

Fondamenti di Informatica T2

Lab12 - Agenda Contatti

Corso di Laurea in Ingegneria Informatica

Anno accademico 2021/2022

Prof. ROBERTA CALEGARI

Prof. AMBRA MOLESINI

Dipartimento di Informatica – Scienza e Ingegneria (DISI)



Agenda Contatti - Requisiti

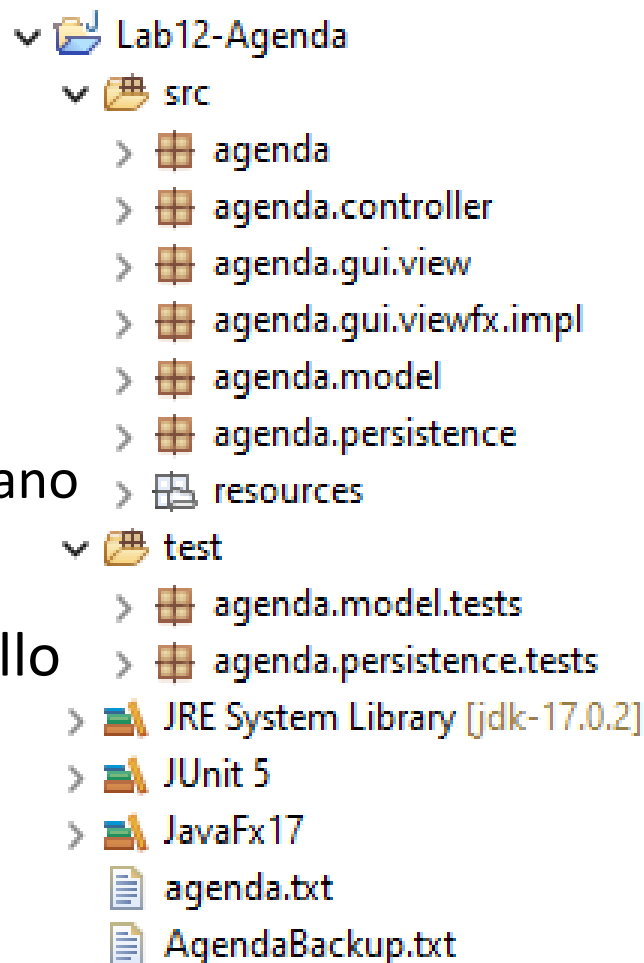
- Si desidera realizzare un'applicazione per la gestione di contatti.
- Ogni **contatto** è caratterizzato da:
 - nome e cognome
 - un insieme di dettagli
- I **dettagli** possono essere di diverso tipo
 - telefono, email, indirizzo, ecc.
 - un dettaglio è **sicuramente** caratterizzato da un **nome** (Phone, Email, Address, ecc.) e da una **descrizione** (Cell1, Cell2, Home, Job, ecc.)
- L'applicazione **deve** essere realizzata in modo che sia **semplice aggiungere un nuovo tipo di dettaglio**

NOTA: un contatto **può contenere più dettagli dello stesso tipo** (es. due numeri di telefono)

È un requisito!!!

Agenda Contatti

- L'applicazione sarà strutturata seguendo il pattern MVC
- In particolare ci focalizzeremo sulla progettazione e sulla realizzazione di:
 - **Modello:** le classi che contengono i dati
 - **Logica di persistenza:** le classi che si occupano di salvare e recuperare i dati
- Il controller e la View li troverete pronti nello Start Kit
- Nello Start Kit troverete anche i test pronti



Il modello

- Quali sono le entità in gioco?
 - **Agenda**
 - **Contatto**
 - **Dettaglio di contatto** – in tutte le sue varianti
- Un'agenda è un insieme ordinato di contatti
- Un contatto può essere **composto** di tanti dettagli
 - **composizione**
- Un dettaglio è modellato tramite un'astrazione realizzata in diverse varianti
 - **ereditarietà** o **implementazione d'interfaccia**?

Detail

Ereditarietà o implementazione d'interfaccia?

Un 'dettaglio' è caratterizzato da:

- un **nome** che dipende dal tipo di dettaglio
 - a livello di astrazione è solo una dichiarazione
- una **descrizione**
 - esiste per ogni dettaglio



Detail è una **classe astratta** con

- due metodi (**getNome** e **getValues**) astratti
- due metodi *accessor* **get/setDescrizione** concreti

Detail

Package agenda.model

abstract getName

restituisce il nome del tipo di contatto

abstract getValues

una rappresentazione in stringa adatta ad essere mostrata su una sola riga (da non confondere con toString):

restituisce in formato stringa tutto ciò che caratterizza un dettaglio (a meno di nome e descrizione che invece sono incluse nella toString)

<i>Detail</i>	
-	description: String
+	getDescription(): String
+	getName(): String
+	getValues(): String
+	setDescription(String): void
+	toString(): String



Sottoclassi di Detail

- **Phone**: memorizza numeri telefonici
- **EMail**: memorizza indirizzi email
- **Address**: memorizza indirizzi

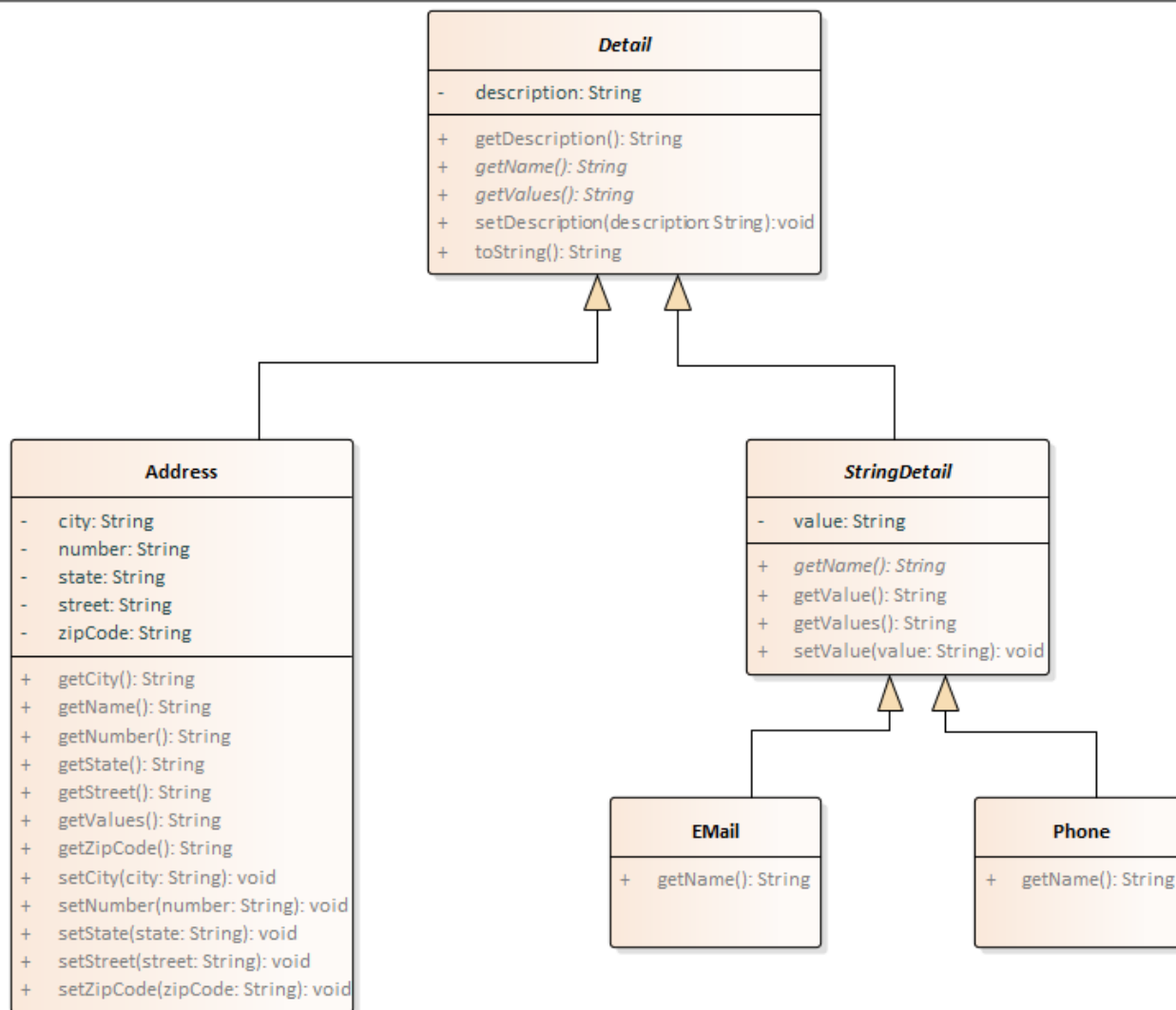
- **Phone e EMail**

- ereditano le caratteristiche di **Detail** (nome, descrizione) ed “aggiungono” un valore stringa (telefono e mail)
- evitare replicazione di codice introducendo una classe astratta **StringDetail** che contiene un valore stringa e da cui ereditino le classi con questa caratteristica (non implementa ancora **getName()**)

- **Address**

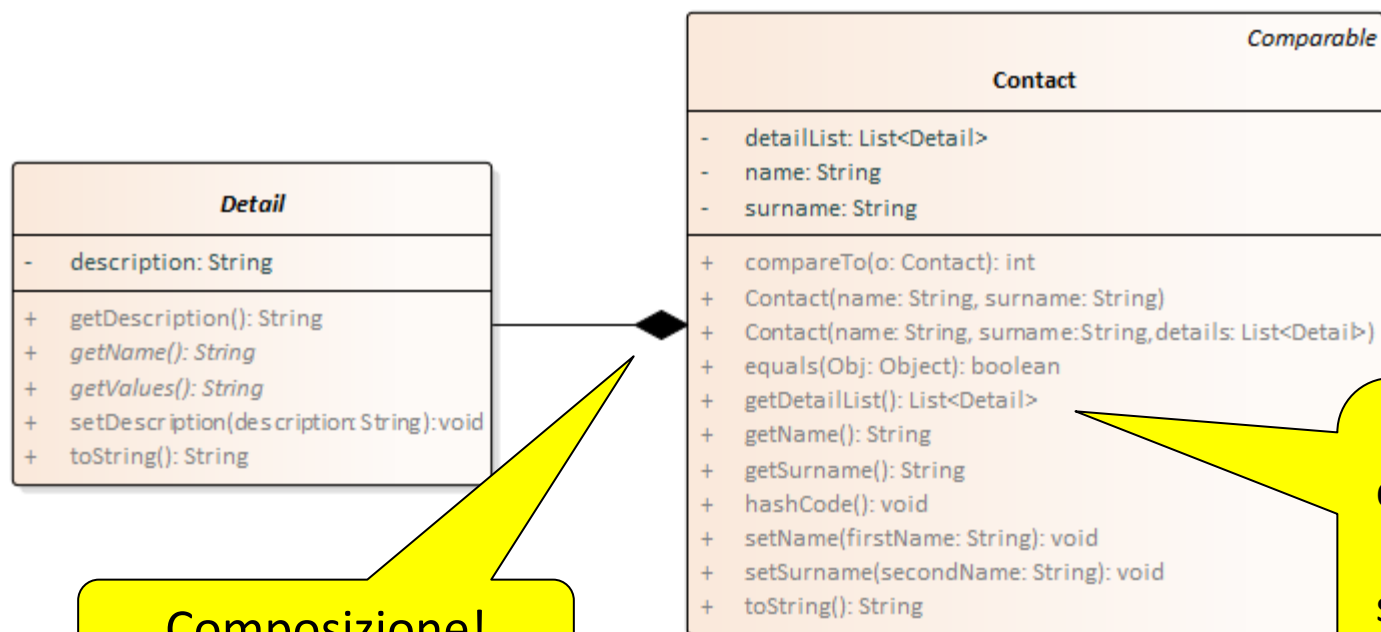
- eredita da **Detail** ed aggiunge le caratteristiche del caso

Gerarchia Dettagli



Contact

- Un **Contact** è caratterizzato da:
 - nome e cognome;
 - un **elenco** di dettagli.
- Deve essere **Comparable**, perché agenda è ordinata



Composizione!

L'aggiunta/rimozione di dettagli può essere fatta direttamente sulla lista ottenuta da getDetailList()

Agenda

Per terminare il modello manca l'ultimo tassello

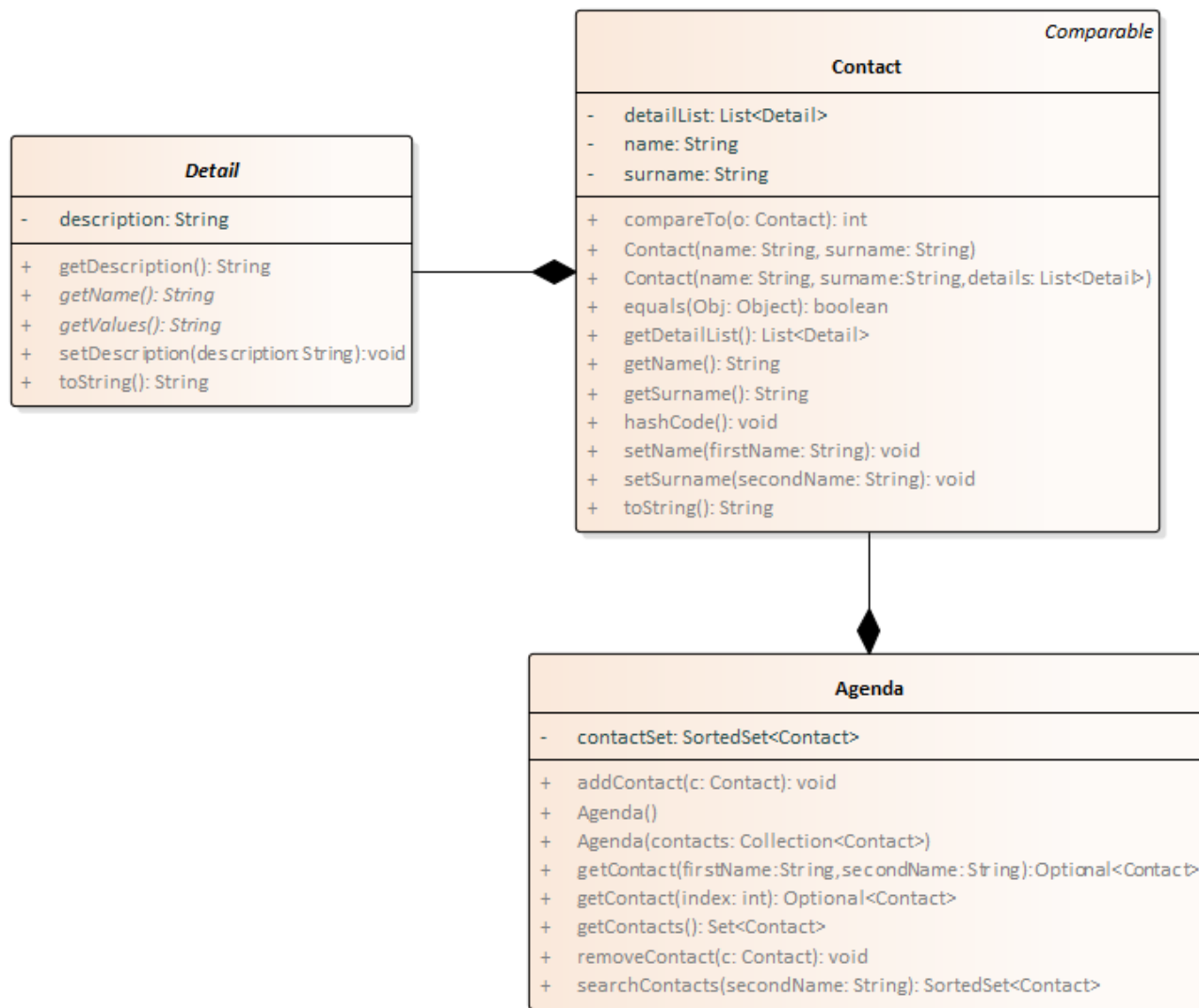
- L'**Agenda** è una «collezione di» **Contact**:
che tipo di collezione?
- L'Agenda è definita come un insieme ordinato (alfabeticamente) di **Contact** senza duplicati → usiamo un **SortedSet**

Agenda

- Due costruttori:
 - Costruttore di default senza parametri
 - Costruttore che riceve una **Collection** di **Contact**
- Diversi metodi accessor
 - ***getContacts()***: restituisce una copia dell'insieme dei **Contact**
 - ***getContact(String name, String surname)***: restituisce un **Optional<Contact>** con quel nome e quel cognome
 - ***getContact(int index)***: restituisce un **Optional<Contact>** nella posizione index
- Un metodo ***addContact*** e un metodo ***removeContact*** per aggiungere e rimuovere un contatto dall'Agenda
- Un metodo ***searchContacts(String surname)*** che ricerca tutti i contatti con un dato cognome

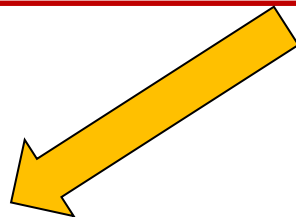
Agenda	
-	contactSet: SortedSet<Contact>
+	addContact(c: Contact): void
+	Agenda()
+	Agenda(contacts: Collection<Contact>)
+	getContact(firstName:String,secondName:String):Optional<Contact>
+	getContact(index: int): Optional<Contact>
+	getContacts(): Set<Contact>
+	removeContact(c: Contact): void
+	searchContacts(secondName: String): SortedSet<Contact>

Agenda



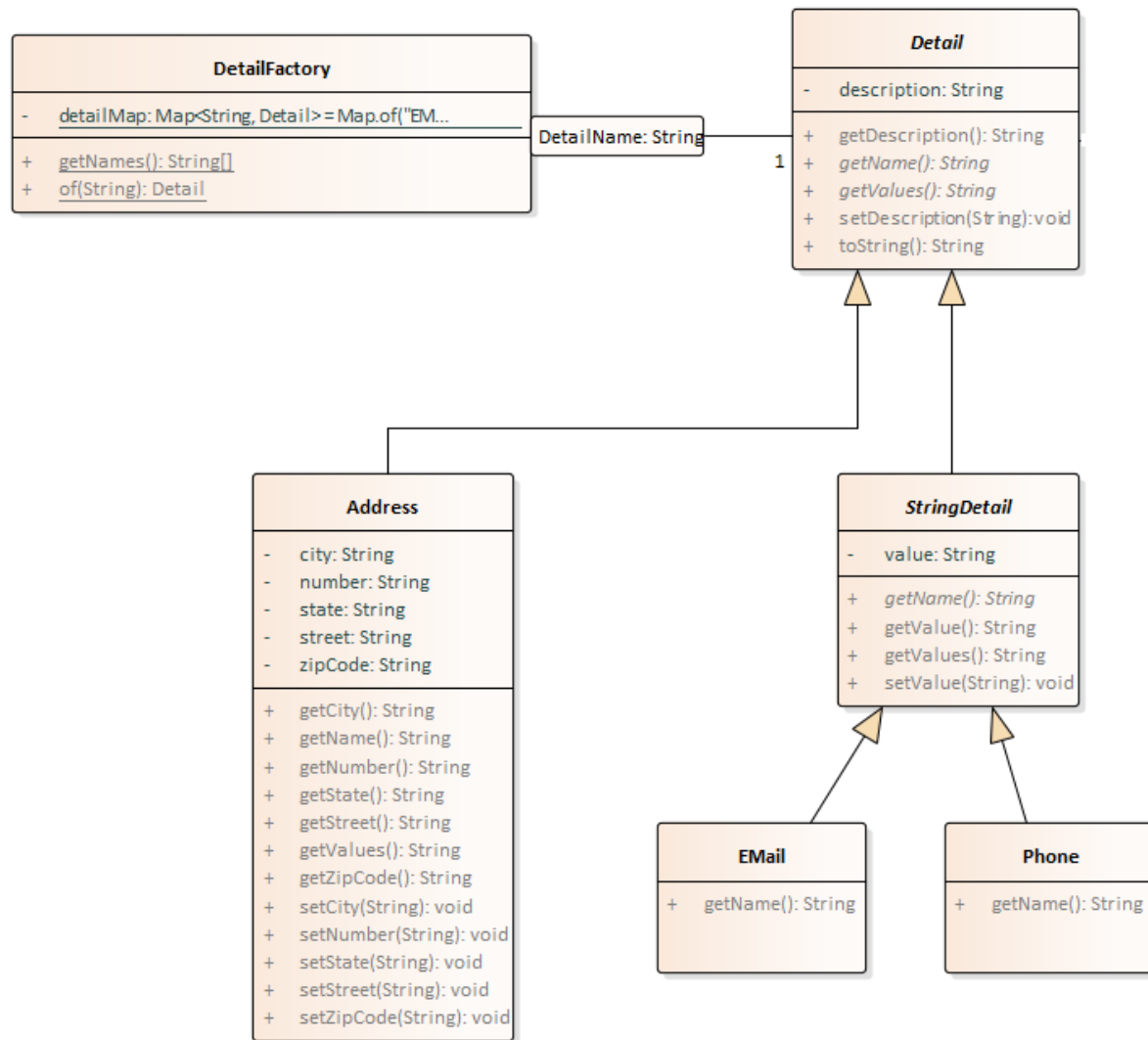
Modello

- Tutto a posto?
- Quasi... ci manca da considerare uno dei requisiti:
«L'applicazione **deve** essere realizzata in modo che sia **semplice aggiungere un nuovo tipo di dettaglio**»

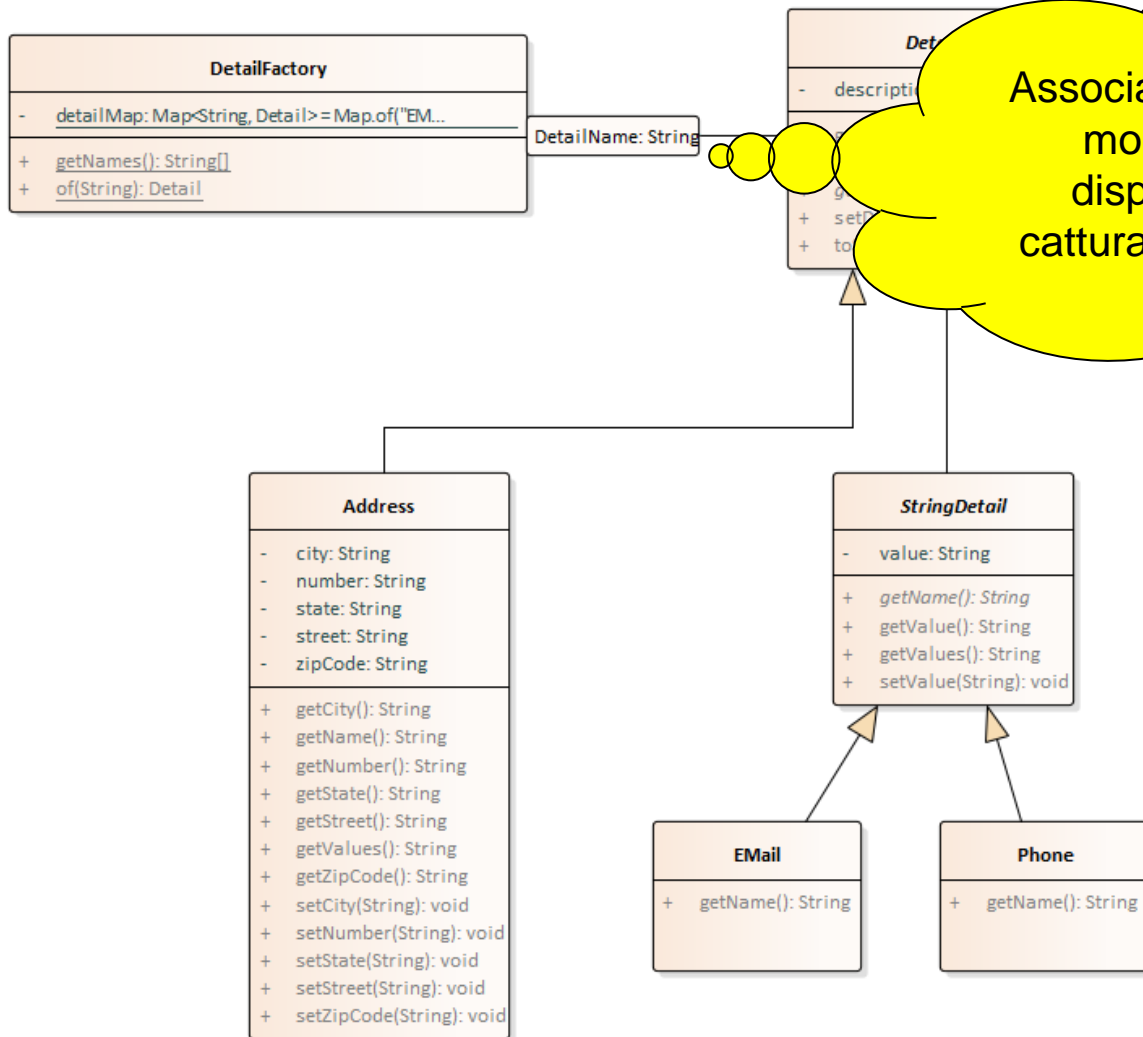


- Significa che:
 - opportuno **mascherare in una factory** la creazione dello specifico (sotto)tipo di dettaglio, dato il nome del **Detail** desiderato
 - una volta ottenuto il **Detail**, lo si può usare per memorizzare i dati relativi a quello specifico dettaglio

DetailFactory

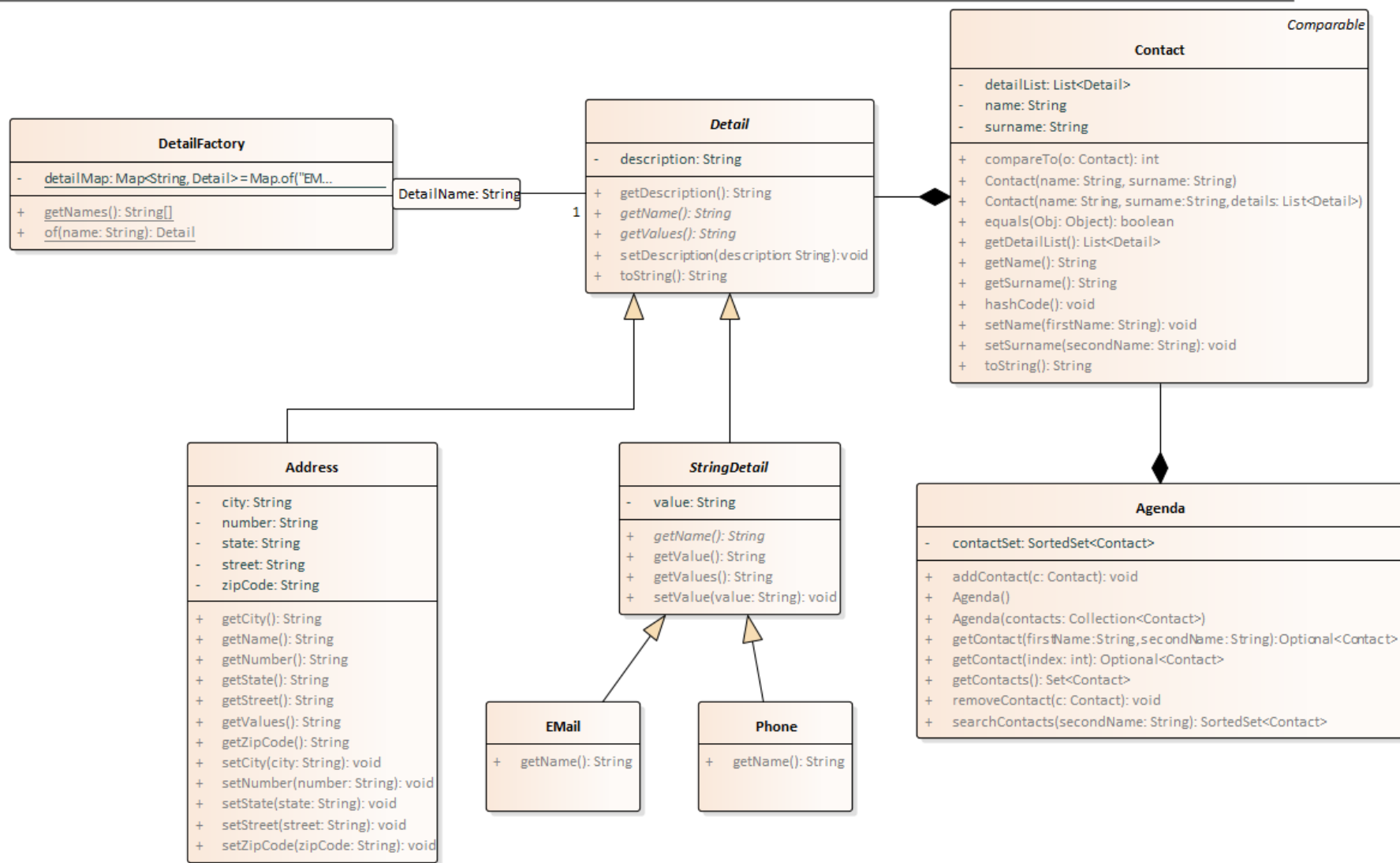


DetailFactory



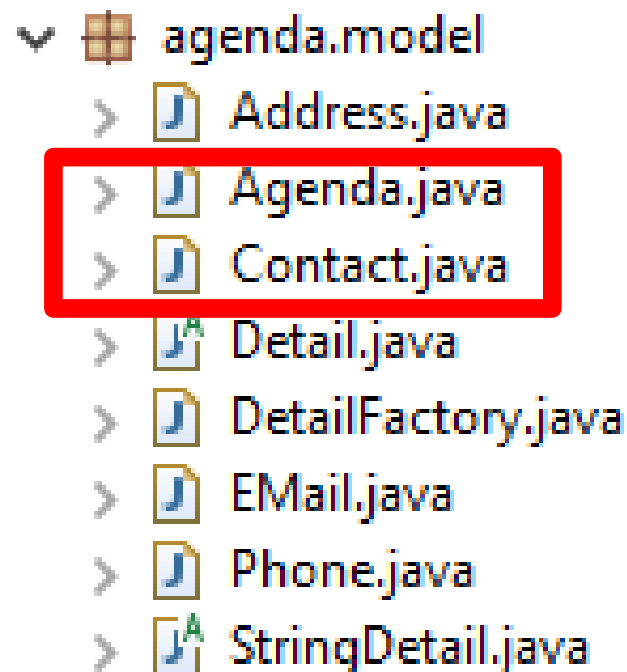
Associazione qualificata: è il modo che ci mette a disposizione UML per catturare il concetto di Map

Il modello completo

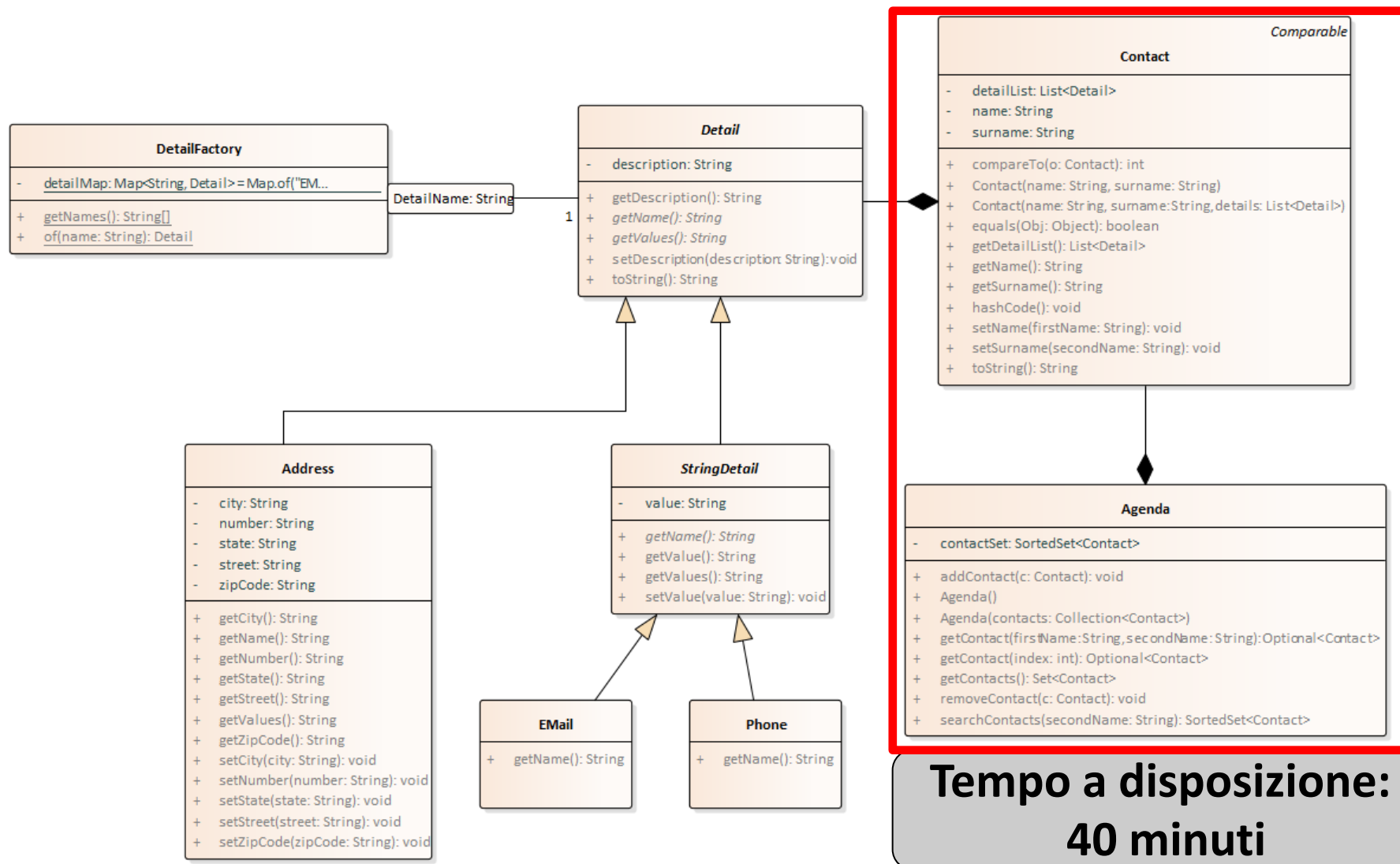


Realizzazione

- **Realizzazione** delle classi dati (package **agenda.model**)
 - **Agenda**
 - **Contact**
- Pronti nello Start Kit:
 - **Address**
 - **DetailFactory**
 - **Detail**
 - **Email**
 - **Phone**
 - **StringDetail**
 - **Test**



Il model completo



**Tempo a disposizione:
40 minuti**

Logica di persistenza



Persistenza su stream di testo

- Abbiamo a che fare con **composizione** ed **ereditarietà**
- È bene **partire con la definizione di un formato dei dati di facile lettura**
- Formattazione di contatto e dettagli: due possibilità:
 1. Tutto su una unica riga
 2. Su righe diverse
- L'ipotesi 1 è un ottimo esercizio per casa...
- Proviamo la **strada indicata dall'ipotesi 2**

Contact: formato

- Obiettivo secondario (ma non troppo) è ottenere un formato che sia *facilmente leggibile*
- Per semplificare la lettura, occorre introdurre dei marcatori di INIZIO e FINE contatto:

StartContact

EndContact

- delimitano tutte le informazioni relative ad un contatto
- compresi i dettagli



Contact: formato

- Ogni contatto inizia con il marcatore **StartContact**
- Alla riga successiva, *name* e *surname* → separati da ‘;’
- segue l'*elenco dei dettagli*, uno per riga
- Infine il contatto termina con il marcatore **EndContact**
- Esempio:

StartContact

Roberta;Calegari

Dettaglio

Dettaglio

...

EndContact



Detail: formato

- I dettagli possono essere di diverso tipo
 - quindi, per comodità, è bene mettere in testa il **tipo** del dettaglio (Phone, EMail, Address, ecc.) così da identificarlo subito
 - segue la **descrizione** perché esiste per tutti i tipi di dettagli
 - infine, le **caratteristiche del tipo specifico di dettaglio** perché si differenziano da un tipo di dettaglio all'altro
 - Esempio:
`Phone;Cell.;333122334`
`Phone;Home;0544881122`
`EMail;@Job;roberta.calegari@unibo.it`
`EMail;@Home;robbyXX@gmail.com`
`Address;Home;Giardini del Pesco;7;40100;Bologna;Italia`



Contact: formato completo

StartContact

Roberta;Calegari

Phone;Cell.;355667788

Phone;Home;0511234567

EMail;@Job;roberta.calegari@unibo.it

EMail;@Home;robbyXX@gmail.com

Address;Home;Giardini del Pesco;7;40121;Bologna;Italia


Address;Job;Viale del Risorgimento;2;40136;Bologna;Italia

EndContact

La lettura: algoritmo

Lettura del singolo **Contact**

1. Leggere la prima riga e verificare che **sia esattamente** "StartContact" – altrimenti si segnala errore
2. Leggere la seconda riga e verificare che **contenga nome e cognome nel giusto formato** – per farlo, incapsularla in uno `StringTokenizer` opportunamente configurato
 1. devono esserci due **token** che vanno inseriti, rispettivamente, in *name* e *surname* del contatto
 2. se non ci sono due **token**, errore
3. Leggere tutti i dettagli del contatto
 - come farlo?



Magari tramite un altro metodo...

La lettura: algoritmo

Lettura dei vari **Detail**

- i dettagli possono essere in numero variabile (anche nessuno)
- occorre quindi una *struttura ciclica* che li legga uno alla volta
 - facendo attenzione a quando compare il marcatore "**EndContact**"
 - che potrebbe anche essere subito all'inizio (in caso non ci siano dettagli)

Loop {

- leggere una riga e verificare se per caso sia "**EndContact**"
- se **NON** lo è, la riga rappresenta un dettaglio → va elaborata
 - a. incapsularla in un opportuno **StringTokenizer**
 - b. In base al primo *token* (che specifica il sottotipo di dettaglio) *effettuare la lettura del dettaglio in base alle sue specifiche caratteristiche attese*

}



La scrittura

- La scrittura è duale alla lettura
 - anzi, è più facile poiché non ci sono problemi di conversione

Persistenza: Design

Persistenza: architettura

- La persistenza può essere "riassunta" nel concetto di **ContactsPersister**, un'interfaccia che dichiara la capacità di:
 - leggere un elenco di **Contact** da un **Reader**
 - scrivere un elenco di **Contact** su un **Writer**



- In fase di lettura (**load**), eventuali problemi:
 - di formato vengono segnalati con l'eccezione **BadFileFormatException**
 - di input/output vengono segnalati con l'eccezione **IOException**



Persistenza: punti di attenzione

- Requirement: **semplice aggiungere nuovi tipi di dettaglio**
→ la lettura/scrittura di un dato tipo di dettaglio **non può essere discriminata con una catena di if o uno switch**

ciò implicherebbe *cablare nel codice l'elenco dei tipi di dettagli ammessi*, quindi per aggiungerne un altro occorrerebbe *modificare il codice esistente* → questa NON è estendibilità!

- **Estendibilità (vera)** significa che *l'aggiunta di una nuova funzionalità costa solo l'aggiunta della nuova classe che la implementa*
- Serve una nuova astrazione che mascheri e incapsuli la gestione dei dettagli → **DetailPersister**
 - una nuova interfaccia con molte possibili concretizzazioni diverse

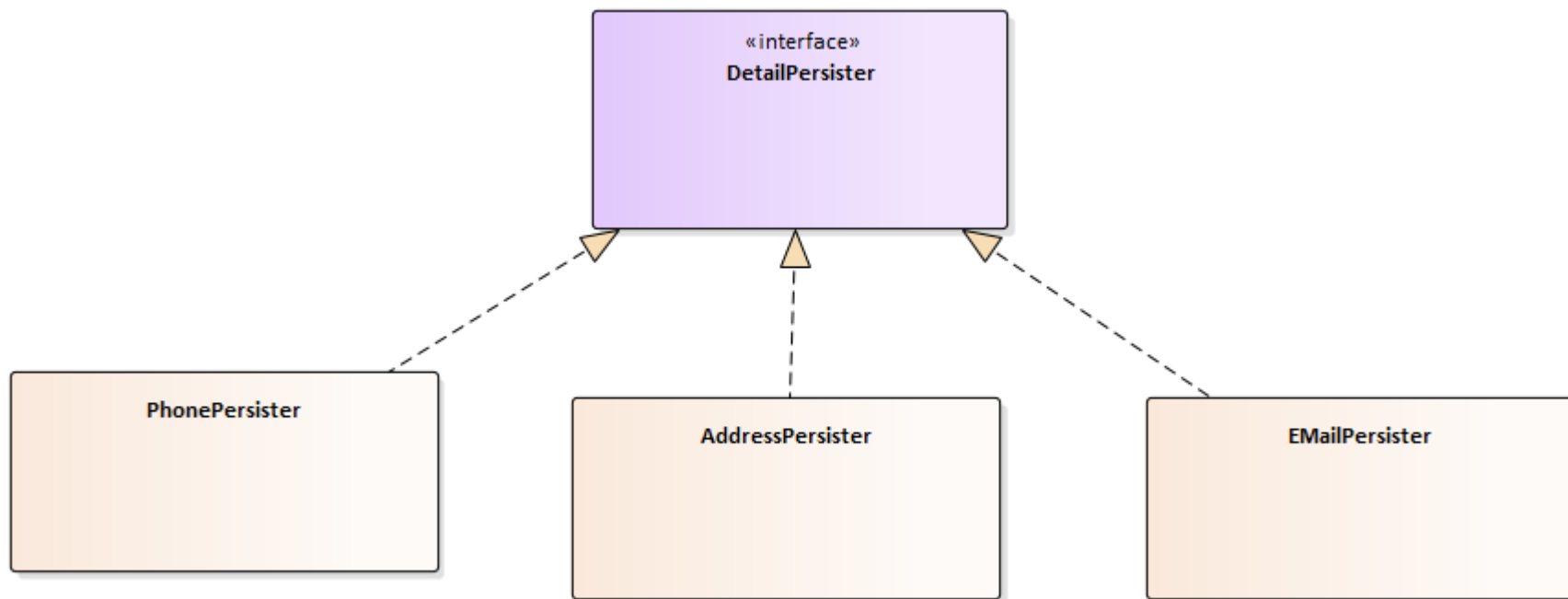


DetailPersister

- IDEA: la persistenza del generico dettaglio è gestita da un **DetailPersister**
 - ogni specifico tipo di **Detail** avrà il suo **DetailPersister**, che sarà *il solo a conoscerne i dettagli interni*
- Così, aggiungere un nuovo dettaglio implica semplicemente:
 - aggiungere il nuovo **Detail**
 - aggiungere il corrispondente **DetailPersister**e tutto continua a funzionare ! 😊

DetailPersister

- Per ora sono previsti *tre tipi di possibili dettagli*
 - indirizzo, email, telefono
- Quindi, prevediamo tre specifici **DetailPersister**



DetailPersister

- **Durante le lettura, come si fa a capire quale persister usare?**
 - il formato del file è stato studiato appositamente!
 - guarda caso, il **primo token** di ogni dettaglio è proprio il **tipo** 😊
→ da quello si può scegliere il persister da utilizzare
 - peraltro, avere in mano il primo token significa anche avere in mano lo **StringTokenizer** da cui è stato estratto
- **Ma cos'è esattamente un DetailPersister ?**
 - è un'astrazione in grado di:
 - **leggere** un dettaglio da uno **StringTokenizer** → **load**
 - **scrivere** un dettaglio su uno **StringBuilder** → **save**
 - NB: la scelta di scrivere su una stringa anziché su un **Writer** è dettata da un'esigenza di simmetria: leggi da un *tokenizer* e scrivi su un **Writer**? Allora perché non hai letto da un **Reader**?

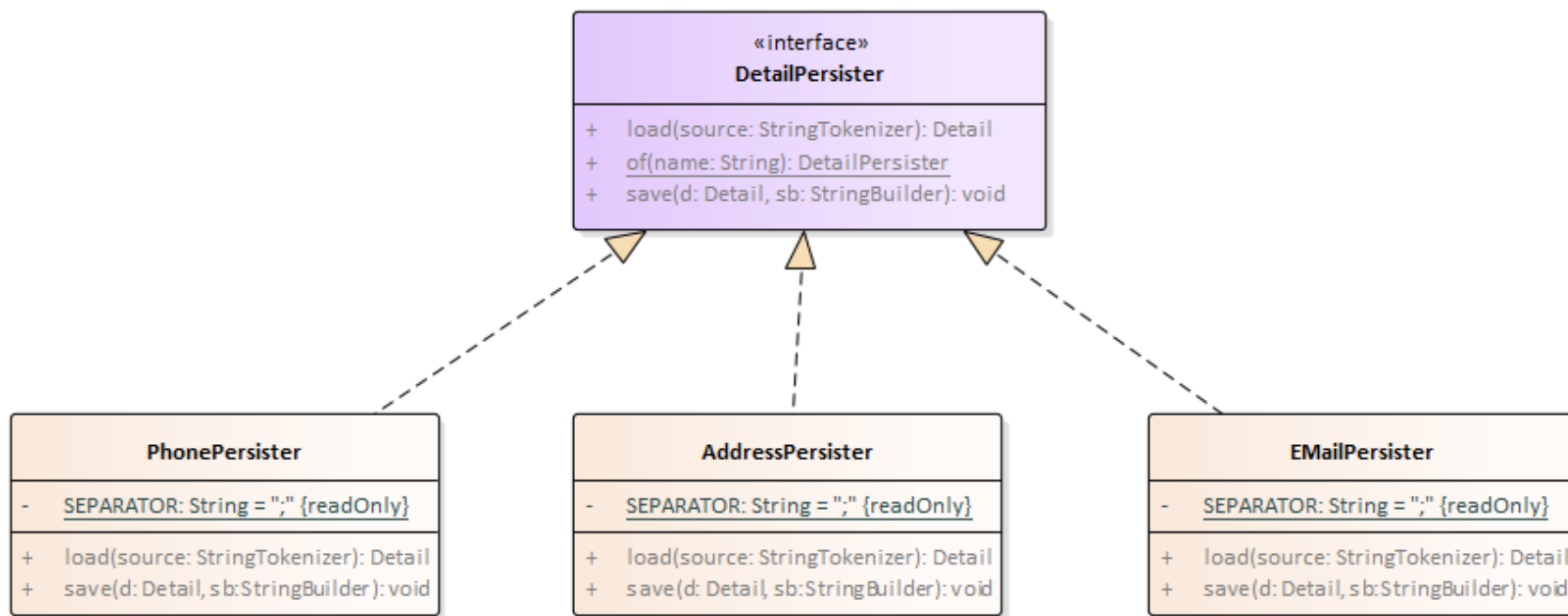
```
«interface»
DetailPersister

+ load(source: StringTokenizer): Detail
+ of(name: String): DetailPersister
+ save(d: Detail, sb: StringBuilder): void
```

DetailPersister

- Eccezioni

- Poiché un **DetailPersister** non lavora con l'input/output non dichiara di lanciare **IOException**
- Però può avere a che fare con errori di lettura per un formato inatteso: quindi nel metodo **load**, lancia una **BadFileFormatException**



Algoritmo di lettura affinato

Rivediamo l'algoritmo di lettura alla luce del **DetailPersister**

Loop {

- leggere una riga e verificare se per caso sia "**EndContact**"
- se NON lo è, la riga rappresenta un dettaglio → va elaborata
 - a. incapsularla in un opportuno **StringTokenizer**
 - b. in base al primo *token* (che specifica il sottotipo di dettaglio) *effettuare la lettura del dettaglio in base alle sue specifiche caratteristiche attese*

}

- Occorre costruire il *giusto persister* in base al tipo di dettaglio
- Al solito, se facessimo direttamente la **new** dovremmo *riempire il codice di catena di if o switch per distinguere i casi* → **NO!**
- Qui vuole una **factory** che incapsuli la logica di creazione



Algoritmo di lettura affinato

- La factory
 - dato il nome del dettaglio
 - crea e restituisce il corrispondente **DetailPersister**
- Nella lettura:
 - ci si fa fare dalla factory il "giusto" **DetailPersister**
 - gli si delega la lettura dei vari token dallo **StringTokenizer**
 - si ottiene in uscita il *giusto Detail*



La Factory

- Dato il nome di un tipo di dettaglio, il metodo `factory of` restituisce il giusto **DetailPersister**

DetailPersister of(String name)

- Per realizzarlo senza infarcire il codice di **if** e **switch**, un modo furbo, che minimizza il gap astrazione/implementazione, può essere una **mappa dettaglio → persister**
 - **Map<String, DetailPersister>**
 - fattibile perché i vari **DetailPersister** non hanno stato
→ possibile crearli una volta per sempre all'inizio, e non toccarli più
 - è una **mappa immutabile** perché non possono nascere nuovi tipi di dettaglio "dinamicamente", dal nulla → costruibile con **Map.of**

La Factory

- Definizione diretta di mappa immutabile:

```
Map<String, DetailPersister> persisterMap =  
    Map.of(  
        "EMail", new EMailPersister(),  
        "Phone", new PhonePersister(),  
        "Address", new AddressPersister());
```




- Alternativa: caricamento mappa "vecchio stile":

```
persisterMap.put("EMail", new EMailPersister());  
persisterMap.put("Phone", new PhonePersister());  
persisterMap.put("Address", new AddressPersister());  
...
```

La Factory

- Chi predispone questa factory?
 - se è statica, nessuno – esiste già
 - se è un tipo di dato, qualcuno deve crearne l'istanza – il main?
 - internalizzata, qualcuno la deve internalizzare – DetailPersister?
- Cosa scegliamo di fare?
 - se optiamo per la semplicità, sicuramente **la prima**
DIFETTO: da una classe statica non si può un domani ereditare...
 - se optiamo per l'estendibilità futura, probabilmente **la seconda**
VANTAGGIO: un domani, se volessimo una factory più specifica, potremmo averla facilmente – basterebbe una sottoclasse!
 - estendibilità futura e semplicità, probabilmente la **terza**
VANTAGGIO: basta modificare l'interfaccia senza dover aggiungere una nuova sottoclasse

La Factory

- Chi predispone questa factory?
 - se è statica, nessuno – esiste già
 - se è un tipo di dato, qualcuno deve crearne l'istanza – il main?
 - internalizzata, qualcuno la deve internalizzare – DetailPersister?
- Cosa scegliamo di fare?
 - se optiamo per la semplicità,
DIFETTO: da una classe statica
 - se optiamo per l'estendibilità futura, probabilmente la **seconda**
VANTAGGIO: un domani, se volessimo  una factory più specifica, potremmo averla facilmente – bastorebbe una sottoclasse!
 - estendibilità futura e semplicità,  probabilmente la **terza**
VANTAGGIO: basta modificare l'interfaccia senza dover aggiungere una nuova sottoclasse

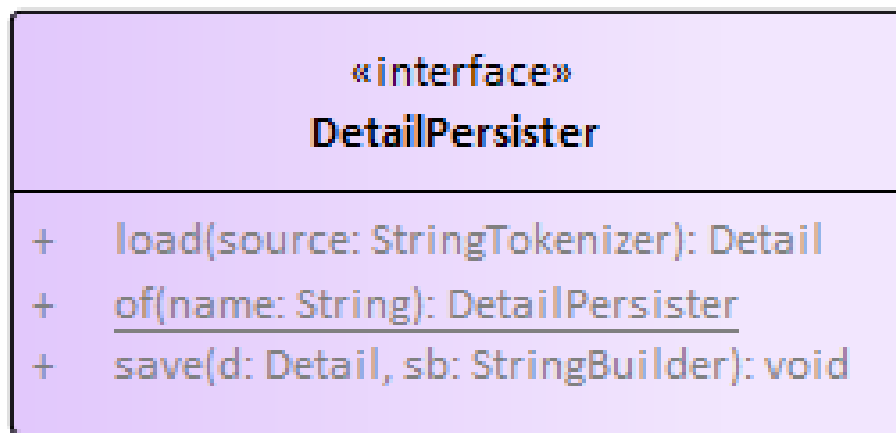
design for change ☺

→ ci piace di più la terza
soluzione

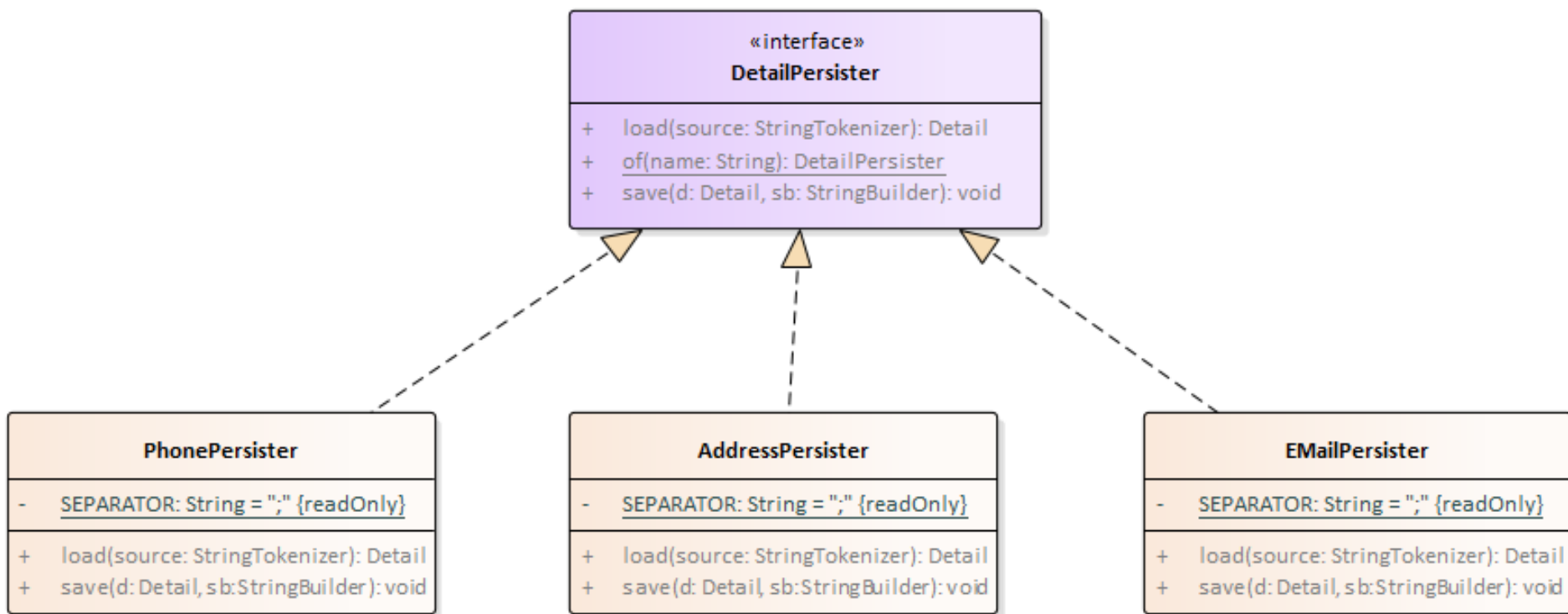


DetailPersister come Factory

- **DetailPersister** fornisce una factory internalizzata attraverso il metodo statico **of**



Detail Persistence Model





Scrittura di un Detail

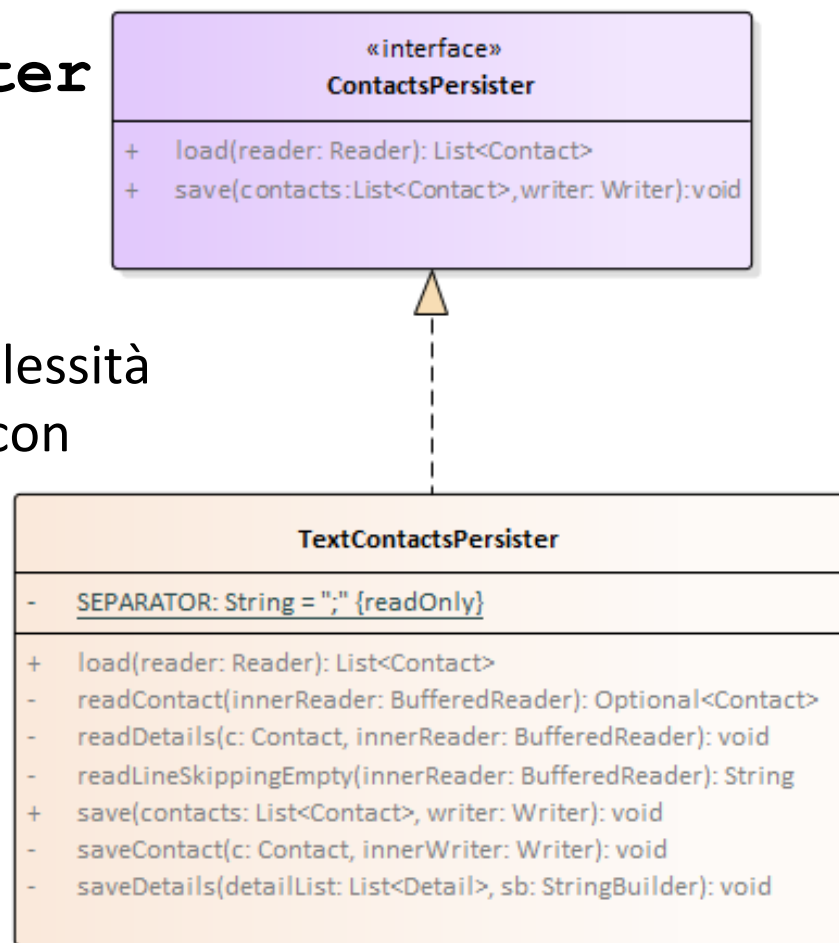
Il metodo **save** di un **DetailPersister** prende in ingresso il **Detail** da salvare e uno **StringBuilder**

- Le caratteristiche del **Detail** devono essere inserite nello **StringBuilder** avendo cura di utilizzare gli stessi formati e separatori usati per la lettura
- Dopo l'esecuzione del metodo, colui che invoca **save** di **DetailPersister** si aspetta di trovare nello **StringBuilder** i dati opportunamente formattati del **Detail**...
- Procedendo per tutti i dettagli di un contatto (ognuno deve ovviamente essere salvato dal proprio **DetailPersister**), nello **StringBuilder** ci sarà l'elenco di tutti i contatti nel formato corretto...
- ...pronto per essere scritto **direttamente** sul file.

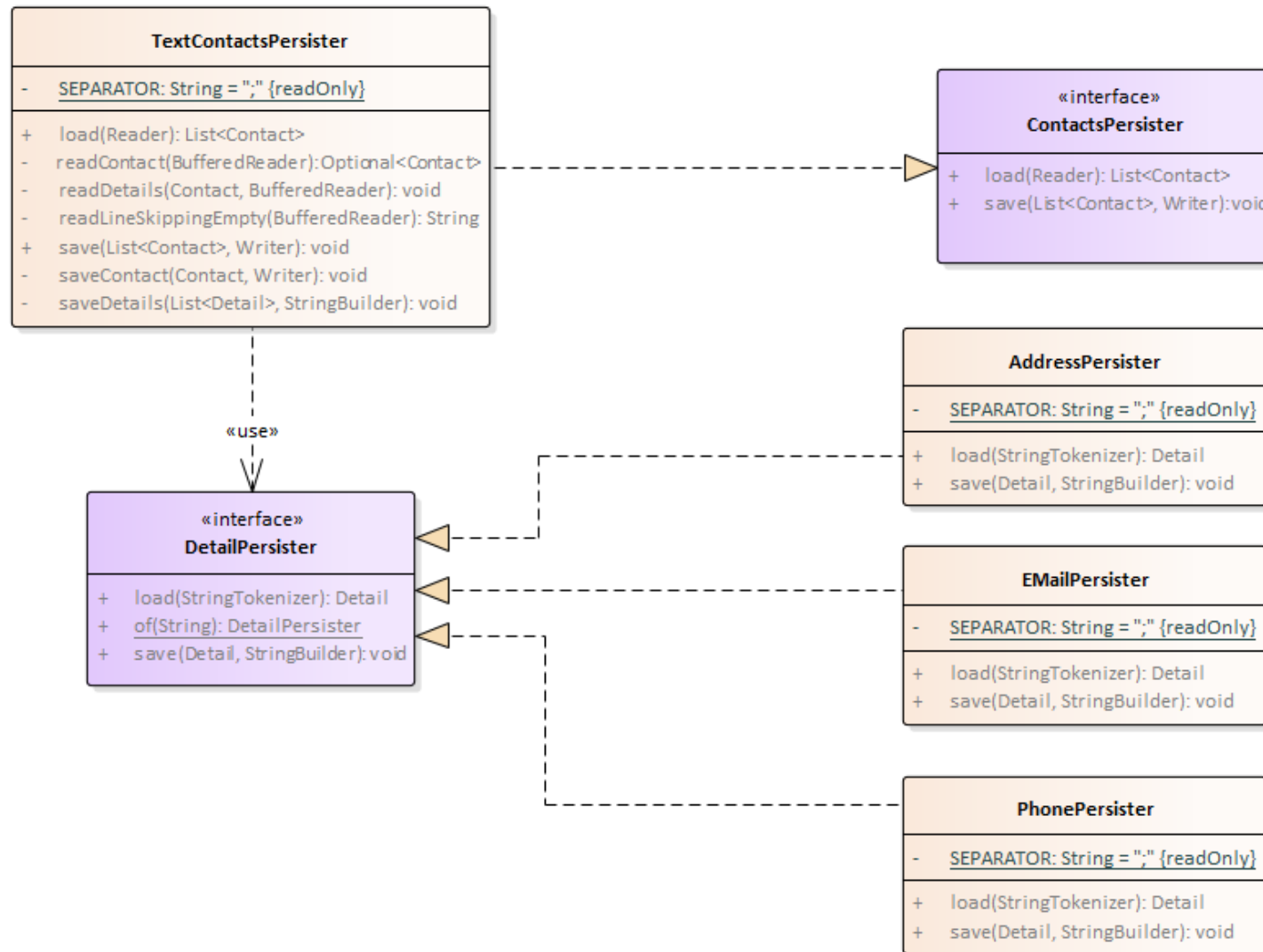
Persistence : Architettura Completa

TextContactsPersister

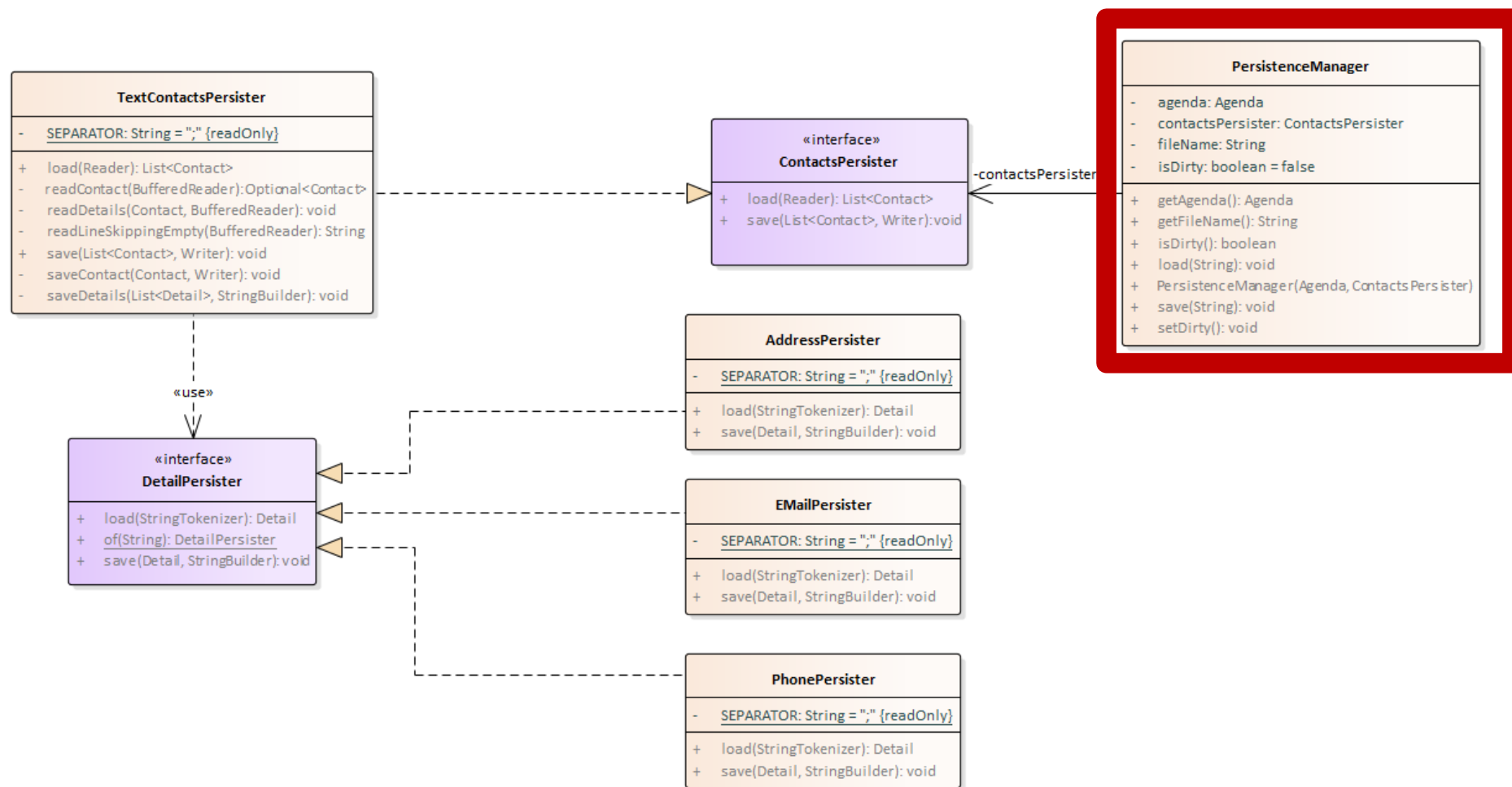
- **TextContactsPersister** implementa **ContactsPersister**
- Realizza l'algoritmo discusso nascondendone i dettagli
 - Da fuori non si vede tutta la complessità relativa ai **DetailPersister** con relativa *factory*!



Logica Persistenza: Modello (quasi) Completo



Logica Persistenza: Modello Completo



PersistenceManager

- È il «capo» della logica di persistenza
- È l'«**aiutante**» del **Controller**
- Il **Controller** lo userà quando sarà necessario caricare/salvare i contatti da/sul file
- Lo trovate pronto nello Start kit 😊

PersistenceManager	
-	agenda: Agenda
-	contactsPersister: ContactsPersister
-	fileName: String
-	isDirty: boolean = false
+	getAgenda(): Agenda
+	getFileName(): String
+	isDirty(): boolean
+	load(fileName: String): void
+	PersistenceManager(agenda: Agenda, contactsPersister: ContactsPersister)
+	save(fileName: String): void
+	setDirty(): void



Persistenza

- **Realizzazione delle classi di persistenza**
 - **AddressPersister**
 - **EMailPersister**
 - **PhonePersister**
 - **TextContactsPersister**
- **Nello Start Kit**
 - **ContactsPersister** e **DetailPersister**
 - **PersistenceManager**
 - test pronti nello Start Kit
 - classe **FileUtils** definisce il concetto di «separatore di linea»
- **Quanto ci devo mettere?**
 - il restante... (e forse oltre)

Package agenda.persistence

Il Sistema Completo

Agenda Contatti

▼ Lab12-Agenda

▼ src

> agenda

← Contiene AgendaApplication per il run dell'applicazione

> agenda.controller

← Il controller

> agenda.gui.view

> agenda.gui.viewfx.impl

} ← La view

> agenda.model

> agenda.persistence

> resources

← Immagini usate nella View

▼ test

> agenda.model.tests

> agenda.persistence.tests

> JRE System Library [jdk-17.0.2]

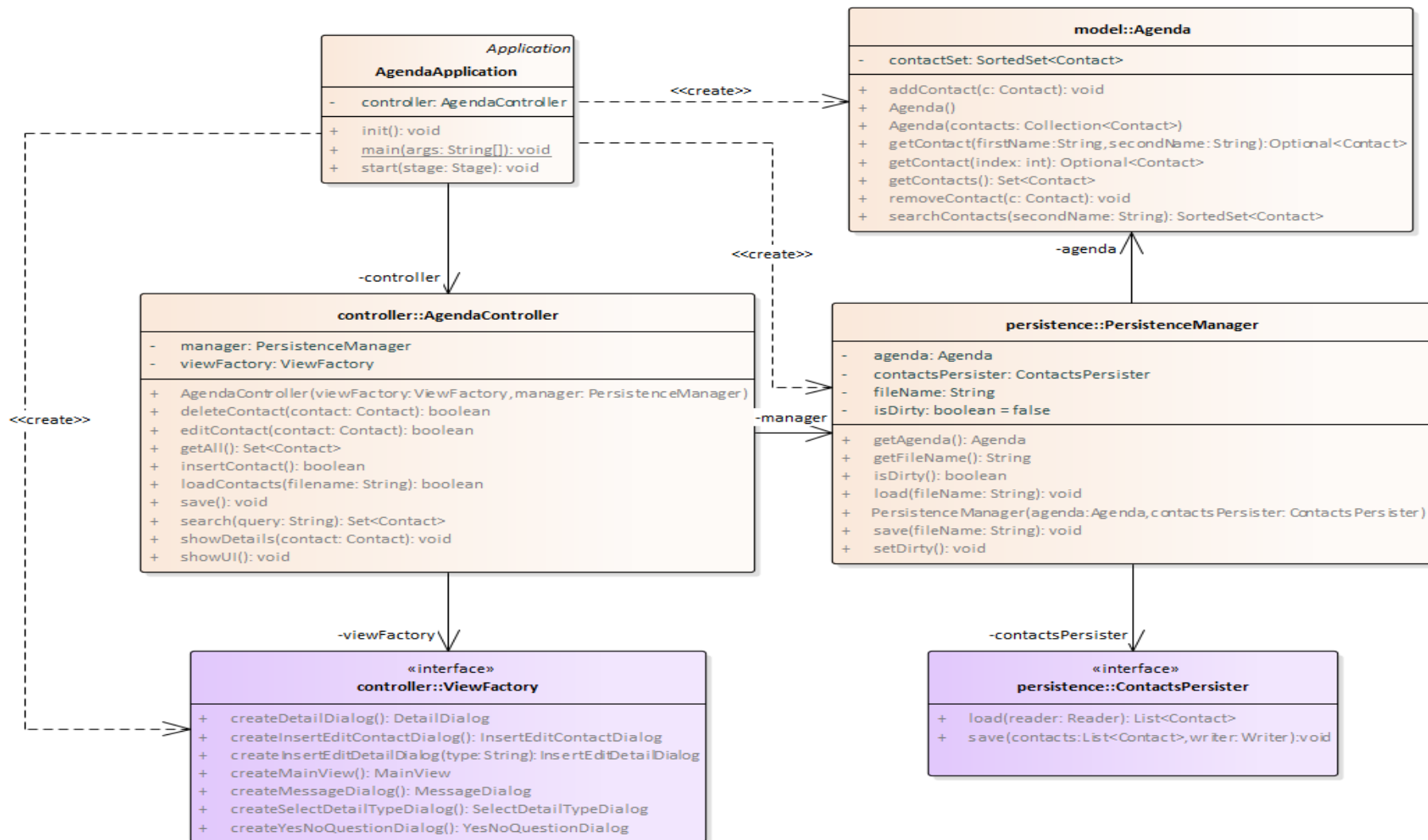
> JUnit 5

> JavaFx17

agenda.txt

AgendaBackup.txt

Il Sistema



Valutazione

Obiettivi?

- Quanto “**costa**” aggiungere un nuovo tipo di dettaglio di contatto?
 - Creare la nuova classe che estenda la classe astratta **Detail**
 - Creare la nuova classe che consenta il caricamento/salvataggio del nuovo dettaglio: deve implementare **DetailPersister**
 - Aggiungere una *entry* nelle rispettive factory
- Costa poco, esattamente come richiesto



well done

Buon Lavoro!

THINK TWICE
CODE ONCE!