
```

```
    if (pos>=0)
      this.items.remove(pos);
    else{
      System.out.println("The id you have specified is not in the cart");
    }
    return this.items.size();
  }

  private void computeTotalCost(){
    float cost = 0;
    for(Item item: this.items){
      cost += item.getFinalPrice();
    }
    this.totalCost = cost;
  }

  public float getTotalCost(){
    this.computeTotalCost();
    return this.totalCost;
  }

  public List<Item> getItems(){
    return this.items;
  }

  public String showDetails(){
    String intro = "------------------------\n" +
                   "       CART ITEMS       \n" +
                   "------------------------\n";

    String itemsString = "";
    for(int i = 0; i < this.items.size(); i++){
      //System.out.println("***Item id: " + i + "***");
      itemsString = itemsString + items.get(i).getDescriptionInDetail();
    }
    String costString = "Total cost: " + this.getTotalCost() + "\n";

    String result = intro + itemsString + costString;
    System.out.println(result);

    return result;
  }
}
```

```
package gui;

import java.util.Scanner;
import bookstore.Item;
import bookstore.ItemFactory;
import bookstore.ItemManager;
import bookstore.ShoppingCart;

public class AdvancedBookstoreApplication {

  private Scanner reader;

  public AdvancedBookstoreApplication(){
    reader = new Scanner(System.in);
  }

  public int printMenu(){
    int answerOperation = 0;
    while( answerOperation > 5 || answerOperation <= 0){
      System.out.println("Choose(1-4)\n 1. Show items\n 2. Add item to cart\n " +
          + " 3. Show cart\n 4. Remove item from cart\n 5. Exit");
      answerOperation = reader.nextInt();
      if(answerOperation > 5 || answerOperation <= 0)
        System.out.println("Wrong answer! Try again...");
    }
    return answerOperation;
  }
```

```
  public Scanner getReader(){
    return reader;
  }


  public static void main(String args[]){

    AdvancedBookstoreApplication app = new AdvancedBookstoreApplication();

    ItemManager itemManager = new ItemManager();
    ItemFactory itemFactory = new ItemFactory();

    //  Book bookRef;

    Item item = itemFactory.createBook("Discours de la methode", 50.00, "Rene
Descartes", 1637,  0,0);
    itemManager.addItem(item);
    item = itemFactory.createBook("The Meditations", 30.00,  "Marcus Aurelius", 180,
1,1);
    itemManager.addItem(item);
    item = itemFactory.createBook("The Bacchae",30.00, "Euripides", -405, 1,2);
    itemManager.addItem(item);
    item = itemFactory.createBook("The Trojan Women",40.00, "Euripides",-415, 0,3);
    itemManager.addItem(item);

    //CD cdRef;
    item = itemFactory.createCD("Piece of Mind", 10.0,"Iron Maiden",4.0,4);
    itemManager.addItem(item);
    item = itemFactory.createCD("Matter of Life and Death", 12.0,"Iron Maiden",2.0,5);
    itemManager.addItem(item);
    item = itemFactory.createCD("Perfect Strangers", 12.00, "Deep Purple",1.0,6);
    itemManager.addItem(item);

    ShoppingCart cart = new ShoppingCart();

    while(true){
      int operation = app.printMenu();
      if(operation == 1){
        itemManager.reportAllItems();
      }
      else if(operation == 2){
        System.out.println("Choose the id of the item you want to add to your cart");
        int id = app.getReader().nextInt();
        if(id > itemManager.getAllItems().size())
          System.out.println("Error: there is no product with the specified id");
        else
          cart.addItem(itemManager.getItem(id));
      }
      else if(operation == 3){
        cart.showDetails();
      }
      else if(operation == 4){
        System.out.println("Choose the id of the item to remove from your cart");
        int id = app.getReader().nextInt();
        if(id > itemManager.getAllItems().size())
          System.out.println("Error: there is no product with the specified id");
        else
          cart.removeItem(id);
      }
      else{
        break;
      }
    }
```

```
package testPackage;
```

```
package test.testPackage;

import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;

@RunWith(Suite.class)
@SuiteClasses({ BookTest.class, CDTest.class, ItemManagerTest.class,
ShoppingCartTest.class })
public class AllTests {
  //no need to add sth here. The above directives simply run all tests
}
```

```
package test.testPackage;

import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import bookstore.CD;

public class CDTest {

  private static CD cdToTest; //attn, must be static because we will use it at beforeClass!

  @BeforeClass
  public static void setUpBeforeClass() throws Exception {
    //this runs once, before _all_ tests. Here we create a single object(to save time)
    //and use it in all tests.
    cdToTest = new CD("Piece of Mind", 10.0,"Iron Maiden",4.0, 7);
  }

  @Before
  public void setUp() throws Exception {
    //see BookTest for explanations.
  }

  //Also here, we omit the tests for null / invalid construction values, as we did in
Book

  @Test
  public final void testGetFinalPrice() {
    //CD cdToTest = new CD("Piece of Mind", 10.0,"Iron Maiden",4.0);
    assertEquals("test if getFinalPrice() works OK", 6.00,cdToTest.getFinalPrice(),3);

    //fail("Not yet implemented"); // TODO
  }

  @Test
  public final void testGetPrice() {
    //CD cdToTest = new CD("Piece of Mind", 10.0,"Iron Maiden",4.0);
    assertEquals("test if getPrice() of the ITEM CLASS works OK", 10.00,
cdToTest.getOriginalPrice(), 3);

    //fail("Not yet implemented"); // TODO
  }
}
```

```java
package test.testPackage;

import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import bookstore.Book;

public class BookTest {
  private Book bookToTest;

  @BeforeClass
  public static void setUpBeforeClass() throws Exception {
    //this runs once, before _all_ tests. Nothing todo here.
    //See CDTest for explanations
  }

  @Before
  public void setUp() throws Exception {
    bookToTest = new Book("Discours de la methode", "Rene Descartes",1637, 50.00,0,1);
    //this runs before _each_ test. create a book to test constr., price & final price
  }

  @Test
  public final void testBookNull() {
    //remember: the setUp() method has run already: Book should have been initialized!
    assertNotNull("After setup, the book is not null", bookToTest);

    //at the beginning to see that the test works.
    //fail("Not yet implemented");
  }

  @Test
  public final void testBookNoTitle() {

    //See how this test fails. It gets a null title. The constructor should trap this
    //and avoid creating a book without a tile. Unfortunately, it fails...
    bookToTest = new Book(null, "Rene Descartes", 1637, 50.00, 0, 2);
    assertNull("test if constructor prevents creation with null title", bookToTest);

    //at the beginning to see that the test works.
    //fail("Not yet implemented");
  }

  @Test
  public final void testBookNegativePrice() {
  //should do the same with negative price. Again, intentionally put to show failure!
    bookToTest = new Book("Discours de la methode", "Rene Descartes",1637,-12.00,0,3);
    assertNull("test if the constructor prevents creation with negative price",
bookToTest);

    //at the beginning to see that the test works.
    //fail("Not yet implemented");
  }

  @Test
  public final void testGetFinalPrice() {
    assertEquals("test if getFinalPrice() works OK", 50.00,
                         bookToTest.getFinalPrice(), 3);
    //fail("Not yet implemented");
  }

  @Test
  public final void testGetPrice() {
    assertEquals("test if getOriginalPrice of the ITEM CLASS works OK", 50.00,
                           bookToTest.getOriginalPrice(), 3);
    //fail("Not yet implemented");
  }
}
```

```java
package test.testPackage;

import static org.junit.Assert.*;

import org.junit.Before;
import org.junit.Test;

import bookstore.Book;
import bookstore.Item;
import bookstore.ItemManager;

public class ItemManagerTest {

  @Before
  public void setUp() throws Exception {
    //this is supposed to run before _all_ tests
    //nothing to set up here
  }

  @Test
  public void testItemManager() {
    ItemManager amazon = new ItemManager();
    assertNotNull(amazon.getAllItems());
    //before implementing the above, try having just the following fail.
    //Run the test and see that it fails indeed.
    //Then build your test.
    //fail("Intentional failure. Not yet implemented");
  }

  @Test
  public void testAddItem() {
    ItemManager amazon = new ItemManager();
    assertEquals(0,amazon.getAllItems().size());

    Book bookRef;
    bookRef = new Book("Discours de la methode", "Rene Descartes", 1637, 50.00, 0, 9);
    amazon.addItem(bookRef);
    assertNotEquals(0,amazon.getAllItems().size());
    assertEquals(1,amazon.getAllItems().size());

    //fail("Not yet implemented");
  }

  @Test
  public void testReportAllItems() {
    ItemManager itemManager = new ItemManager();

    Item item = new Book("Discours de la methode","Rene Descartes", 1637, 50.00, 0,0);
    itemManager.addItem(item);
    item = new Book("The Meditations",   "Marcus Aurelius", 180,30.00, 1,1);
    itemManager.addItem(item);

    String expectedResult = "***Item id: 0***\n" +
    "Discours de la methode    Price:50.0\n" +
    " by Rene Descartes at 1637\n" +
    "with simple packaging.\n" +
    "\n" +
    "***Item id: 1***\n" +
    "The Meditations    Price:30.0\n" +
    " by Marcus Aurelius at 180\n" +
    "with hard packaging.\n" +
    "\nTotal number of items: 2\n";
    assertEquals(expectedResult, itemManager.reportAllItems());
  }

}
```

```java
package test.testPackage;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNotEquals;
import static org.junit.Assert.assertNotNull;
import org.junit.Test;

import bookstore.Book;
import bookstore.Item;
import bookstore.ShoppingCart;

public class ShoppingCartTest {

  @Test
  public void testItemManager() {
    ShoppingCart cart = new ShoppingCart();
    assertNotNull(cart.getItems());
  }

  @Test
  public void testAddItem() {
    ShoppingCart cart = new ShoppingCart();
    assertEquals(0,cart.getItems().size());

    Book bookRef;
    bookRef = new Book("Discours de la methode", "Rene Descartes", 1637, 50.00, 0, 9);
    cart.addItem(bookRef);
    assertNotEquals(0,cart.getItems().size());
    assertEquals(1,cart.getItems().size());

  }

  @Test
  public void testRemoveItem(){
    ShoppingCart cart = new ShoppingCart();
    assertEquals(0,cart.getItems().size());

    Book bookRef;
    bookRef = new Book("Discours de la methode", "Rene Descartes", 1637, 50.00, 0, 9);
    cart.addItem(bookRef);
    assertNotEquals(0,cart.getItems().size());
    assertEquals(1,cart.getItems().size());

    cart.removeItem(9);
    assertEquals(0,cart.getItems().size());
  }

  @Test
  public void testShowItems(){
    ShoppingCart cart = new ShoppingCart();

    Item item = new Book("Discours de la methode","Rene Descartes", 1637, 50.00, 0,0);
    cart.addItem(item);
    item = new Book("The Meditations",   "Marcus Aurelius", 180,30.00, 1,1);
    cart.addItem(item);

    String expectedResult =
        "------------------------\n" +      "       CART ITEMS       \n" +
        "------------------------\n" +
        "***Item id: 0***\n" +
        "Discours de la methode    Price:50.0\n" +
        " by Rene Descartes at 1637\n" +      "with simple packaging.\n" +
        "\n" +
        "***Item id: 1***\n" +
        "The Meditations    Price:30.0\n" +      " by Marcus Aurelius at 180\n" +
        "with hard packaging.\n\n" +      "Total cost: 80.0\n";
    assertEquals(expectedResult, cart.showDetails());

  }
}
```

ΣΗΜΕΙΩΣΕΙΣ

Αγώνας ταχύτητας ποδηλάτων

Εταιρεία εκπαιδευτικού λογισμικού κατασκευάζει ηλεκτρονικό παιχνίδι για παιδιά δημοτικού, στο οποίο οι μικροί μαθητές συναρμολογούν ένα ποδήλατο από τα εξαρτήματά του και διεξάγουν αγώνα. Το παιχνίδι προσφέρει επιλογές για τα είδη πεταλιών και φρένων που μπορεί να μπουν σε ένα ποδήλατο και των οποίων η επίδοση υπολογίζεται με βάση κάποιους αριθμητικούς τύπους.

Πιο συγκεκριμένα, η επιλογή των φρένων γίνεται από δύο διαφορετικά είδη:

- Απλά φρένα: προσφέρουν μείωση της ταχύτητας κατά 35*force (βαθμός της πίεσης των φρένων) και έχουν μια πιθανότητα αποτυχίας η οποία εξαρτάται από κάποιο κατώφλι.
- Duper φρένα: προσφέρουν μείωση της ταχύτητας κατά 45*force (βαθμός της πίεσης των φρένων) και δεν υπάρχει πιθανότητα βλάβης κατά την διάρκεια ενός αγώνα.

Επιπλέον, η επιλογή των πεταλιών γίνεται από δύο διαφορετικά είδη πεταλιών:

- Απλά πετάλια: η ταχύτητα που δίνουν δίνεται από τον τύπο 35*pedaling rate (ρυθμός κίνησης των πεταλιών από τον αναβάτη)
- Zuper πετάλια: η ταχύτητα που δίνουν δίνεται από τον τύπο 45* pedaling rate (ρυθμός κίνησης των πεταλιών από τον αναβάτη) + 650

Με βάση τα παραπάνω, η ταχύτητα του ποδηλάτου εξαρτάται από δύο παράγοντες: (α) τον ρυθμό με τον οποίο ο αναβάτης ποδηλατεί (σε συνδυασμό με το είδος των πεταλιών) και (β) από την πίεση που ασκείται στα φρένα του ποδηλάτου (σε συνδυασμό με το είδος των φρένων που χρησιμοποιούνται).

Η διεξαγωγή του αγώνα γίνεται σε μία πίστα η οποία αποτελείται από 4 διαδρομές. Στην αρχή κάθε διαδρομής, για κάθε ποδηλάτη, τόσο η πίεση των φρένων όσο και ο ρυθμός κίνησης των πεταλιών επιλέγεται τυχαία στο διάστημα [0,30] για τα φρένα και στο [0,40] για τα πετάλια. Ο μαθητής (i) κατασκευάζει κάποια ποδήλατα (όνομα ποδηλάτη, πετάλια, φρένα, αρχική ταχύτητα), (ii) εκκινεί τον αγώνα και (iii) στη συνέχεια, συλλέγει τα διαγνωστικά μηνύματα του προγράμματος. Κάθε φορά που ο μαθητής επιλέγει την κατασκευή ενός ποδηλάτου, πρέπει να δώσει όνομα ποδηλάτη, τύπο φρένων και πεταλιών και αρχική ταχύτητα με την οποία ο ποδηλάτης εκκινεί το ποδήλατο στην αρχική διαδρομή. Ο αγώνας προσομοιώνεται ως εξής: για κάθε διαδρομή, για κάθε ποδηλάτη, υπολογίζεται η επιτάχυνση από τα πετάλια και η επιβράδυνση από τα φρένα, καθώς και ο χρόνος που του παίρνει να περάσει τη διαδρομή. Αν το ποδήλατο χαλάσει μέσα στη διαδρομή αυτή, αποσύρεται από τον αγώνα. Νικητής αναδεικνύεται όποιος ολοκληρώσει και τις 4 διαδρομές και έχει το μικρότερο άθροισμα χρόνων συνολικά.

Με βάση τα διαγνωστικά μηνύματα, και πλέον, ευρισκόμενος εκτός της εφαρμογής, χρησιμοποιεί τα αποτελέσματα για να επαληθεύσει τους αριθμούς χειροκίνητα, και να εξηγήσει τι έγινε σε μια αναφορά.

Θέλουμε να δημιουργήσουμε μια εφαρμογή η οποία προσομοιώνει:

- την κατασκευή των διαφορετικών ποδηλάτων με βάση τα διαφορετικά είδη εξαρτημάτων τα οποία μπορούν να προστεθούν σε αυτά.
- τον αγώνα ταχύτητας μεταξύ των κατασκευασθέντων ποδηλάτων.

ΚατασκεύασεΠοδήλατο

ID: UC 1

Description and Goal

Η use case «ΚατασκεύασεΠοδήλατο» υλοποιεί την βασική λειτουργία κατασκευής ενός ποδήλατο. Πιο συγκεκριμένα, συλλέγει τις επιλογές του χρήστη για την κατασκευή ενός ποδηλάτου.

Actors (esp. primary actor)

Ο μαθητής.

Preconditions

-

Basic Flow

1. Η UC ξεκινάει όταν ο χρήστης της εφαρμογής επιλέξει από το μενού την δημιουργία νέου ποδηλάτου.
2. Το σύστημα συλλέγει από τον χρήστη τις κατάλληλες πληροφορίες για την κατασκευή ενός ποδηλάτου.

    2.1 Για κάθε ποδήλατο

    2.1.1 Το σύστημα συλλέγει το όνομα του ποδηλάτη

    2.1.2 Το σύστημα συλλέγει το είδος των φρένων που θα χρησιμοποιηθούν στο ποδήλατο

    2.1.3 Το σύστημα συλλέγει το είδος των πεταλιών που θα χρησιμοποιηθούν στο ποδήλατο.

    2.1.4 Το σύστημα συλλέγει την αρχική ταχύτητα του ποδηλάτου.

Extensions / Variations

Καμία

Post conditions

Το ποδήλατο έχει καταχωρηθεί στην λίστα με τα διαγωνιζόμενα ποδήλατα.

ΠροσομοίωσεΑγώναΤαχύτητας

ID: UC 2

Description and Goal

Το use case «ΠροσομοίωσεΑγώναΤαχύτητας» υλοποιεί την λειτουργία της εκτέλεσης του αγώνα ταχύτητας μεταξύ των διαγωνιζόμενων ποδηλάτων.

Actors (esp. primary actor)

Ο μαθητής.

Preconditions

Πρέπει να υπάρχουν διαθέσιμα ποδήλατα στην λίστα με τους διαγωνιζόμενους.

Basic Flow

1. Το use case ξεκινάει όταν ο χρήστης επιλέγει την επιλογή του αγώνα ταχύτητας.
2. Το σύστημα εκτελεί την προσομοίωση του αγώνα ταχύτητας.

2.1 Για κάθε διαδρομή

2.1.1 Για κάθε ποδήλατο

2.1.1.1 Το σύστημα υπολογίζει την επιτάχυνση από τα πετάλια

2.1.1.2 Το σύστημα υπολογίζει την επιβράδυνση από τα φρένα

2.1.1.3 Το σύστημα υπολογίζει τον χρόνο ολοκλήρωσης της διαδρομής.

2.1.1.4 Αν το ποδήλατο έχει χαλάσει,

2.1.1.4.1 Το σύστημα αποσύρει από τον αγώνα τον ποδηλάτη

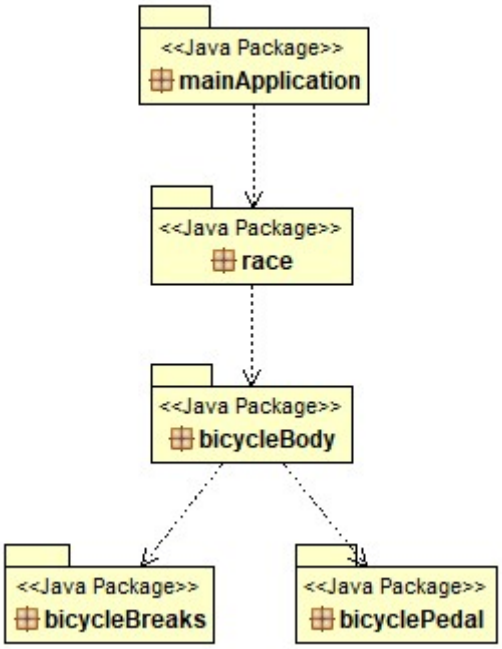
2.1.1.5 Το σύστημα εμφανίζει τα τρέχοντα στοιχεία του ποδηλάτου

2.2 Το σύστημα υπολογίζει το νικητή.

3. Το σύστημα εμφανίζει στην κονσόλα τα αποτελέσματα του αγώνα ταχύτητας.

Post conditions

Ο αγώνας ταχύτητας έχει ολοκληρωθεί και τα αποτελέσματα έχουν αναφερθεί στον χρήστη της εφαρμογής.


<<Java Package>>
mainApplication
<<Java Package>>
race
<<Java Package>>
bicycleBody
<<Java Package>>
bicycleBreaks
<<Java Package>>
bicyclePedal

```
package mainApplication;
```

```
package mainApplication;
import java.util.Scanner;
import race.IRace;
import race.RaceTypeFactory;

public class BicycleSimpleApplication {
  private Scanner reader;

  public BicycleSimpleApplication(){
    reader = new Scanner(System.in);
  }

  public Scanner getReader(){
    return reader;
  }

  public int chooseType(String message){
    int answer = 0;
    while( answer > 2 || answer <= 0){
      System.out.println(message);
      answer = reader.nextInt();
      if(answer > 2 || answer <= 0)
        System.out.println("Wrong answer! Try again...");
    }
    return answer;
  }


  public int printMenu(){
    int answerOperation = 0;
    while( answerOperation > 3 || answerOperation <= 0){
      System.out.println("Choose(1-3)\n 1. Add vehicle\n 2. Race\n 3. Exit");
      answerOperation = reader.nextInt();
      if(answerOperation > 3 || answerOperation <= 0)
        System.out.println("Wrong answer! Try again...");
    }

    return answerOperation;
  }

  public void closeReader(){
    reader.close();
  }

  public static void main(String[] args) {
    BicycleSimpleApplication bTester = new BicycleSimpleApplication();
    RaceTypeFactory raceFactory = new RaceTypeFactory();
    IRace r = raceFactory.createRaceWithStages();

    while(true){
      int operation = bTester.printMenu();
      if(operation == 1){
        System.out.println("Choose a name for your bicycle:");
        String name = bTester.getReader().next();
        System.out.println(name);

        int answerBrake = bTester.chooseType("Choose brake type (1/2)\n"
            + "1. Nice brakes\n2. Duper brakes");
        int answerPedal = bTester.chooseType("Choose pedal type (1/2)\n"
            + "1. Simple pedal\n2. Zuper pedal");

        System.out.println("Select the initial velocity for your bicycle:");
        double velocity = bTester.getReader().nextDouble();
```

```
        if(answerBrake == 1 && answerPedal == 1){
          r.setupVehicle("SimplePedal", "NiceBrakes", name, velocity);
        }
        else if(answerBrake == 1 && answerPedal == 2){
          r.setupVehicle("ZuperPedal", "NiceBrakes", name, velocity);
        }
        else if(answerBrake == 2 && answerPedal == 1){
          r.setupVehicle("SimplePedal", "DuperBrakes", name, velocity);
        }
        else{//answerBrake == 2 && answerPedal == 2
          r.setupVehicle("ZuperPedal", "DuperBrakes", name, velocity);
        }
      }
      else if(operation == 2){
        r.runRace();
      }
      else{
        break;
      }
    }
    //r.setupVehicles();
    bTester.closeReader();
  }
}
```

```
package bicycleBody;
```

```
import bicycleBreaks.IBreaks;
import bicycleBreaks.BreakFactory;
import bicyclePedal.IPedal;
import bicyclePedal.PedalFactory;

public class Bicycle {

  public Bicycle(String pedalsName, String breaksName, String name){
    breakFactory = new BrakeFactory();
    pedalFactory = new PedalFactory();
    breaks = breakFactory.constructBrake(breaksName);
    pedal = pedalFactory.constructPedal(pedalsName);
    velocity = 0.0;
    timeRun = 0.0;
    this.name = name;
  }

  public double getVelocity(){
    return velocity;
  }

  public double getTimeRun(){
    return timeRun;
  }

  public void computeTime(double distance){
    timeRun += distance/velocity;
  }

  public String getName(){
    return name;
  }

  public void setOriginalVelocity(double originalVelocity){
    velocity = originalVelocity;
  }

  public double setPedaling(double rate){
    velocity += pedal.getSpeed(rate);
    return velocity;
  }

  public double setBraking(double force){
    velocity -= breaks.getSpeedReduction(force);
    if(velocity < 0)
      velocity = 0;
    return velocity;
  }

  public void reportDetails(){

    System.out.println(getName() + ": velocity is: " + String.format(
"%.2f",getVelocity()) + " and time run is: " + String.format( "%.2f",getTimeRun()));

    if(reportIfDamageExists() == false)
      System.out.println("Status: OK");
    else
      System.out.println("Status: Broken");
    System.out.println("--------------------");
  }

  public boolean reportIfDamageExists(){
    boolean brokenStatus = false;
    if (breaks.reportIfBroken(0.7) == true){
      brokenStatus = true;
      System.out.println("Breaks are broken");
```

```
    }
    if (pedal.reportIfBroken() == true){
      brokenStatus = true;
      System.out.println("Pedals are broken");
    }
    return brokenStatus;
  }

  public boolean equals(Bicycle bicycle){
    if(getName().equals(bicycle.getName()))
      return true;

    return false;
  }

  private double timeRun;
  private String name;
  private double velocity;
  private IBrakes breaks;
  private IPedal pedal;
  private BrakeFactory breakFactory;
  private PedalFactory pedalFactory;
}
```

```
package bicycleBreaks;
```

```
public interface IBrakes {
  public abstract double getSpeedReduction(Double excertedForce);
  public abstract boolean reportIfBroken(double threshold);
}
```

```
public class BrakeFactory {
  public IBrakes constructBrake(String concreteClassName){
    if (concreteClassName.equals("NiceBrakes"))
      return new NiceBrakes();
    else if (concreteClassName.equals("DuperBrakes"))
      return new DuperBrakes();

    System.out.println("If the code got up to here, you passed a wrong argument to
BreakFactory");
    return null;
  }
}
```

```
import java.util.Random;

public class NiceBrakes implements IBrakes {
  @Override
  public double getSpeedReduction(Double excertedForce){
    return 35*excertedForce;
  }

  @Override
  public boolean reportIfBroken(double threshold) {
    if ((threshold > 1.0) || (threshold > 1.0)){
     System.out.println("The threshold for breaks' health should be between 0 and 1");
      System.exit(-1);
    }

    Random randomDice = new Random();
    if (randomDice.nextDouble() > threshold)
      return false;
    else
      return true;
  }
}

public class DuperBrakes implements IBrakes {

  @Override
  public double getSpeedReduction(Double excertedForce){
    return 45*excertedForce;
  }

  @Override
  public boolean reportIfBroken(double threshold) {
    //Intentional issue, food for thought: parameter unused!
    return false;
  }

}
```

```
package bicyclePedal;
```

```
public interface IPedal {
  public abstract double getSpeed(double pedalingRate);
  public abstract boolean reportIfBroken();
}
```

```
public class PedalFactory {
  public IPedal constructPedal(String concreteClassName){
    if (concreteClassName.equals("SimplePedal"))
      return new SimplePedal();
    else if (concreteClassName.equals("ZuperPedal"))
      return new ZuperPedal();

    System.out.println("If the code got up to here, you passed a wrong argument to
PedalFactory");
    return null;
  }
}
```

```
import java.util.Random;

public class SimplePedal implements IPedal {

  @Override
  public double getSpeed(double pedalingRate) {
    return 35 * pedalingRate;

  }

  @Override
  public boolean reportIfBroken() {
    Random randomDice = new Random();
    if (randomDice.nextDouble() > 0.6)
      return false;
    else
      return true;
  }
}
```

```
package bicyclePedal;

public class ZuperPedal implements IPedal {

  @Override
  public double getSpeed(double pedalingRate) {
    return 45 * pedalingRate + 650;
  }

  @Override
  public boolean reportIfBroken() {
    // TODO Auto-generated method stub
    return false;
  }

}
```

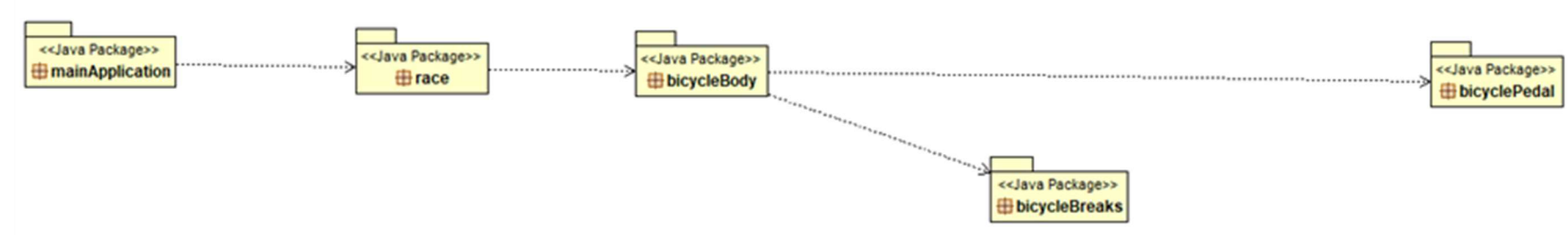
```
package race;
```

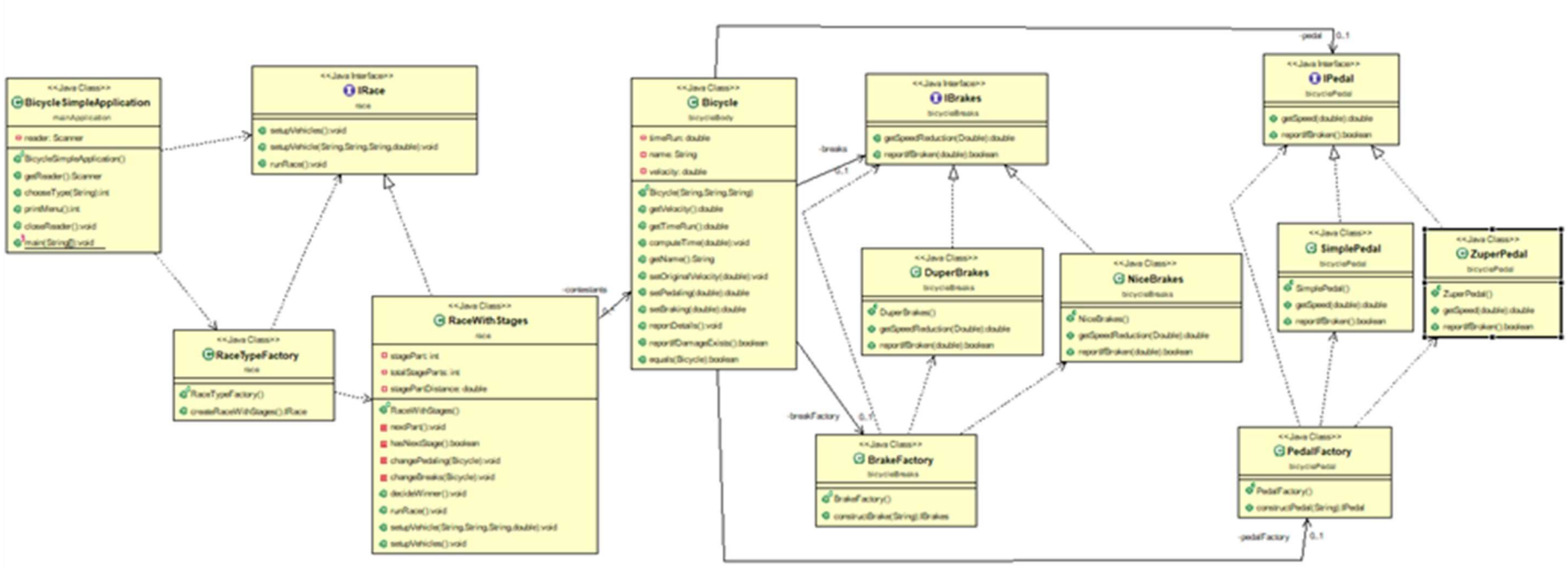
```
public interface IRace {
  public void setupVehicles();
  public void setupVehicle(String pedalType, String brakeType, String name, double
velocity);
  public void runRace();
}
```

```
public class RaceTypeFactory {
  public IRace createRaceWithStages(){
    return new RaceWithStages();
  }
}
```

```
public class RaceWithStages implements IRace{


  private int stagePart = 0;
  private int totalStageParts = 4;
  /*We suppose for simplicity that all the parts
   *of the stage have the same length
   */
  private double stagePartDistance = 2000;
  private ArrayList<Bicycle> contestants;

  public RaceWithStages(){
    contestants = new ArrayList<Bicycle>();
  }

  private void nextPart(){
    stagePart++;
  }

  private boolean hasNextStage(){
    if(stagePart < totalStageParts)
      return true;

    return false;
  }

  private void changePedaling(Bicycle contestant){
    Random randomDice = new Random();
    double rate = 40 * randomDice.nextDouble();
    contestant.setPedaling(rate);
    System.out.println("Contestant " + contestant.getName() + " pedals with rate " +
String.format( "%.2f",rate));
  }

  private void changeBreaks(Bicycle contestant){
    Random randomDice = new Random();
    double force = 30 * randomDice.nextDouble();
    contestant.setBraking(force);
    System.out.println("Contestant " + contestant.getName() + " breaks with force " +
String.format( "%.2f",force));
  }


  public void decideWinner(){
    double minTime = Double.MAX_VALUE;
    Bicycle winner = null;

    if(contestants.size() == 0){
      System.out.println("There is no winner for this race...");
      return;
    }
```

```
    for(Bicycle contestant: contestants){
      if(contestant.getTimeRun() < minTime){
        minTime = contestant.getTimeRun();
        winner = contestant;
      }
    }
    System.out.println("Winner Details");
    winner.reportDetails();
  }

  @Override
  public void runRace(){
    while(hasNextStage()){
      System.out.println("====================\nPart: "
          + stagePart + "\n");
      ArrayList<Bicycle> brokenContestants = new ArrayList<Bicycle>();
      for(Bicycle bm: contestants){
        changePedaling(bm);
        changeBreaks(bm);
        bm.computeTime(stagePartDistance);
        //Mark broken contestant
        if(bm.reportIfDamageExists())
          brokenContestants.add(bm);
        //report contestant's details
        bm.reportDetails();
      }
      //Remove the contestants that are broken
      for(Bicycle brokenContestant: brokenContestants){
        contestants.remove(brokenContestant);
      }
      nextPart();
    }
    System.out.println("End of RaceWithStages\n\n");
    decideWinner();
  }

  @Override
  public void setupVehicle(String pedalType, String brakeType, String name, double
velocity){
    Bicycle bicycleConstructed = new Bicycle(pedalType, brakeType, name);
    bicycleConstructed.setOriginalVelocity(velocity);
    System.out.println(name + ": velocity at start is: " +
bicycleConstructed.getVelocity());
    contestants.add(bicycleConstructed);
  }

}
```


<<Java Package>>
mainApplication
<<Java Package>>
race
<<Java Package>>
bicycleBody
<<Java Package>>
bicyclePedal
<<Java Package>>
bicycleBreaks


BicycleSimpleApplication
mainApplication
reader: Scanner
BicycleSimpleApplication()
getReader():Scanner
chooseType(String):int
printMenu():int
closeReader():void
main(String[]):void
IRace
race
setupVehicles():void
setupVehicle(String,String,String,double):void
runRace():void
RaceTypeFactory
race
RaceTypeFactory()
createRaceWithStages():IRace
RaceWithStages
race
stagePart: int
totalStageParts: int
stagePartDistance: double
RaceWithStages()
nextPart():void
hasNextStage():boolean
changePedaling(Bicycle):void
changeBreaks(Bicycle):void
decideWinner():void
runRace():void
setupVehicle(String,String,String,double):void
setupVehicles():void
-contestants
Bicycle
bicycleBody
timeRun: double
name: String
velocity: double
Bicycle(String,String,String)
getVelocity():double
getTimeRun():double
computeTime(double):void
getName():String
setOriginalVelocity(double):void
setPedaling(double):double
setBraking(double):double
reportDetails():void
reportIfDamageExists():boolean
equals(Bicycle):boolean
-breaks
0..1
IBrakes
bicycleBreaks
getSpeedReduction(Double):double
reportIfBroken(double):boolean
DuperBrakes
bicycleBreaks
DuperBrakes()
getSpeedReduction(Double):double
reportIfBroken(double):boolean
NiceBrakes
bicycleBreaks
NiceBrakes()
getSpeedReduction(Double):double
reportIfBroken(double):boolean
-brakeFactory
0..1
BrakeFactory
bicycleBreaks
BrakeFactory()
construct(String):IBrakes
-pedal
0..1
IPedal
bicyclePedal
getSpeed(double):double
reportIfBroken():boolean
SimplePedal
bicyclePedal
SimplePedal()
getSpeed(double):double
reportIfBroken():boolean
ZuperPedal
bicyclePedal
ZuperPedal()
getSpeed(double):double
reportIfBroken():boolean
PedalFactory
bicyclePedal
PedalFactory()
construct(String):IPedal
-pedalFactory
0..1

```
package tests;
```

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;

@RunWith(Suite.class)
@SuiteClasses({ BicycleTest.class, BrakesTest.class, PedalsTest.class })
public class AllTests {
  //no need to add sth here. The above directives simply run all tests
}
```

```
import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;

import bicycleBody.Bicycle;


public class BicycleTest {

  private Bicycle bicycle;

  @BeforeClass
  public static void setUpBeforeClass() throws Exception { }

  @Before
  public void setUp() throws Exception {
    bicycle = new Bicycle("SimplePedal","DuperBrakes", "Test1");
  }

  @Test
  public final void testBicycleFactoryNull() {
    assertNotNull("After setup, the bicycle is not null", bicycle);
  }


  @Test
  public final void testBrakeCreation() {
    assertEquals("Test if getVelocity() works OK", 0.00, bicycle.getVelocity(),1);
    bicycle.setOriginalVelocity(50);
    assertEquals("Test if getVelocity() works OK", 50.00, bicycle.getVelocity(),1);
    bicycle.setPedaling(10);
    assertEquals("Test if getVelocity() works OK", (50.0+35*10),
                            bicycle.getVelocity(),1);

    //Resets velocity
    bicycle.setOriginalVelocity(50);
    bicycle.setBraking(20);
    assertEquals("Test if getVelocity() works OK", 0.0 , bicycle.getVelocity(),1);
  }

  @Test
  public final void testTimeComputation() {
    bicycle.setOriginalVelocity(10);
    double distance = 1000.0;
    bicycle.computeTime(distance);
    assertEquals(100, (int)bicycle.getTimeRun());
  }

}
```

```java
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNotNull;
import static org.junit.Assert.assertTrue;

import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;

import bicycleBreaks.BrakeFactory;
import bicycleBreaks.DuperBrakes;
import bicycleBreaks.IBrakes;
import bicycleBreaks.NiceBrakes;



public class BrakesTest {

  private BrakeFactory brakeFactory;

  @BeforeClass
  public static void setUpBeforeClass() throws Exception {}

  @Before
  public void setUp() throws Exception {
    brakeFactory = new BrakeFactory();
  }

  @Test
  public final void testBrakesFactoryNull() {
    assertNotNull("After setup, the brakesFactory is not null", brakeFactory);
  }


  @Test
  public final void testBrakeCreation() {
    IBrakes brake = brakeFactory.constructBrake("NiceBrakes");
    assertTrue(brake instanceof NiceBrakes);

    brake = brakeFactory.constructBrake("DuperBrakes");
    assertTrue(brake instanceof DuperBrakes);

    brake = brakeFactory.constructBrake("wrong type");
    assertEquals(brake,null);
  }

}
```

```java
import static org.junit.Assert.*;

import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;

import bicyclePedal.IPedal;
import bicyclePedal.PedalFactory;
import bicyclePedal.SimplePedal;
import bicyclePedal.ZuperPedal;

public class PedalsTest {

  private PedalFactory pedalsFactory;

  @BeforeClass
  public static void setUpBeforeClass() throws Exception {

  }

  @Before
  public void setUp() throws Exception {
    pedalsFactory = new PedalFactory();
  }

  @Test
  public final void testPedalsFactoryNull() {
    assertNotNull("After setup, the pedalsFactory is not null", pedalsFactory);
  }


  @Test
  public final void testBrakeCreation() {
    IPedal brake = pedalsFactory.constructPedal("SimplePedal");
    assertTrue(brake instanceof SimplePedal);

    brake = pedalsFactory.constructPedal("ZuperPedal");
    assertTrue(brake instanceof ZuperPedal);

    brake = pedalsFactory.constructPedal("wrong type");
    assertEquals(brake,null);
  }
}
```

Programming for OO systems (esp., with Java)

Ανάπτυξη Λογισμικού (Software Development)

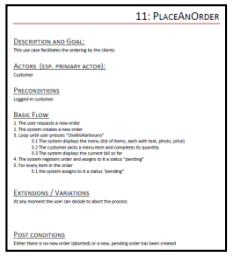
www.cs.uoi.gr/~pvassil/courses/sw_dev/

ΜΥΥ301/ΠΛΥ 308

Σημειώσεις ΠΒ στο βιβλίο “Effective Java” 3rd Ed., by Joshua Bloch
https://github.com/jbloch/effective-java-3e-source-code

Real world problem

Requirements Engineering


11: PLACEANORDER

DESCRIPTION AND GOAL:
This use case facilitates the ordering by the clients.

ACTORS (ESP. PRIMARY ACTOR):
Customer

PRECONDITIONS
Logged-in customer

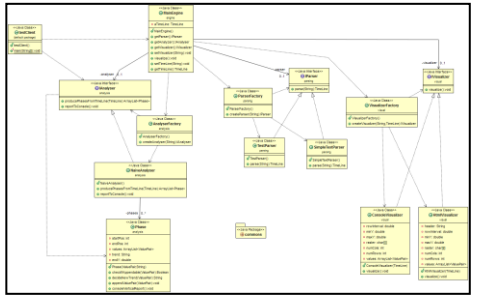
BASIC FLOW
1. The user requests a new order
2. The system creates a new order
3. Loop until user presses "CheckNoMoreItems"
3.1 The system displays the menu (list of items, each with text, photo, price)
3.2 The customer picks a menu item and completes its quantity
3.3 The system displays the current bill so far
4. The system asks for payment and assigns to it a status "pending"
5. For every item in the order
5.1 the system assigns to it a status "pending"

EXTENSIONS / VARIATIONS
At any moment the user can decide to abort the process

POST CONDITIONS
Either there is no new order (aborted) or a new, pending order has been created

Requirements Specification

Design



Design Specification

Implementation

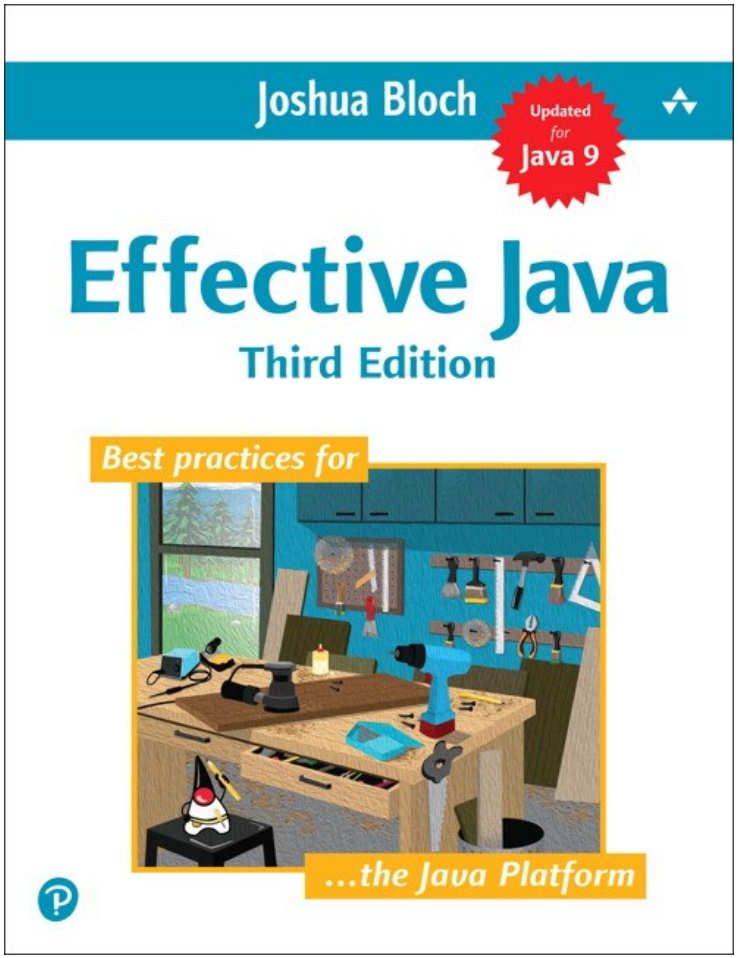
Working code

Testing

Working code with quality assurances

Maintenance

Πηγή



- Εδώ παρέχω μερικές σκέψεις / σημειώσεις στο βιβλίο του Joshua Bloch, Effective Java (3rd ed., 2018)…
- … που θεωρείται ένα από τα πιο σημαντικά βιβλία για τον ορθό προγραμματισμό της γλώσσας Java
- Το Effective Java ΔΕΝ είναι textbook για να μάθει κανείς Java. Δεν είναι καν textbook για τη φιλοσοφία του αντικειμενοστρεφούς προγραμματισμού.
- Ωστόσο, παρέχει συμβουλές για τεχνικά προβλήματα προγραμματισμού σε Java που δεν έχουν εύκολες απαντήσεις, για developers που ήδη κατέχουν τα βασικά.
- Θα επισημάνω μόνο επιλεγμένα στοιχεία του βιβλίου, και όχι με τη σειρά που παρουσιάζονται στο βιβλίο

Το pdf του effective java 3rd edition δεν είναι διαθέσιμο δωρεάν στο διαδίκτυο, αλλά θα συζητήσουμε παραδείγματα στις διαλέξεις

Κεντρικές ιδέες

- Ό,τι επιτρέπεται από τον compiler, δε σημαίνει ότι είναι και απαραίτητα σωστό!
- Είναι σημαντικό να ξέρουμε αν, πού, και πώς πρέπει να χρησιμοποιήσουμε κάθε στοιχείο οποιασδήποτε γλώσσας!
- Επίσης, είναι σημαντικό να μπαίνουμε «στη φιλοσοφία» κάθε οικογένειας γλωσσών προγραμματισμού!

ΚΕΦΆΛΑΙΟ 9: ΓΕΝΙΚΌΣ ΠΡΟΓΡΑΜΜΑΤΙΣΜΌΣ

Java code conventions (#68)

Construct	Form. Syntax	Essence	Example
package	**lowercase**	**ουσιαστικό** που αφορά στα περιεχόμενα του πακέτου	**mainengine, dataload**
Class	**CamelCase**	**Ουσιαστικό** που περιγράφει τι αναπαριστά η κλάση στον πραγματικό κόσμο, ή τι ρόλο έχει στον κώδικα	**MainEngine, DataLoader**
Interface	**CamelCase**	**Επίθετο** (συχνά) ή **ουσιαστικό** που εξηγεί τι ρόλο μπορεί να φέρει εις πέρας όποια κλάση υλοποιεί το interface (frequently starts with an I)	**IReporter, ISortedList**
method	**mixedCase**	**ρήμα** (ενεργητικό) που περιγράφει τι κάνει η μέθοδος. Σύνταξη: **ρήμαΠεριγραφήΑντικειμένου** ΔΕΝ ΒΑΡΙΟΜΑΣΤΕ ΝΑ ΤΟ ΔΩΣΟΥΜΕ ΣΩΣΤΑ!	**getMonetarySum(), produceTotalEuroAmount()**
variable	**mixedCase**	**ουσιαστικό** που περιγράφει επαρκώς το ρόλο του πεδίου στην κλάση Κατ' αντιστοιχία με τις μεθόδους!	**receivedPackages, euroAmountSpent**
CONSTANT	**UPPERCASE**	σαν variable, αλλά για να διακρίνεται ότι είναι σταθερά, όλα κεφαλαία. Συχνά ξεκινά και με _. Το μόνο construct where _ is allowed	**_TOTAL_NUM_OWNERS**

(#68) Element order inside a java file

```java
 * %W% %E% Firstname Lastname
package java.blah;
import java.blah.blahdy.BlahBlah;
/**
 * Class description goes here.
 *
 * @version 1.10 04 Oct 1996
 * @author Firstname Lastname
 */
public class Blah extends SomeClass {
    /* A class implementation comment can go here. */
    /** classVar1 documentation comment */
    public static int classVar1;
    /*** classVar2 documentation comment that happens to be
     * more than one line long
     */
    private static Object classVar2;
    /** instanceVar1 documentation comment */
    public Object instanceVar1;
    /** instanceVar2 documentation comment */
    protected int instanceVar2;
    /** instanceVar3 documentation comment */
    private Object[] instanceVar3;
    /**
     * ...method Blah documentation comment...
     */
    public Blah() {
        // ...implementation goes here...
    }
    /**
     * ...method doSomething documentation comment...
     */
    public void doSomething() {
        // ...implementation goes here...
    }
    /**
     * ...method doSomethingElse documentation comment...
     * @param someParam description
     */
    public void doSomethingElse(Object someParam) {
        // ...implementation goes here...
    }
}
```

package *xxx.xxx*
import *java.xxx ;*
import external.stuff.xxx;
import from.this.prj.xxx;

*/** class comments */*

<visibility> **class** <**ClassName**>{
  {public; ;protected;private} **static** <type><**classVrbl**>;
  {~~public; ;protected;~~ private} <type><**instanceVrbl**>;

  <visibility>**constructor**(){ ...}
  <visibility>**constructor**(...params...){...}

  {public/ /protected/private} **doSth**(...){...}
    /* No particular order for methods:
    "methods should be grouped by functionality rather than by scope or accessibility. ... The goal is to make reading and understanding the code easier." */

}//end class

Package-private has no modifier; denoted via an empty space here

https://www.oracle.com/technetwork/java/codeconventions-150003.pdf

Know thy libraries (#59)

- Millions of “users” of **standard libraries** guarantee a much more debugged, safe, **robust code** than your own
- **Do not re-invent the wheel**; we stand on the shoulders of giants, not their foot toes!

- For Java, essential libraries:
  - `java.lang <-- Types/excep./errors`
  - `java.io   <-- Files`
  - `java.util <-- Collections`
  - `java.util.stream`
  - `java.util.concurrent`
- PV: repeatedly found very useful
  - `java.nio.file`
  - `Apache commons`
- Revisit with each new release (ωραία θα ήταν)!!!

Όχι, δεν τα ξέρω όλα αυτά. Θα έπρεπε όμως!

Minimize the scope of local variables (#57)

- Declare a local variable where it is first used!
- … almost always initialize it!
  - Keep methods short & focused: minimizes the #vrbl's needed
  - Postpone declaration until you know the initial value
  - Special case: prefer `for` to `while` loops: easier to declare a variable inside the for loop (both (i) traditional for-loops and (ii) collection-related

    ```
    for (Element e: collection){ …use e …}
    ```

  - Exception: variables to be used inside `try/catch` blocks

For-each is better than for (#58)

BOTH for collections AND arrays: Use

```
for (Element e : elements) {
     ... // Do something with e …
}
```

.. instead of (the visual clutter of)..

```
for (Iterator<Element> i = c.iterator(); i.hasNext(); ) {
     Element e = i.next(); … // Do something with e …
} or
for (int i = 0; i < a.length; i++) {
     … // Do something with a[i] …
}
```

If time: make a lengthier discussion && also cover exceptions

- Why? Because it allows the essential part of working with collections of similar items to be bluntly obvious to the reader:
  - Iterate through the collection, by visiting one-item-at-a-time;
  - Work with this single item you visit each time;
- And yes, arrays are yet-another case of collection-of-similar-items

Data types: summary

If time: make a lengthier discussion for #61, #62

- (#61) Prefer primitive types to boxed primitives (e.g., use int rather than Integer), wherever possible
- (#60) For monetary types, use `int`, `long` (18 decimal digits) or `BigDecimal` (albeit not primitive && slower) rather than float and double (both known with problems of precision)!
- (#62) Strings should NOT take the place of other data types/enum's/composite objects!
  - Το βλέπω και το ξαναβλέπω όλη την ώρα: φτιάξτε domain classes/enumerations/… όπως πρέπει, αντί να δηλώνετε/επιστρέφετε ένα string για να ξεμπερδεύουμε! Αμάν…
- (#63) Contrary to what the 2018 book says, it is now perfectly OK to use `String s = “’; s += “Extra stuff appended”;`

Prefer Interfaces to Classes (#64)

- (#64) If it is possible to refer to an object via an interface, use the interface (which is a contract) to declare the object, rather than the concrete class
- Instead of (the concrete class)
  - `LinkedHashSet<Son> sonSet = new LinkedHashSet<>();`
- …use (the interface)
  - `set<Son> sonSet = new LinkedHashSet<>();`
- Ideally (granted, this is not always possible):

```
<Interface> vrbl = new <ConcreteClass>();
```

- Why? Because it guarantees public methods && it is absolutely feasible to switch to another implementation if needed
  - PV: which is the basic maintenance strategy for evolving maintaining our code too!!

CHAPTER 4: CLASSES & INTERFACES

(#25) Every file has a single top-level class

- …and maybe some nested auxiliary classes too
- Always put top-level classes in a file that has exactly their name (e.g., class `Person` in file `Person.java`)
- The reasons are mainly, but not solely, due to the need that multiple persons cooperate in a team of developers under the principle of single location of the code, providing:
  1. Understandability of the code by everyone
  2. Easy location of code-of-interest by anyone, when in need, … and…
  3. Avoidance of multiple definitions of the same construct, by different persons in different locations

Encapsulation is paramount (#15, #16)

- All that you make publicly available will be (ab)used by clients you do not control =>
- Access to classes, methods and attributes should be given as sparingly as possible!
  - If the visibility of a class can be made `package-private`, make it so!
  - All attributes should be `private`; if possible, `final` too. This includes composite attributes too: e.g., all arrays should be `private`!
  - Simple `FINAL_STATIC_CONSTANTS` can be `public`; but not if they are collections (as they point to mutable objects).
  - All methods not publicly needed: `private` (in extremis: `protected`) or `package-private` for too cohesive packages

Why attribute encapsulation?

- Why all this fuss (and resulting language verbosity) with making fields private and accessing them via public methods?

- There is also the notion of “consenting adults”: we (are supposed to) know what we are doing with the classes that we use…

Why attribute encapsulation?

- Typically, you prefer objects to be immutable! You do not want your clients to alter the state of your objects (not only in multithreaded environments, but overall).
- [GOOD PRACTICE] The gurus will tell you that typically, the only change in the state of an object should be done via a parameterized constructor.
- Later, if you want to change it, you want to consider the possibility of creating a new version.
- This means: NO SETTERS ALLOWED!

Why on earth no public attributes and no setter methods?

- Multiple composite objects can point to the same object: a change in the state of the object is not necessarily to be propagated to all of them (maybe some need a newer version, maybe some need the old one).

Ok, so we make the attribute private and we disallow setters. What about getters?

- Again: you make a class available, but you do not control who is gonna use it!
- The only way to control the usage is via a PUBLIC API OF METHODS
- Whatever you make public is a “contract” to clients: you are supposed to support it “forever”; if you change it, the syntactic and semantic correctness of the code is in danger!

What about getters?

- So, you allow a method to be public only if you know that the method is providing a service to other parts of the software, outside the class where the method is defined!

- Exactly the same holds for classes and packages! You make a class public, i.e., you promise to support it “forever”, only if it is necessary for providing services to the rest of the system!

(#18) Prefer composition to inheritance

- Strongest possible coupling of two classes
- Not only all properties, but also all flaws of the parent are passed on to the child class
- A small change in the parent class
  - … can break the child class syntactically or semantically
  - … actually changes the child class too (i.e., a change in class C1 implies a change in class C2)
- In Java, specifically, a class can inherit only a single class (although it can implement several interfaces and be composed of several components) => if a class inherits a certain class, no further specialization can be via inheritance (e.g., if you want to implement a combination of {colored, non colored} X {rectangle, triangle, circle}, which is the mama class?)

(#18) Prefer composition to inheritance

- All these problems are avoided if one uses composition
- Inheritance is NOT a friend; overriding is, for specific situations where you can have multiple, alternative implementations of a generic functionality, provided via components implementing the same interface
- Inheritance is appropriate in situations like the Template Method design pattern, where an abstract parent class implementing a sequence of steps can be customized in some of these steps
- In all other cases, make your class a composite class, delegate the work from the composite object to the methods of the appropriate component, and handle their result in the composite class: delegation via composition is the most appropriate method to handle complex tasks (see Decorator, Façade, … design patterns)

(#18) Prefer composition to inheritance (mental checklist)

- To avoid:

ALWAYS: Think HAS-A before IS-A!

  - is a class declared `final` to prohibit subclasses?
- If (inheritance), check:
  - the child IS-A mama in real-world?
  - can we replace IS-A (inheritance) with HAS-A (composition)?
  - Can we avoid `protected` at mama?

If time: make a lengthier discussion for #19, not covered here...

(#21) Design interfaces for posterity

Interfaces are contracts

- An interface `ServiceProvider` specifies a behavior of its implementing classes, say `Srvc1` and `Srvc2`, such that a client class, say `Client`, defines variables of type `ServiceProvider` in its code, and deal with `Srvc1` or `Srvc2`, only at object creation.
- This means that if the implementors are updated, removed, or new implementors are introduced, the code dealing with the service is left untouched – only object creation is affected…

(#21) Design interfaces for posterity

- … being contracts, to guarantee the provision of a service, means that interfaces change rarely, or else client code breaks!

- With interfaces …
  - You can extend them (but must retrofit implementor classes), or introduce new interfaces extending them
  - You can complement them with new interfaces (as implementors can support multiple interfaces)
  - But you cannot easily modify (even worse: delete) them, as both implementors and clients crash!

(#21) Design interfaces for posterity

- Always test introduced interfaces with multiple implementations, ideally from different developers, before introducing them!

- You cannot control your clients in OO! Once a code structure is made publicly available, its providers are bound to support it…

(#20) Prefer Interfaces to Abstract Classes to define contracts of behavior

- Interfaces are more easy to manipulate when code changes:
  - Existing classes can easily be retrofitted to fit an interface; much more difficult to fit an abstract class that passes structure to the child class
  - Classes can implement multiple interfaces; abstract classes constrain the programmer to extend the code only via single-line inheritance
- Abstract classes are much better for Template Method cases, where structure and basic algorithm are predefined, and what is overridden is just steps of an algorithm

CHAPTER 8: METHODS

(#56) Write doc comments for all exposed API elements

- Precede every publicly exported
  - class / interface / …
  - method / constructor / static field…
- with a comment describing the contract between the provider and the client
- Methods (esp., at interfaces) are in the epicenter of this directive!
- You will need all: @param, @return, @throws, … as well as preconditions and postconditions

(#51) Design methods carefully

- Choose names that are (a) understandable, (b) consistent with the dev community (e.g., toString() has specific implied semantics), (c) consistent with the project's conventions, and, (d) not too long
- Try to avoid too many or too few methods. Understandability of the class and the methods is important!
- Wherever possible, the types of the parameters should better be abstractions (ideally: interfaces) rather than concrete classes, wherever possible: Dependency Inversion Principle at work!
  - Typical example: `doSth(List aList)` rather than `doSth(ArrayList aList)`
- Prefer enum's to Booleans (yes, I am guilty as hell)
- Try not to make long parameter lists – can use helper classes to group parameter values [handle with care]

(#49) Check method parameters for validity

- All method parameters should be checked (PV: and tested) for validity.
- Yes, we are all bored to do it. But we must.
- Where to test for validity?
  - Outside of the method, at the caller's site, before the method is called?
  - First thing inside the method's src, before anything else happens?
- This becomes even more pressing when constructor methods are concerned….

- https://www.oracle.com/java/technologies/javase/seccodeguide.html

(#49) Check first thing inside the method and throw an exception

- Throw one of
- IllegalArgumentException,
- IndexOutOfBoundsException,
- NullPointerException
- Caller can catch it and now the method did not proceed
- Observe the check is the 1st thing done, BEFORE anything else

From the book – observe the documentation too!!

```
/**
* Returns a BigInteger whose value is (this mod m). This method
* differs from the remainder method in that it always returns a
* non-negative BigInteger.
*
* @param m the modulus, which must be positive
* @return this mod m
* @throws ArithmeticException if m is less than or equal to 0
*/
public BigInteger mod(BigInteger m) {
if (m.signum() <= 0)
   throw new ArithmeticException("Modulus <= 0: " + m);
    ... // Do the computation
}
```

```
//Alternative for NullPointerException
//Inline use of Java's null-checking facility
this.strategy = Objects.requireNonNull(strategy, "strategy");
```

https://stackoverflow.com/questions/45632920/why-should-one-use-objects-requirenonnull

(#49) Check first thing inside the method and throw an exception

- You need to keep the checks close to the method (ideally inside it), such that the method and its checks are together in the src
- Must make sure that illegal args do not harm the state of the system (e.g., system exit without closing streams, appropriate logging, …)
- Remember to document the @throws

- [PV] **Constructors** are a special case: due to their very specific nature, apart from the internal checks, one can always consider placing checks at factories, before the constructor is invoked.
- … and, yes, they can throw exceptions too, to avoid object creation (but you must catch them at the invocation location, so as to react properly to what happens if the object is not constructed)
- https://stackoverflow.com/questions/30803650/java-how-to-only-create-an-object-with-valid-attributes
- https://stackoverflow.com/questions/1371369/can-constructors-throw-exceptions-in-java?noredirect=1&lq=1

(#52) If possible avoid overloading

- Overloading = same method name with different arguments inside the same class
- Overriding = identical methods at different classes overriding mama/interface method
- Avoid overloading via different names
  - If not possible, use method signatures with different numbers of parameters
    - If not possible, use signatures where you cannot pass the same parameter via casts
      - If not possible, ensure they produce identical behavior
- But basically, avoid it ☺

(#54) Return empty collections instead of null's

- .. to avoid having to check for null

- To avoid paying the price for allocating new empty collections you can invoke sth like

  return Collections.emptyList();

- or similarly.

CHAPTER 9

- ❑ (#68) Stick to language conventions: (a) naming + (b) ordering within a file
- ❑ (#57) Minimize the scope of local variables + declare/[init] a local variable at its first use
- ❑ (#57)(#58) All iterators (even for arrays(!)):

```
for (Element e : elements) {work with e}
```

- ❑ (#61) Prefer primitive types to boxed primitives (e.g., `int` to `Integer`)
- ❑ (#60) For monetary types, use int, long, or BigDecimal ; avoid float &double!
- ❑ (#62) Strings should NOT take the place of other data types / enum's / composite objects! (use enum)
- ❑ (#64) If it is possible declare an object via an interface

```
<Interface> vrbl = new <ConcreteClass>();
```

Construct	Form. Syntax	Essence	Example
package	lowercase	**ουσιαστικό** που αφορά στα περιεχόμενα του πακέτου	`mainengine, dataload`
Class	CamelCase	**Ουσιαστικό** που περιγράφει τι αναπαριστά η κλάση στον πραγματικό κόσμο, ή τι ρόλο έχει στον κώδικα	`MainEngine, DataLoader`
Interface	CamelCase	**Επίθετο** (συχνά) ή **ουσιαστικό** που εξηγεί τι ρόλο μπορεί να φέρει εις πέρας όποια κλάση υλοποιεί το interface (frequently starts with an I)	`IReporter, ISortedList`
method	mixedCase	**ρήμα** (ενεργητικό) που περιγράφει τι κάνει η μέθοδος. Σύνταξη: **ρήμαΠεριγραφήΑντικειμένου** ΔΕΝ ΒΑΡΙΟΜΑΣΤΕ ΝΑ ΤΟ ΔΩΣΟΥΜΕ ΣΩΣΤΑ!	`getMonetarySum(), produceTotalEuroAmount()`
variable	mixedCase	**ουσιαστικό** που περιγράφει επαρκώς το ρόλο του πεδίου στην κλάση Κατ' αντιστοιχία με τις μεθόδους!	`receivedPackages, euroAmountSpent`
CONSTANT	UPPERCASE	σαν variable, αλλά για να διακρίνεται ότι είναι σταθερά, όλα κεφαλαία. Συχνά ξεκινά και με _. Το μόνο construct where _ is allowed	`_TOTAL_NUM_OWNERS`

CHAPTER 4

- ❑ (#25) Every file: a single top-level class
- ❑ (#15, #16) Enforce encapsulation
- If the visibility of a class can be made package `private`, make it so!
- All methods not publicly needed: `private` or `package-private` for too cohesive packages
- All attributes should be `private`; if possible final too. This includes composite attributes too: e.g., all arrays should be private!
- Simple `FINAL_STATIC_CONSTANTS` can be public; but not if they are collections
- ❑ (#17) If possible, enforce immutability
- No setters, just parameterized constructors
- Methods (e.g., getters) return defensive copies of attributes, esp. if collections
- ❑ (#18) Prefer composition to inheritance!
- ALWAYS: Think HAS-A before IS-A!
- Classes declared final (no subclasses)
- If (inheritance), check: (a) the child IS-A mama in real-world? (b) can we replace IS-A with HAS-A? (c) can we avoid protected at mama?
- ❑ (#20, #21) Prefer Interfaces to abstract classes to define contracts of behavior
- design Interfaces for posterity
- Use abs. classes for Template Method

CHAPTER 8

- ❑ (#56) Write doc comments for all publicly exposed API elements
- ❑ (#51) Design methods carefully!
- Choose names that are (a) understandable, (b) consistent with the dev community & project's conventions, and, (d) not too long
- Choose a useful return type (PV)!!
- Avoid too many/few methods in a class
- Wherever possible, parameter types should be abstractions (ideally: interfaces)
- Prefer enum's to Booleans
- Try not to make long parameter lists – can use helper classes to group parameters
- ❑ (#49) Check parameters for validity …
- … first thing inside the method's body
- throw appropriate exceptions (typically `NullPointer`, `IllegalArgument`)
- (PV) For constructors: throwing an exception annules the object construction; the factory can do it too, externally.
- Make sure exceptions do not destroy object state (use defensive copies) or common resources (e.g., IO streams)
- Document @throws & explain why's & when's
- ❑ (#52) Avoid overloading
- ❑ (#54) Return empty collections instead of null's

Short checklist, "Effective Java" 3rd Ed., by Joshua Bloch

Very Short Checklist from the book "Effective Java" 3rd Ed., by Joshua Bloch

Panos Vassiliadis 2022/09/31

1 Language Conventions (#68) [in Greek]

Construct	Form. Syntax	Essence	Example
package	**lowercase**	**ουσιαστικό** που αφορά στα περιεχόμενα του πακέτου	`mainengine, dataload`
Class	**CamelCase**	**Ουσιαστικό** που περιγράφει τι αναπαριστά η κλάση στον πραγματικό κόσμο, ή τι ρόλο έχει στον κώδικα	`MainEngine, DataLoader`
Interface	**CamelCase**	**Επίθετο** (συχνά) ή **ουσιαστικό** που εξηγεί τι ρόλο μπορεί να φέρει εις πέρας όποια κλάση υλοποιεί το interface (frequently starts with an I)	`IReporter, ISortedList`
method	**mixedCase**	**ρήμα** (ενεργητικό) που περιγράφει τι κάνει η μέθοδος. Σύνταξη: **ρήμαΠεριγραφήΑντικειμένου** ΔΕΝ ΒΑΡΙΟΜΑΣΤΕ ΝΑ ΤΟ ΔΩΣΟΥΜΕ ΣΩΣΤΑ!	`getMonetarySum(), produceTotalEuroAmount()`
variable	**mixedCase**	**ουσιαστικό** που περιγράφει επαρκώς το ρόλο του πεδίου στην κλάση Κατ' αντιστοιχία με τις μεθόδους!	`receivedPackages, euroAmountSpent`
CONSTANT	**UPPERCASE**	σαν variable, αλλά για να διακρίνεται ότι είναι σταθερά, όλα κεφαλαία. Συχνά ξεκινά και με _. Το μόνο construct where _ is allowed	`_TOTAL_NUM_OWNERS`

2 Chapter 9: general programming tactics

(#57) Minimize the scope of local variables: Declare a local variable where it is first used + almost always initialize it!

(#57) Prefer *for* to *while* loops, esp: for (Element e: collection){ …use e …}

(#58) Prefer *for-each* to simple *for*: `for (Element e : elements) {work with e}` even for arrays!

(#61) Prefer primitive types to boxed primitives (e.g., use `int` rather than `Integer`), wherever possible

(#60) For monetary types, use int, long (18 decimal digits) or BigDecimal (albeit not primitive && slower) rather than float and double (both known with problems of precision)!

(#62) Strings should NOT take the place of other data types/enum's/composite objects!

(#64) If it is possible to refer to an object via an interface, use the interface (which is a contract) to declare the object, rather than the concrete class. `<Interface> vrbl = new <ConcreteClass>();`

3 Chapter 4: classes and interfaces

(#25) Every file has a single top-level class (and maybe some nested auxiliary classes too)

(#15, #16) Encapsulation is paramount for OO software

- If the visibility of a class can be made `package`, make it so!
- All attributes should be `private`, if possible final too!
- This includes composite attributes too: e.g., all arrays should be `private`!
- Simple `FINAL_STATIC_CONSTANTS` can be `public`; but not if they are collections (as they point to mutable objects).
- All methods not publicly needed: `private`! (in extremis: `protected`), or `package-private` for too cohesive packages

(#18) Prefer composition to inheritance! Think HAS-A before IS-A ALWAYS!

  - To avoid: is a class declared final to prohibit subclasses?

- If (inheritance), check:

  - the child IS-A mama in real-world?
  - can we replace IS-A (inheritance) with HAS-A (composition)?
  - can we avoid protected at mama?

(#20, #21) Prefer Interfaces to abstract classes to define contracts of behavior && design them for posterity

4 Chapter 8: methods

(#51) Design methods carefully!

- Choose names that are (a) understandable, (b) consistent with the dev community and the project's conventions, and, (d) not too long
- Try to avoid too many or too few methods.
- Wherever possible, the types of the parameters should be abstractions (ideally: interfaces)
- Prefer enum's to Booleans
- Try not to make long parameter lists – can use helper classes to group parameter values

(#56) Write doc comments for all exposed API elements

(#49) Check parameters for validity first thing inside the method's body

- Can throw appropriate exceptions (typically NullPointer, IllegalArgument) & document them, esp., their why's & when's; Make sure exceptions do not destroy object state (use defensive copies) or common resources (e.g., IO streams)
- (PV) For constructors: throwing an exception annules the object construction; the factory can do it too, externally.

(#52) Avoid overloading

(#54) Return empty collections instead of null's