

# Ingegneria Del Software

E. Tramontana

## Puntatori :-)

- Materiale, link utili, avvisi
  - <http://www.dmi.unict.it/~tramonta/se>
- Forum
  - <http://forum.informatica.unict.it>

2

Prof. Tramontana - Marzo 2020

## Lezioni

- Coprono tutto il programma del corso
- Partecipazione obbligatoria: si impara di più, e si ascolta da un esperto, è possibile fare domande ed ottenere risposte
- Orario di ricevimento
  - Mercoledì dalle 15:30 alle 17:30 (guardare avvisi sul forum)
- Per rendere efficace lo studio: esercitarsi con il codice, usare i concetti spiegati e i tool consigliati, partecipare alle lezioni
- Modalità Esami
  - Test a risposte multiple, test a risposta aperta (implementare codice, disegnare alcuni diagrammi UML), orale

3

Prof. Tramontana - Marzo 2020

## Libri Consigliati

Le slide non bastano :-(

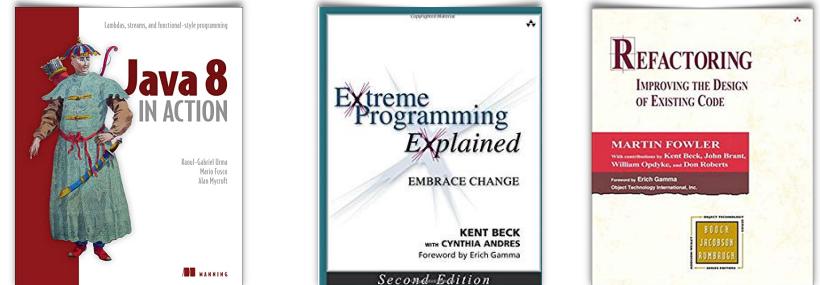
- Sommerville. Ingegneria del Software. Pearson
- oppure
- Pressman. Principi di Ingegneria del Software. McGraw-Hill
- Fowler. UML Distilled. Pearson
- Gamma, Helm, Johnson, Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley



4

## Libri Consigliati

- Urma, Fusco, Mycroft. Java 8 in Action. Manning
- Beck. Extreme Programming Explained. Addison-Wesley
- Fowler. Refactoring: Improving the design of existing code. Addison-Wesley

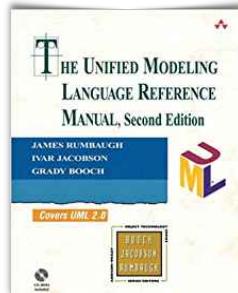
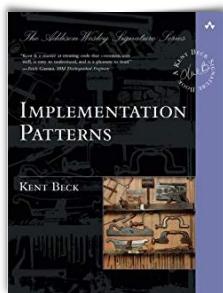


5

Prof. Tramontana - Marzo 2020

# Libri Per Approfondimenti

- Beck. Implementation Patterns. Addison-Wesley
- Rumbaugh, Jacobson, Booch. The Unified Modeling Language Reference Manual. Addison-Wesley



6

Prof. Tramontana - Marzo 2020

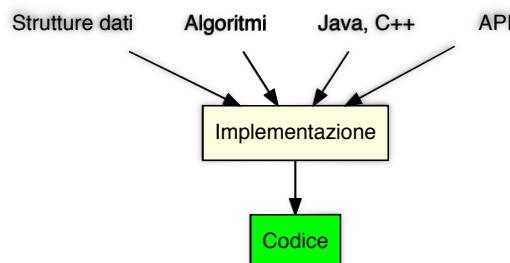
# Obiettivi Del Corso

- Descrivere come si sviluppa un sistema software di **grandi dimensioni**, che deve andare in produzione
- Fasi dei processi di sviluppo del software: analisi (requisiti), progettazione (OOP, **Design Pattern**, Refactoring), implementazione, test (convalida), manutenzione
- Processi di sviluppo: cascata, agili (XP), etc.
- Ci si baserà sulla progettazione orientata agli oggetti (OOP)
- Si useranno: lo standard UML, il **linguaggio Java**
- Java è attualmente molto diffuso e richiesto. Primo su Tiobe index, secondo su Pypl index (dopo Python), terzo su GitHub (dopo JavaScript e Python)

7

Prof. Tramontana - Marzo 2020

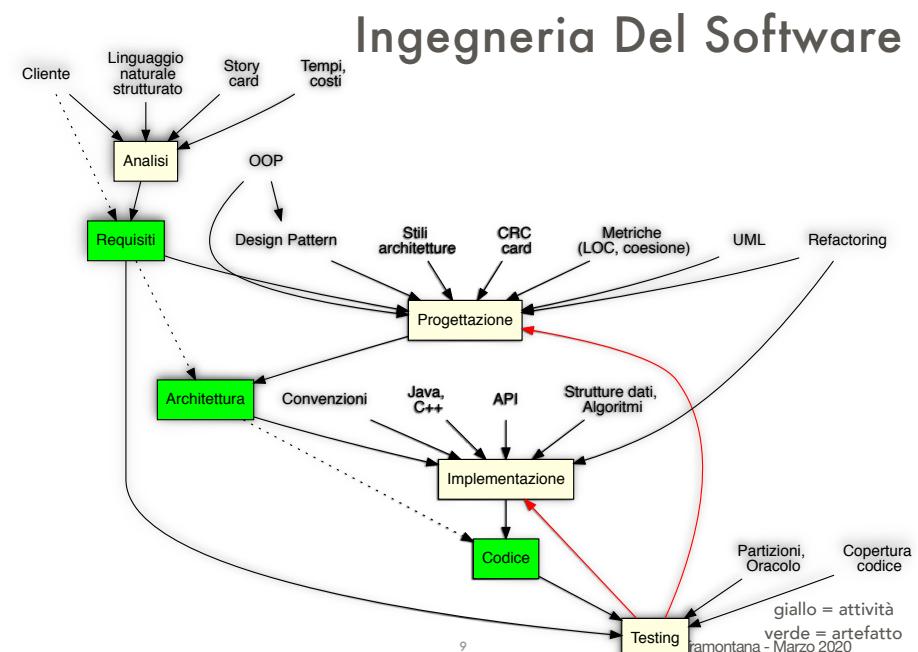
## Sviluppo ...



8

Prof. Tramontana - Marzo 2020

giallo = attività  
verde = artefatto



9

Prof. Tramontana - Marzo 2020

giallo = attività  
verde = artefatto

# Caratteristiche Del Software

- **Modificabilità:** un sistema software è intrinsecamente modificabile, poiché non ha parti fisiche (non è costituito da atomi)
- Se un sistema software è di successo vi è necessità di cambiarlo
  - Per adattarlo ad una realtà che cambia (mutate esigenze)
- Le richieste di estensione aumentano al crescere del successo
- Poiché di successo, il sistema software sopravvive all'hardware per cui era stato sviluppato inizialmente, generando una nuova esigenza di adattamento alla nuova piattaforma

10

Prof. Tramontana - Marzo 2020

# Qualità Del Software

- Le tecniche dell'ingegneria cercano di produrre sistemi software entro i costi e i tempi preventivi e con qualità accettabile
- Criteri operativi per valutare la qualità
  - **Correttezza:** il sistema software aderisce allo scopo ed è **conforme alle specifiche**
    - Il sistema software fa quello che il cliente vuole?
    - Il sistema software soddisfa le specifiche che erano state raccolte? [vedi Testing]
  - Efficienza, manutenibilità, dependability (sicurezza e affidabilità), usabilità

11

Prof. Tramontana - Marzo 2020

## Hello World

```
import java.time.LocalDate;

/**
 * Classe che stampa sullo schermo un messaggio e la data corrente
 */
public class HelloWorld { // definizione classe
    // dichiarazione e assegnazione campi
    private static final String msg = "Lezione di Ingegneria del Software";
    private static final LocalDate d = LocalDate.now();

    /**
     * Metodo da cui inizia l'esecuzione del programma
     *
     * @param args parametri passati al metodo all'avvio della classe
     */
    public static void main(String[] args) {
        System.out.println("Hello World");
        System.out.println(msg);
        System.out.println(d);
    }
}
```

Output  
Hello World  
Lezione di Ingegneria del Software  
2020-03-03

12

Prof. Tramontana - Marzo 2020

```
import java.time.LocalDate; // indica dove trovare la classe LocalDate

public class HelloWorld { // dichiara classe HelloWorld
    private static final String msg = "Lezione di Ingegneria del Software";
    private LocalDate d; // dichiara campo d di tipo LocalDate

    public static void main(String[] args) {
        System.out.println("Hello World"); // scrive su schermo
        System.out.println(msg);
        final HelloWorld world = new HelloWorld(); // crea oggetto
        world.printDate(); // chiama metodo
    }

    private void printDate() { // metodo
        d = LocalDate.now(); // chiama metodo static now
        System.out.println(d);
    }
}
```

- Il codice della classe HelloWorld deve essere salvato sul file HelloWorld.java, compilato con javac HelloWorld.java ed eseguito con java HelloWorld

Output  
Hello World  
Lezione di Ingegneria del Software  
2020-03-03

13

Prof. Tramontana - Marzo 2020

HelloWorld
- msg: String
- d: LocalDate
+ main(args: String[*])
- printDate()

# Parole Chiave Di Java

- **class** permette di definire un tipo, e quindi le sue istanze
- **final** definisce un campo o una variabile che non può essere assegnata più di una volta (una costante). Una classe final non può essere ereditata, un metodo final non può essere ridefinito (override)
- **import** indica dove trovare la definizione di una classe che sarà usata nel seguito
- **new** permette di creare un'istanza di una classe
- **private** e **public** indicano l'accessibilità di classi, campi e metodi
- **static** è usata per dichiarare un campo o un metodo appartenente alla classe (e non all'istanza)
- **void** indica che il metodo non ritorna alcun valore
- Tipi usati: **String**, per rappresentare insiemi di caratteri; **LocalDate**, per accedere alla data attuale; **System** per scrivere sullo schermo

14

Prof. Tramontana - Marzo 2020

# Un Esempio Pratico

- Riusciamo a sviluppare un componente software che risulti: riusabile, modificabile e corretto?
- Consideriamo un componente estremamente piccolo (potrebbe far parte di un sistema più grande)
- Descrizione dei requisiti
  - Dati vari file contenenti valori numerici, con un valore per ciascuna riga del file
    1. Leggere da ciascun file la lista di valori
    2. Tenere solo i valori non duplicati
    3. Calcolare la somma dei numeri letti dal file (non duplicati)
    4. Calcolare il massimo fra i numeri letti

15

Prof. Tramontana - Marzo 2020

```
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import java.io.LineNumberReader;
import java.util.ArrayList;
import java.util.List;

public class CalcolaImporti { // classe Java vers 0.0.1
    private final List<String> importi = new ArrayList<>();
    // List e ArrayList sono tipi della libreria Java
    private float totale;

    public float calcola(String c, String n) throws IOException { // metodo
        LineNumberReader f = new LineNumberReader(new FileReader(new File(c, n)));
        // lettura file tramite le API Java: File, FileReader, LineNumberReader

        totale = 0;
        while (true) {
            final String riga = f.readLine(); // legge una riga dal file
            if (null == riga) break; // esce dal ciclo
            importi.add(riga); // aggiunge in lista
            totale += Float.parseFloat(riga); // converte da String a float
        }
        f.close(); // chiude file
        return totale; // restituisce totale al chiamante
    }
}
```

16

Prof. Tramontana - Marzo 2020

# Linguaggio Java

- Parole chiave
  - **float** si usa per dichiarare una variabile che può tenere numeri in virgola mobile; si usa pure per dichiarare che un metodo restituisce un valore float
  - **if** si usa per creare un'istruzione condizionale
  - **return** si usa per concludere l'esecuzione di un metodo, se seguita da un valore quest'ultimo è restituito al chiamante
  - **throws** si usa nelle dichiarazioni di metodi per indicare quali eccezioni non sono gestite dal metodo ma passate
  - **while** si usa per creare un ciclo

17

Prof. Tramontana - Marzo 2020

# Progressi

- Passi implementati
  - lettura da file
  - calcolo del totale
- Da fare
  - controlli su valori unici
  - estrazione del massimo

```
public class CalcolaImporti { // classe Java vers 0.0.2
    private final List<String> importi = new ArrayList<>();
    private float totale, massimo;

    public float calcola(String c, String n) throws IOException {
        LineNumberReader f = new LineNumberReader(new FileReader(new File(c, n)));
        // lettura file tramite le API Java: File, FileReader, LineNumberReader

        totale = massimo = 0;
        while (true) {
            final String riga = f.readLine(); // legge una riga dal file
            if (null == riga) break; // esce dal ciclo
            importi.add(riga); // aggiunge in lista
            float x = Float.parseFloat(riga); // converte da String a float
            totale += x; // aggiorna totale
            if (massimo < x) massimo = x; // aggiorna massimo
        }
        f.close(); // chiude file
        return totale; // restituisce il totale al chiamante
    }
}
```

18

Prof. Tramontana - Marzo 2020

19

Prof. Tramontana - Marzo 2020

# Progressi

- Passi implementati
  - lettura da file
  - calcolo del totale
  - estrazione del massimo
- Da fare
  - controlli su valori unici

```
public class CalcolaImporti { // classe Java vers 0.1
    private final List<String> importi = new ArrayList<>();
    private float totale, massimo;

    public float calcola(String c, String n) throws IOException {
        LineNumberReader f = new LineNumberReader(new FileReader(new File(c, n)));
        // lettura file tramite le API Java: File, FileReader, LineNumberReader

        totale = massimo = 0;
        while (true) {
            final String riga = f.readLine(); // legge una riga dal file
            if (null == riga) break; // esce dal ciclo
            if (!importi.contains(riga)) { // se non presente
                importi.add(riga); // aggiunge in lista
                float x = Float.parseFloat(riga); // converte da String a float
                totale += x; // aggiorna totale
                if (massimo < x) massimo = x; // aggiorna massimo
            }
        }
        f.close(); // chiude file
        return totale; // restituisce il totale al chiamante
    }
}
```

20

Prof. Tramontana - Marzo 2020

21

Prof. Tramontana - Marzo 2020

# Librerie Java

- Riepilogo di alcuni tipi e metodi di librerie Java utilizzati
- List, interfaccia utile a tenere una sequenza di elementi
- ArrayList, implementazione di List, la sua dimensione cresce automaticamente
- add(), metodo di List, aggiunge un elemento alla fine della lista
- contains(), metodo di List, ritorna true se la lista contiene l'elemento specificato
- parseFloat(String s), metodo di Float, ritorna un nuovo float con il valore specificato nel parametro stringa s, o ritorna un'eccezione se la stringa non contiene un numero
- readLine(), metodo di LineNumberReader, ritorna una stringa contenente la linea del file, o null se si raggiunge la fine del file

22

Prof. Tramontana - Marzo 2020

# Progressi

- Passi implementati
  - lettura da file
  - calcolo del totale
  - estrazione del massimo
  - controlli su valori unici
- Il codice è conforme alla programmazione OO?
- E se il codice prodotto fosse invece ...

23

Prof. Tramontana - Marzo 2020

```
public class CalcolaImporti { // classe Java vers 0.2

    private final List<String> importi = new ArrayList<>();
    private float totale, massimo;

    public float calcola(String c, String n) throws IOException {
        LineNumberReader f = new LineNumberReader(new FileReader(new File(c, n)));
        String riga;
        while (true) {
            riga = f.readLine();
            if (null == riga) break;
            if (!importi.contains(riga))
                importi.add(riga);
        }
        f.close();
        // calcola totale
        totale = 0;
        for (int i = 0; i < importi.size(); i++)
            totale += Float.parseFloat(importi.get(i));
        // calcola massimo
        massimo = Float.parseFloat(importi.get(0));
        for (int i = 1; i < importi.size(); i++)
            if (massimo < Float.parseFloat(importi.get(i)))
                massimo = Float.parseFloat(importi.get(i));
        return totale;
    }
}
```

24

Prof. Tramontana - Marzo 2020

# Problemi?

- Il metodo calcola di entrambe le versioni è spaghetti code (un antipattern)
- Il codice è monolitico: fa troppe cose in un unico flusso. Non è un codice Object-Oriented. Conseguenze: non si può riusare, né verificarne la correttezza
  1. Come verificare che tutti i valori del file siano stati letti? Si dovrà modificare il metodo. Non è una soluzione, si dovrebbe poter verificare il comportamento del metodo dall'esterno
  2. Analogamente per verificare il calcolo di somma e totale, in più punti si dovrebbero aggiungere alcuni controlli
  3. Non si riesce a modificare facilmente o riusare il codice. Per es. se si volessero conservare tutti i valori letti, quali ulteriori effetti provoca la modifica?
- Quindi: difficoltà di comprensione e modifiche che coinvolgono varie operazioni

25

Prof. Tramontana - Marzo 2020

# Spaghetti Code

- Metodi lunghi, senza parametri, e che usano variabili globali
- Flusso di esecuzione determinato dall'implementazione interna all'oggetto, non dai chiamanti
- Interazioni minime fra oggetti
- Nomi classi e metodi indicano la programmazione procedurale
- Ereditarietà e polimorfismo non usati, riuso impossibile
- Gli oggetti non mantengono lo stato fra le invocazioni
- Cause: inesperienza con OOP, nessuna progettazione

26

Prof. Tramontana - Marzo 2020

```

public class CalcolaImporti { // classe Java vers 0.2

    private final List<String> importi = new ArrayList<>();
    private float totale, massimo;

    public float calcola(String c, String n) throws IOException {
        LineNumberReader f = new LineNumberReader(new FileReader(new File(c, n)));
        String riga;
        while (true) {
            riga = f.readLine();
            if (null == riga) break;
            if (!importi.contains(riga))
                importi.add(riga);
        }
        f.close();
        totale = 0;
        for (int i = 0; i < importi.size(); i++) {
            totale += Float.parseFloat(importi.get(i));
        }
        massimo = Float.parseFloat(importi.get(0));
        for (int i = 1; i < importi.size(); i++)
            if (massimo < Float.parseFloat(importi.get(i)))
                massimo = Float.parseFloat(importi.get(i));
        return totale;
    }

    public void leggiFile() {
        try {
            LineNumberReader f = new LineNumberReader(new FileReader(new File(c, n)));
            String riga;
            while (true) {
                riga = f.readLine();
                if (null == riga) break;
                inserisci(riga);
            }
            f.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public void inserisci(String riga) {
        if (!importi.contains(riga))
            importi.add(riga);
    }

    public void calcolaSomma() {
        totale = 0;
        for (String v : importi)
            totale += Float.parseFloat(v);
    }

    public void calcolaMassimo() {
        massimo = 0;
        for (String v : importi)
            if (massimo < Float.parseFloat(v))
                massimo = Float.parseFloat(v);
    }

    public float getSomma() {
        return totale;
    }

    public float getMassimo() {
        return massimo;
    }
}

```

27

Prof. Tramontana - Marzo 2020

```

public class CalcolaImporti { // classe Java vers 0.1

    private final List<String> importi = new ArrayList<>();
    private float totale, massimo;

    public float calcola(String c, String n) throws IOException {
        LineNumberReader f = new LineNumberReader(new FileReader(new File(c, n)));
        totale = 0;
        massimo = 0;
        while (true) {
            String riga = f.readLine();
            leggiFile()
            if (null == riga) break;
            if (!importi.contains(riga))
                inserisci()
            importi.add(riga);
            float x = Float.parseFloat(riga);
            calcolaSomma()
            totale += x;
            if (massimo < x) massimo = x;
            calcolaMassimo()
        }
        f.close();
        return totale;
    }

    public void leggiFile() {
        try {
            LineNumberReader f = new LineNumberReader(new FileReader(new File(c, n)));
            String riga;
            while (true) {
                riga = f.readLine();
                if (null == riga) break;
                inserisci(riga);
            }
            f.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public void inserisci(String riga) {
        if (!importi.contains(riga))
            importi.add(riga);
    }

    public void calcolaSomma() {
        totale = 0;
        for (String v : importi)
            totale += Float.parseFloat(v);
    }

    public void calcolaMassimo() {
        massimo = 0;
        for (String v : importi)
            if (massimo < Float.parseFloat(v))
                massimo = Float.parseFloat(v);
    }

    public float getSomma() {
        return totale;
    }

    public float getMassimo() {
        return massimo;
    }
}

```

28

Prof. Tramontana - Marzo 2020

<b>Pagamenti</b>
- importi: List<String>
- totale: float
- massimo: float
+ leggiFile(c: String, n: String)
+ inserisci(riga: String)
+ calcolaSomma()
+ calcolaMassimo()
+ svuota()
+ getMassimo(): float
+ getSomma(): float

```

public class Pagamenti { // Pagamenti vers 1.1

    private List<String> importi = new ArrayList<>();
    private float totale, massimo;

    public void leggiFile(String c, String n) throws IOException {
        LineNumberReader f = new LineNumberReader(new FileReader(new File(c, n)));
        String riga;
        while (true) {
            riga = f.readLine();
            if (null == riga) break;
            inserisci(riga);
        }
        f.close();
    }

    public void inserisci(String riga) {
        if (!importi.contains(riga))
            importi.add(riga);
    }

    public void calcolaSomma() {
        totale = 0;
        for (String v : importi) // enhanced for
            totale += Float.parseFloat(v);
    }

    public void calcolaMassimo() {
        massimo = 0;
        for (String v : importi)
            if (massimo < Float.parseFloat(v))
                massimo = Float.parseFloat(v);
    }

    public void svuota() {
        importi = new ArrayList<>();
        totale = massimo = 0;
    }

    public float getMassimo() {
        return massimo;
    }

    public float getSomma() {
        return totale;
    }
}

// chiamate da un'altra classe
public static void main(String[] args) {
    Pagamenti p = new Pagamenti();
    try {
        p.leggiFile("csv", "importi");
    } catch (IOException e) {
        e.printStackTrace();
    }
    p.calcolaSomma();
    p.calcolaMassimo();
}

```

29

Prof. Tramontana - Marzo 2020

# Considerazioni Sul Codice

- Si sta usando bene il paradigma di programmazione ad Oggetti (OOP)
  - Ogni metodo ha una sola piccola responsabilità
  - Il flusso di chiamate ai metodi è indipendente dai singoli algoritmi
  - Posso riusare (richiamandoli) i servizi offerti dai metodi
- Inoltre, sto usando il paradigma **Command e Query**
  - I metodi **Query** restituiscono un risultato (si vede dal parametro di ritorno), e non modificano lo stato del sistema
  - I metodi **Command** (o modificatori) cambiano lo stato del sistema ma non restituiscono un valore
  - I metodi **query** si possono chiamare liberamente, senza preoccupazioni sulla modifica dello stato, mentre si deve stare più attenti quando si chiamano i metodi **command**
- *Enhanced for* indica che si vogliono gli elementi della lista, uno per ogni passata, si può usare con i tipi che implementano **Iterable**

30 Prof. Tramontana - Marzo 2020

# Metriche

- Classe **CalcolalImporti** (vers. 0.1)
  - Metodi 1, LOC 26 (di cui 5 linee sono per i vari import)
- Classe **CalcolalImporti** (vers. 0.2)
  - Metodi 1, LOC 29
- Classe **Pagamenti** (vers 1.1)
  - Metodi 7, LOC 43 (media 6 LOC per metodo)
- Confronto con sistemi software open source (valori approssimativi) JUnit (JU), JHotDraw (JHD):
  - JU LOC 22K, Classi 231, Metodi 1200, Attributi 265, media 18
  - JHD LOC 28K, Classi 600, Metodi 4814, Attributi 1151, media 6

31

Prof. Tramontana - Marzo 2020

# Conclusioni

## Key points

- Correttezza del codice: test
- Antipattern Spaghetti Code
- Ciascun metodo ha un unico compito
- Esempi di domande d'esame
  - Implementare un frammento di codice che usa l'enhanced for
  - Dire come si può controllare se un codice è corretto
  - Implementare un metodo query
  - Dire qual è la differenza fra List ed ArrayList
  - Dire a cosa serve il metodo contains di List

32

Prof. Tramontana - Marzo 2020

# Processi (di sviluppo del) software

- Un processo software descrive le attività (o task) necessarie allo sviluppo di un prodotto software e come queste attività sono collegate tra loro
- Attività o fasi dello sviluppo
  - Analisi dei requisiti (specifiche)
  - Progettazione (design)
  - Codifica o implementazione (codice)
  - Convalida o testing (approvazione)
  - Manutenzione

E. Tramontana - Processi Software - 21-Mar-11 1

## Esempi di Feature e Requisiti

- Feature di Firefox 3.6
  - Browsing privatamente: navigazione del web senza lasciare tracce
  - Password manager: ricordare le password dei siti, senza usare pop-up
  - Awesome Bar: trovare i siti preferiti in pochi secondi
  - One-click bookmark: bookmark, cerca e organizza siti web velocemente e facilmente
- Requisiti (sintetici) di Firefox 3.7
  - Eseguire i plug-in in un processo separato per migliorare la stabilità dell'applicazione e diminuire i tempi di risposta
  - Migliorare i tempi di startup
  - Ottimizzare caricamento delle pagine

E. Tramontana - Processi Software - 21-Mar-11 3

## Fase di Analisi dei Requisiti

- L'analisi dei requisiti è il processo che porta a definire le specifiche, stabilisce i servizi richiesti ed i vincoli del software
  - Requisito: ciascuna delle caratteristiche che il software deve avere
  - Specifiche: descrizione rigorosa delle caratteristiche del software
    - I requisiti tendono ad essere granulari (ovvero: molti e piccoli)
  - Feature: un set di requisiti correlati tra loro
    - Una feature permette all'utente di soddisfare un obiettivo
- Fasi per l'ingegneria dei requisiti [vedi lezioni da 5 a 7]
  - (1) Studio di fattibilità, (2) Analisi dei requisiti, (3) Specifica dei requisiti, (4) Convalida requisiti
- Requisiti
  - Funzionali: Cosa il sistema deve fare (funzionalità)
  - Non-funzionali: Come il sistema lo fa (es. affidabilità, efficienza, prestazioni, manutenibilità, etc.)

E. Tramontana - Processi Software - 21-Mar-11 2

## Progettazione ed implementazione

- Fase di Progettazione [vedi lezioni da 8 a 18]
  - Il processo che stabilisce la struttura software che realizza le specifiche
  - Attività della progettazione
    1. Suddivisione dei requisiti
    2. Identificazione sottosistemi, ovvero progettazione architettura software
    3. Specifica delle responsabilità dei sottosistemi
    4. Progettazione di: interfacce, componenti, strutture dati, algoritmi
  - Ognuna delle attività suddette produce un documento corrispondente (o integra un documento già esistente) che descrive un modello
    - Modello degli oggetti, di sequenza, di transizione stati, strutturale, data-flow
- Fase di Implementazione
  - Produce un programma eseguibile a partire dalla struttura stabilita
- Progettazione ed implementazione sono attività correlate e spesso sono alternate

E. Tramontana - Processi Software - 21-Mar-11 4

# Fase di Implementazione

- Consiste nella programmazione ovvero nella traduzione dei modelli del progetto in un programma (codice) e della rimozione degli errori dal programma
- I programmatore effettuano alcuni test sul programma prodotto per scoprire bug e rimuoverli
- Per rimuovere i bug
  1. Localizzare l'errore nel codice
  2. Rimuovere l'errore nel modello e poi nel codice
  3. Effettuare nuovamente il test nel programma



E. Tramontana - Processi Software - 21-Mar-11 5

# Fase di Test

- Test di componenti o unità (unit test)
  - I singoli componenti sono testati indipendentemente
  - Componenti potranno essere funzioni, o oggetti, o loro raggruppamenti
- Test di sistema
  - L'intero sistema è testato, dando speciale importanza alle proprietà emergenti
- Test di accettazione (alpha testing)
  - Test condotti dagli sviluppatori con dati del cliente per verificare che il sistema soddisfi le esigenze del cliente
- Beta test: test condotti da alcuni clienti sul prodotto quasi completo
- Prodotti software corrispondenti alle varie fasi di test
  - versione alfa, versione beta, versione golden

E. Tramontana - Processi Software - 21-Mar-11 7

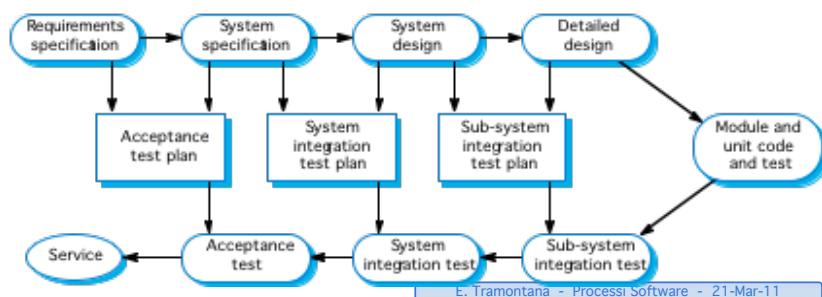
# Fase di Convalida (Verifica & Validazione)

- La fase di convalida o Verifica e Validazione (V & V) del sistema software intende mostrare che il sistema software è conforme alle specifiche e che soddisfa le richieste (aspettative) del cliente
  - Viene condotta tramite processi di revisione e test del sistema software
  - I test mirano ad eseguire il sistema software in condizioni derivate dalle specifiche di dati reali che il sistema software dovrà elaborare
  - [vedi lezione 21]

E. Tramontana - Processi Software - 21-Mar-11 6

# Quadro riassuntivo

- Dai requisiti (R) otteniamo il documento della specifica dei requisiti (SRS)
- Dall'SRS ricaviamo il design del sistema (DS)
- Dal DS ricaviamo il design dettagliato (DD)
- Da DD ricaviamo codice e test
- Da DS e da DD ricaviamo come integrare i sottosistemi e come fare i test di sistema
- Da R e SRS ricaviamo come fare i test di accettazione

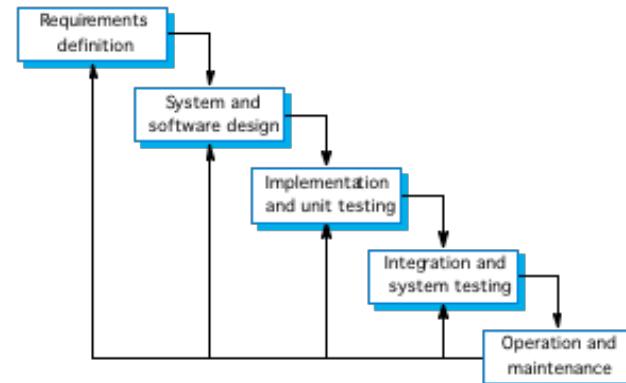


E. Tramontana - Processi Software - 21-Mar-11 8

# Evoluzione

- Il software è intrinsecamente flessibile e può cambiare
- Al cambiare dei requisiti per cambiamenti dell'ambiente a cui è rivolto (business, hardware, etc.), il software deve evolvere se deve rimanere ad essere utile
- [vedi lezione 19]

# Cascata (Waterfall) [Royce 1970]



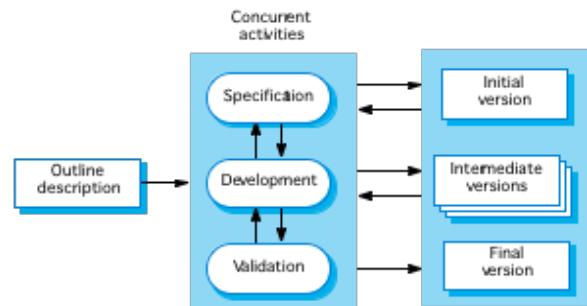
# Cascata (Waterfall)

- Il primo dei processi (anni '70), derivato da altri processi di ingegneria
- Focalizza sul prodotto completo
- Si comincia la fase successiva solo se la fase precedente è completa
  - Prima specifica tutto, poi produci tutto, poi testa tutto, ...
- Processo statico con tanta documentazione
  - Lungo tempo per ottenere il prodotto
  - Poche interazioni con i clienti (solo nella fase iniziale)
  - Difficoltà ad introdurre i cambiamenti richiesti dal cliente
  - + Consistenza tra artefatti
  - + Ampia documentazione
  - + Utile se i requisiti sono stabili e chiaramente definiti
  - + Usato principalmente per sistemi grandi, complessi, critici, per gestire team numerosi
  - + Alta qualità del codice prodotto

# Processo Evolutivo

- Il processo evolutivo ha due varianti: esplorazione e Build and Fix
- Sviluppo per esplorazione
  - Gli sviluppatori lavorano con i clienti
  - Dalle specifiche iniziali si arriva per mezzo di trasformazioni successive (evoluzione) fino al sistema software finale
  - Dovrebbe partire da requisiti ben chiari ed aggiungere nuove caratteristiche definite dal cliente
- Sviluppo Build and Fix
  - Documentazione inesistente o quasi
  - Comprensione limitata del sistema da produrre
    - Costruire la prima versione e modificarla fino a che il cliente è soddisfatto
    - Fase di design pressoché inesistente
    - Codice prodotto di bassa qualità

# Evolutivo



E. Tramontana - Processi Software - 21-Mar-11 13

# Evolutivo

- Problemi
  - Tempi lunghi
  - Sistemi difficilmente comprensibili e modificabili, probabilmente non corretti
  - Mancanza di visione d'insieme del progetto
- Applicabilità
  - Sistemi di piccole dimensioni
  - Singole parti di sistemi grandi (es. interfaccia utente)
  - Sistemi con vita breve (es. prototipi)

E. Tramontana - Processi Software - 21-Mar-11 14

# Altri Processi

- Processo di Sviluppo Incrementale
  - Sono implementate prima le funzionalità di base (o prioritarie)
  - Al codice sviluppato in precedenza è aggiunto altro codice per un altro gruppo di funzionalità
  - Si ripete il passo precedente, fino a completamento
- Processo CBSE o basato su COTS
  - COTS = componenti esistenti (Components Off The Shelf)
  - Analisi dei componenti esistenti
  - Modifica dei requisiti (?)
  - Progettazione tramite riuso
  - Sviluppo ed integrazione

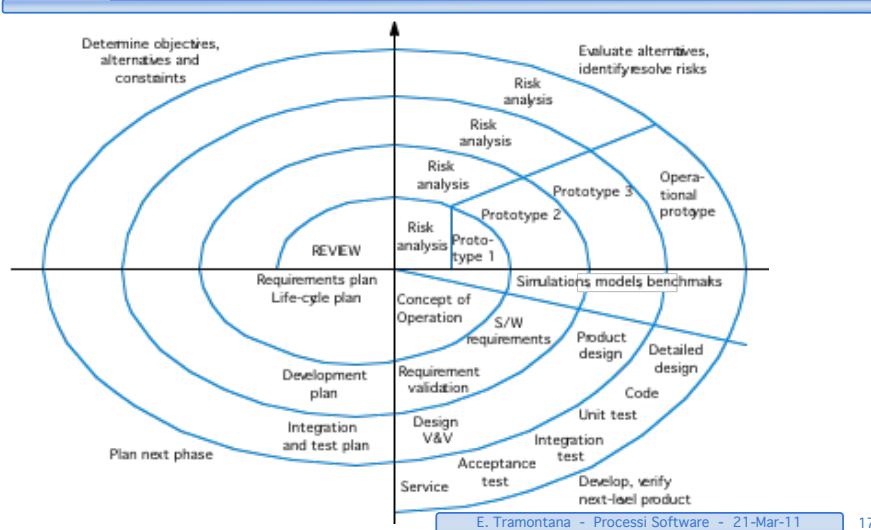
E. Tramontana - Processi Software - 21-Mar-11 15

# A Spirale [Boehm 1988]

- Focalizza su tanti prodotti parziali (sottosistemi funzionali)
- Ogni loop (giro della spirale) è una fase (es. loop per requisiti)
- Ogni loop consiste dei seguenti settori
  1. Identificazione obiettivi specifici per la fase corrente
  2. Valutazione rischi del progetto
    - Terminologia: Rischio = qualcosa che può impedire il successo e che è sconosciuta
    - Successo: soddisfare tutti i requisiti. Attributi del rischio: (i) probabilità di occorrenza; (ii) impatto sul progetto (ovvero gravità, danno peggiore)
  3. Produzione di una parte e convalida della parte
  4. Revisione del progetto e pianificazione fase successiva
- Processo agile
  - + Poco tempo per la prima versione del prodotto
  - + Opportunità di interagire con il cliente
- Ogni fase produce un codice testato ed integrato nel sistema complessivo

E. Tramontana - Processi Software - 21-Mar-11 16

# A Spirale



# Settori del processo a Spirale

- Stabilire obiettivi
  - Gli obiettivi per la fase corrente sono identificati
- Valutare il rischio e ridurlo
  - I rischi sono valutati ed attività sono intraprese per ridurre quelli più importanti
- Sviluppo e convalida
  - Secondo uno dei modelli precedenti
- Pianificazione
  - Il progetto è revisionato e la prossima fase della spirale è pianificata

# Miglioramenti Sul Codice

- Separazione delle varie operazioni, ciascuna in un suo metodo.  
Ciò permette di verificare la correttezza delle operazioni e di riusarle
- Costruzione di astrazioni, via via più utili, ovvero metodi di livello più basso e metodi di livello più alto, che richiamano i precedenti, lo stesso si fa per le classi
- Separazione delle operazioni che aggiornano lo stato (command) da quelle che restituiscono valori (query), in metodi diversi.  
Query sono operazioni che ottengono informazioni (accedendo a dati o facendo calcoli sui dati). Command sono operazioni che effettuano cambiamenti
- Lo stato dell'oggetto diventa osservabile, attraverso metodi opportuni, in modo da poter fare i test

1

Prof. Tramontana - Marzo 2019

```
// continua
public void testLeggiFile() {
    try {
        pgm.leggiFile("csvfiles", "Importi.csv");
        System.out.println("OK test leggi file");
    } catch (IOException e) {
        System.out.println("FAILED test leggi file");
    }
}

public static void main(String[] args) {
    TestPagamenti tl = new TestPagamenti();
    tl.testLeggiFile();
    tl.testSommaValori();
    tl.testMaxValore();
}
}

Output dell'esecuzione di TestPagamenti quando il file Importi.csv è presente nella cartella csv (dentro la cartella con gli eseguibili)
OK test leggi file
OK test somma val
OK test massimo val

Esecuzione quando il file non viene trovato
FAILED test leggi file
OK test somma val
OK test massimo val
```

3

Prof. Tramontana - Marzo 2019

```
public class TestPagamenti { // per classe Pagamenti vers 0.9
    private Pagamenti pgm = new Pagamenti();

    private void initLista() {
        pgm.svuota();
        pgm.inserisci("321.01");
        pgm.inserisci("531.7");
        pgm.inserisci("1234.5");
    }

    public void testSommaValori() {
        initLista();
        pgm.calcolaSomma();
        if (pgm.getSomma() == 2087.21f)
            System.out.println("OK test somma val");
        else System.out.println("FAILED test somma val");
    }

    public void testMaxValore() {
        initLista();
        pgm.calcolaMassimo();
        if (Math.abs(pgm.getMassimo() - 1234.5f) < 0.01)
            System.out.println("OK test massimo val");
        else System.out.println("FAILED test massimo val");
    }

// continua ...
```

2

Prof. Tramontana - Marzo 2019

```
public class Pagamenti { // Pagamenti vers 1.2
    private List<Float> importi = new ArrayList<>();
    public void leggiFile(String c, String n) throws IOException {
        LineNumberReader f = new LineNumberReader(new FileReader(new File(c, n)));
        String riga;
        while (true) {
            riga = f.readLine();
            if (riga == null) break;
            inserisci(Float.parseFloat(riga));
        }
        f.close();
    }
    public void inserisci(float x) {
        if (!importi.contains(x)) importi.add(x);
    }
    public float calcolaSomma() {
        float risultato = 0;
        for (float v : importi) risultato += v;
        return risultato;
    }
    public float calcolaMassimo() {
        float risultato = 0;
        for (float v : importi)
            if (risultato < v) risultato = v;
        return risultato;
    }
    public void svuota() {
        importi = new ArrayList<>();
    }
}
```

## Pagamenti

- importi: List<Float>
+ leggiFile(c: String, n: String)
+ inserisci(x: float)
+ calcolaSomma(): float
+ calcolaMassimo(): float
+ svuota()

4

Prof. Tramontana - Marzo 2019

```

public class TestPagamenti { // per classe Pagamenti vers 1.2
    private Pagamenti pgm = new Pagamenti();
    private void initLista() {
        pgm.svuota();
        pgm.inserisci(321.01f);
        pgm.inserisci(531.7f);
        pgm.inserisci(1234.5f);
    }
    public void testSommaValori() {
        initLista();
        if (pgm.calcolaSomma() == 2087.21f) System.out.println("OK test somma val");
        else System.out.println("FAILED test somma val");
    }

    public void testListaVuota() {
        pgm.svuota();
        if (pgm.calcolaSomma() == 0) System.out.println("OK test somma val empty");
        else System.out.println("FAILED test somma val empty");
        if (pgm.calcolaMassimo() == 0) System.out.println("OK test massimo val empty");
        else System.out.println("FAILED test massimo val empty");
    }
}

```

5

Prof. Tramontana - Marzo 2019

## Ulteriori Risorse

- Per i tool di sviluppo e il runtime Java: [www.java.com](http://www.java.com)
- Per compilare una classe Java da shell dei comandi
  - javac NomeClasse.java, es. javac Pagamenti.java
- Se la compilazione va a buon fine non viene dato nessun messaggio e viene generato un file NomeClasse.class
- Per eseguire un programma Java
  - java NomeClasse, es. java Pagamenti
- La classe che si dà come argomento del comando java deve avere il metodo main (`public static void main(String[] args) { ... }`), che è il metodo da cui comincia l'esecuzione del codice
- La classe Java specifica con import dove trovare il tipo che fa parte della libreria Java ed è usato all'interno della classe (per es. List, File), es.  
`import java.io.File,` inserito all'inizio del file NomeClasse.java

7

Prof. Tramontana - Marzo 2019

## Punti Importanti Da Ricordare

- Correttezza: è stato possibile eseguire test che verificano il codice. Soddisfare (e verificare) i requisiti permette di ottenere la qualità del sistema
  - I test documentano le condizioni sotto le quali il codice funziona e protegge il codice da modifiche indesiderate. I test devono essere indipendenti fra loro ed auto valutarsi
- Test Driven Development (TDD): scrivere prima il test e poi il codice che risolve un requisito. Lo sviluppatore sceglie nomi di classi e metodi (progettazione)
- Responsabilità: i compiti sono suddivisi su vari metodi (e quindi su più classi), questo permette di ottenere coesione del codice
- Principio di Singola Responsabilità. La singola responsabilità è cruciale per la comprensione, il riuso, l'ereditarietà (OOP)
- Astrazioni: lo sviluppatore OOP costruisce astrazioni. Il nome delle astrazioni è estremamente importante: il nome che si dà a classi e metodi ne descrive l'obiettivo

6

Prof. Tramontana - Marzo 2019

## Conclusioni

- Key points
  - Test driven development
  - Single Responsibility Principle
- Esempi di domande d'esame
  - Implementare un test per un metodo che prende in ingresso un intero
  - Dire come si compila una classe in Java
  - Implementare una classe Java che può essere data alla JVM per essere eseguita

8

Prof. Tramontana - Marzo 2019

# Refactoring

- Per passare dalla versione 0.2 alla versione 0.9 è stata usata la tecnica di Refactoring Estrai Metodo
- La versione 1.2, eliminando due attributi, previene eventuali effetti collaterali (ovvero non voluti) di altre operazioni

# Refactoring

- Refactoring è il processo che cambia un sistema software in modo che il comportamento esterno del sistema non cambi, ovvero i requisiti funzionali soddisfatti sono gli stessi, per far sì che la struttura interna sia migliorata
- Si migliora la progettazione a poco a poco, durante e dopo l'implementazione del codice
- Esempio semplice: consolidare (ovvero eliminare) frammenti di codice duplicati all'interno di rami condizionali
- Libro Consigliato: Fowler et al. Refactoring: Improving the Design of Existing Code. Pearson Addison-Wesley. 1999

1

Prof. Tramontana - Marzo 2019

2

Prof. Tramontana - Marzo 2019

# Refactoring

- L'idea del refactoring è di riconoscere che è difficile ottenere fin dall'inizio una progettazione adeguata (e un codice adeguato) e quindi si deve cambiare
- Questi cambiamenti permettono quindi di incorporare più facilmente nuovi requisiti, o requisiti che cambiano
- Il refactoring fornisce tecniche per evolvere la progettazione in piccoli passi
- Vantaggi
  - Spesso la dimensione del codice si riduce dopo il refactoring
  - Le strutture complicate si trasformano in strutture più semplici più facili da capire e manutenere
  - Si evita di introdurre un debito tecnico all'interno della progettazione

3

Prof. Tramontana - Marzo 2019

# Perché Fare Refactoring

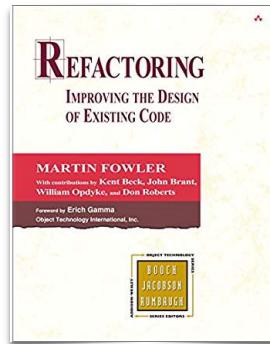
- Migliora la progettazione del sistema software
  - Senza refactoring la progettazione si deteriora mano a mano che si apportano modifiche al codice
- Rende il codice più facile da capire, perché la struttura del codice viene migliorata, il codice duplicato rimosso, etc.
- Aiuta a scoprire bug, fare refactoring promuove la comprensione profonda del codice e questo aiuta il programmatore a trovare bug o anticipare potenziali bug
- Permette di programmare più velocemente, perché una buona progettazione è più facile da cambiare

4

Prof. Tramontana - Marzo 2019

# Come Si Fa Refactoring

- Usare tecniche di refactoring [Fowler]
- Fare test costantemente: prima scrivere i test, dopo fare refactoring e verificare che tutti i test siano ancora superati
- Se un test non è superato, il refactoring ha compromesso qualcosa che funzionava, si è subito avvertiti e bisogna subito intervenire
- [Fowler] Fowler. Refactoring Improving the Design of Existing Code. Addison-Wesley



5

Prof. Tramontana - Marzo 2019

## (1) Estrai Metodo

- Un frammento di codice può essere raggruppato. Far diventare quel frammento un metodo il cui nome spiega lo scopo del frammento

```
public void printDovuto(double amount) {  
    printBanner();  
    System.out.println("nome: " + nome);  
    System.out.println("tot: " + somma);  
}
```

- Diventa

```
public void printConto(double somma) {  
    printBanner();  
    printDettagli(somma);  
}  
public void printDettagli(double amount) {  
    System.out.println("nome: " + nome);  
    System.out.println("tot: " + somma);  
}
```

6

Prof. Tramontana - Marzo 2019

## Estrai Metodo: Motivazioni

- Cercare metodi lunghi o codice che necessita di commenti per essere comprendibile
- Ridurre la lunghezza dei metodi, poiché metodi piccoli sono con più probabilità utilizzabili richiamandoli da altri metodi
- Scegliere un buon nome per i metodi piccoli, che esprime ciò che il metodo fa
- I metodi di alto livello, quelli che invocano i metodi piccoli, sono più facili da comprendere, sembrano essere costituiti da una sequenza di commenti
- Fare overriding (ridefinire nella sottoclasse) è più semplice se si hanno metodi piccoli

7

Prof. Tramontana - Marzo 2019

## Estrai Metodo: Meccanismi

- Creare un nuovo metodo il cui nome comunica l'intenzione del metodo
- Copiare il codice estratto dal metodo sorgente al nuovo metodo
- Guardare se il codice estratto ha variabili locali al metodo sorgente e far diventare tali variabili parametri del metodo nuovo
- Se alcune variabili sono usate solo all'interno del codice estratto farle diventare variabili temporanee del nuovo metodo
- Se una variabile locale al metodo sorgente è modificata dal codice estratto, vedere se è possibile far sì che il metodo estratto sia una query e assegnare il risultato alla variabile locale del metodo sorgente
- Sostituire nel metodo sorgente il codice estratto con una chiamata al metodo nuovo

8

Prof. Tramontana - Marzo 2019

## Esempio Estrai Metodo

```
public class Ordine {  
    private double importo;  
    public double getImporto() {  
        return importo;  
    }  
  
    public class Stampe {  
        private ArrayList<Ordine> ordini;  
        private String nome = "Mike";  
  
        public void printDovuto() {  
            Iterator<Ordine> i = ordini.iterator();  
            double tot = 0;  
  
            // scrivi banner  
            System.out.println("-----");  
            System.out.println("- Cliente Dare -");  
            System.out.println("-----");  
  
            // calcola totale  
            while (i.hasNext()) {  
                Ordine o = i.next();  
                tot += o.getImporto();  
            }  
  
            // scrivi dettagli  
            System.out.println("nome: " + nome);  
            System.out.println("tot: " + tot);  
        }  
    }  
}
```

```
public class Stampe2 {  
    private ArrayList<Ordine> ordini;  
    private String nome = "Mike";  
  
    public void printDovuto() {  
        printBanner();  
        double tot = getTotale();  
        printDettagli(tot);  
    }  
  
    public double getTotale() {  
        Iterator<Ordine> i = ordini.iterator();  
        double tot = 0;  
        while (i.hasNext()) {  
            Ordine o = i.next();  
            tot += o.getImporto();  
        }  
        return tot;  
    }  
  
    public void printBanner() {  
        System.out.println("-----");  
        System.out.println("- Cliente Dare -");  
        System.out.println("-----");  
    }  
  
    public void printDettagli(double somma) {  
        System.out.println("nome: " + nome);  
        System.out.println("tot: " + somma);  
    }  
}
```

Prof. Tramontana - Marzo 2019

## (2) Sostitisci Temp Con Query

- Una variabile temporanea (temp) è usata per tenere il risultato di un'espressione
- Estrarre l'espressione ed inserirla in un metodo. Sostituire tutti i riferimenti a temp con la chiamata al metodo che incapsula l'espressione. Il nuovo metodo può essere richiamato anche altrove

```
double prezzoBase = quantita * prezzo;
```

- Diventa

```
private double prezzoBase() {  
    return quantita * prezzo;  
}
```

10

Prof. Tramontana - Marzo 2019

## Sostitisci T Con Q: Motivazioni

- Variabili temp sono temporanee e locali. Possono essere viste solo all'interno del contesto del metodo e per questo inducono ad avere metodi lunghi, perché è il solo modo per raggiungere tali variabili
- Con la sostituzione effettuata tramite il refactoring, qualsiasi metodo può avere il dato
- Spesso si effettua il refactoring (2) Sostitisci Temp con Query prima di (1) Estrai Metodo
- Questo refactoring è facile se temp è assegnata solo una volta

## Sostitisci T Con Q: Meccanismi

- Cercare una variabile temporanea assegnata solo una volta
- Dichiare temp final
- Compilare (per verificare che è assegnata una volta sola)
- Estrarre la parte destra dell'assegnazione e creare un metodo

## Esempio Sostitisci Temp Con Query

```
private double quantita, prezzo;

public double getPrezzo1() {
    double prezzoBase = quantita * prezzo;
    double sconto;
    if (prezzoBase > 1000) sconto = 0.95;
    else sconto = 0.98;
    return prezzoBase * sconto;
}
```

- Diventa, sostituendo entrambe le variabili `prezzoBase` e `sconto`

```
private double quantita, prezzo;

public double getPrezzo2() {
    return prezzoBase() * sconto();
}

private double prezzoBase() {
    return quantita * prezzo;
}

private double sconto() {
    if (prezzoBase() > 1000) return 0.95;
    return 0.98;
}
```

13

Prof. Tramontana - Marzo 2019

## (3) Dividi Variabile Temp

- Una variabile temporanea è assegnata più di una volta, ma non è una variabile assegnata in un loop o usata per accumulare valori

- Usare una variabile separata per ciascun assegnamento

```
double temp = 2 * (height + width);
System.out.println(temp);
temp = height * width;
System.out.println(temp);
```

- Diventa

```
final double perim = 2 * (height + width);
System.out.println(perim);
final double area = height * width;
System.out.println(area);
```

14

Prof. Tramontana - Marzo 2019

## Dividi Var Temp: Motivazioni

- Le variabili temporanee (`temp`) hanno vari usi. Alcuni usi portano ad assegnare `temp` più volte, es. variabili che cambiano ad ogni passata di un ciclo (queste assegnazioni multiple sono ok)
- Variabili `temp` che tengono il risultato di un metodo lungo, per essere usate dopo, dovrebbero essere assegnate una volta soltanto
- Se sono assegnate più di una volta allora hanno più di una responsabilità all'interno del metodo. Ogni variabile dovrebbe avere una sola responsabilità e dovrebbe essere sostituita con una `temp` per ciascuna responsabilità
- Usare una stessa `temp` per due cose diverse confonde il lettore

15

Prof. Tramontana - Marzo 2019

## Dividi Var Temp: Meccanismi

- Cambiare il nome della variabile `temp` al momento della dichiarazione e alla sua prima assegnazione
- Dichiarare la nuova `temp` come `final`
- Cambiare tutti i riferimenti a `temp` fino alla sua seconda assegnazione
- Dichiarare una nuova `temp` per la seconda assegnazione
- Compilare e testare
- Ripetere per singoli passi, ogni passo rinomina una dichiarazione e cambia i riferimenti fino alla prossima assegnazione

16

Prof. Tramontana - Marzo 2019

# Esempio Dividi Var Temp

```
private double primaryForce, secondaryForce, mass, delay;

public double getDistanceTravelled1(int time) {
    double result;
    double acc = primaryForce / mass; // prima assegnazione
    int primaryTime = (int) Math.min(time, delay);
    result = 0.5 * acc * primaryTime * primaryTime;
    int secondT = (int) (time - delay);
    if (secondT > 0) {
        double primaryVel = acc * delay;
        acc = (primaryForce + secondaryForce) / mass; // seconda assegnazione
        result += primaryVel * secondT + 0.5 * acc * secondT * secondT;
    }
    return result;
}
```

- Diventa

```
public double getDistanceTravelled2(int time) {
    double result;
    final double primAcc = primaryForce / mass;
    int primaryTime = (int) Math.min(time, delay);
    result = 0.5 * primAcc * primaryTime * primaryTime;
    int secondT = (int) (time - delay);
    if (secondT > 0) {
        double primaryVel = primAcc * delay;
        final double secondAcc = (primaryForce + secondaryForce) / mass;
        result += primaryVel * secondT + 0.5 * secondAcc * secondT * secondT;
    }
    return result;
}
```

## Sviluppo Agile [Cockburn 2002]

- Sono più importanti auto-organizzazione, collaborazione, comunicazione tra membri del team e adattabilità del prodotto rispetto ad ordine e coerenza delle attività del progetto
- Privilegiare
  - Individui *rispetto* a processi e strumenti
  - Disponibilità di software funzionante *rispetto* alla documentazione
  - Collaborazione con il cliente *rispetto* alla negoziazione dei contratti
  - Pronta risposta ai cambiamenti *rispetto* all'esecuzione di un piano
- Agilità
  - Considerare positivamente le richieste di cambiamento anche in fase avanzata di sviluppo
  - Fornire release del sistema software funzionante frequentemente
  - Costruire sistemi software con gruppi di persone motivate
  - Continua attenzione all'eccellenza tecnica

E. Tramontana - Processo XP - 17-Mar-10 1

## Extreme Programming (XP) [Beck 2000]

- Approccio basato sullo sviluppo e la consegna di piccoli incrementi di funzionalità
  - Solo 2 settimane per lo sviluppo degli incrementi
  - Piccoli gruppi di sviluppatori (da 2 a 12 persone)
  - Costante miglioramento del codice
  - Poca documentazione: uso di Story Card e CRC (Class Responsibility Collabor)
  - Enfasi su comunicazione diretta tra persone
  - Iterazioni corte e di durata costante
  - Involgimento di sviluppatori, clienti e manager
  - Testabilità dei prodotti e prodotti testati sin dall'inizio
- Adatto per progetti in cui
  - I requisiti non sono stabili, XP è fortemente adattativo
  - I rischi sono grandi, es. tempi di consegna brevi, software innovativo per gli sviluppatori

E. Tramontana - Processo XP - 17-Mar-10 2

## Extreme Programming (XP)

- Princìpi di XP
  - Avere feedback rapidamente
  - Assumere la semplicità
  - Cambiamenti incrementali
  - Supportare i cambiamenti
  - Produrre lavoro di qualità
- Libro Consigliato
  - Beck. Extreme Programming Explained. Addison-Wesley
- Siti web
  - [www.xprogramming.com](http://www.xprogramming.com)
  - [www.extremeprogramming.org](http://www.extremeprogramming.org)

E. Tramontana - Processo XP - 17-Mar-10 3

## 12 Pratiche di XP [Beck]

- Gioco di pianificazione
- Piccole release
- Metafora
- Testing
- Refactoring
- Pair Programming (programmazione a coppie)
- Cliente in sede
- Design semplice
- Possesso del codice collettivo
- Integrazione continua
- Settimana di 40 ore
- Usare gli standard per il codice

E. Tramontana - Processo XP - 17-Mar-10 4

# Story Card

- Storie utente (story card) = casi d'uso leggeri
- Dimensioni card 5" x 3" circa 12x7 cm
- Descrizione storie: 2-3 frasi su una card che
  - Sono importanti per il cliente e sono scritte dal cliente
  - Possono essere testate
  - Permettono di ricavare una stima del loro tempo di sviluppo
  - Possono essere associate a priorità
- Template per story card (ovvero campi di una story card)
  - Data, Numero, Priorità, Tempo stimato, Riferimenti
  - Descrizione requisito
  - Lista di task per ciascun requisito, ovvero ciò che lo sviluppatore dovrà fare
  - Note

E. Tramontana - Processo XP - 17-Mar-10 5

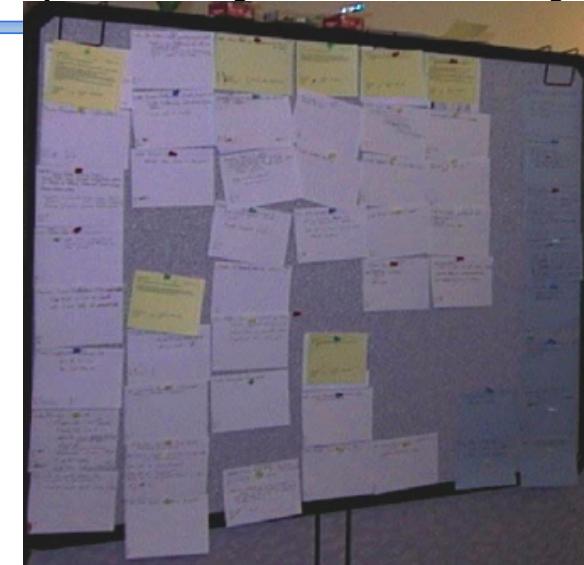
Titolo progetto Gestione Immagini				Autore Jim	Rigo piano progetto 1	Priorità 1
<b>Storia</b>						
Mostrare le immagini (jpg, png) presenti su una cartella del file system su una griglia di 10x6 immagini, indipendentemente dalla risoluzione dello schermo.						
<b>Stimatore</b>						
Jack	Tempo	3 giorni	Titolo progetto Gestione Immagini	Autore Jim	Rigo piano progetto 2	Priorità 1
<b>Storia</b>						
Mostrare le immagini scalate (ridimensionate) rispettando le proporzioni iniziali delle immagini.						
<b>Stimatore</b>						
Jack	Tempo st	1 giorno	Titolo progetto Gestione Immagini	Autore Jim	Rigo piano progetto 3	Priorità 1
<b>Storia</b>						
L'utente potrà selezionare immagini digitando lettere all'interno di una casella di testo. Le immagini selezionate saranno quelle i cui nomi di file contengono il testo inserito.						
<b>Stimatore</b>						
Jack	Tempo stimato	Data	Sviluppatori		Tempo impiegato	
		15-03-2011				

# Story Card

Customer Story and Task Card						
DATE:	3/19/98	TYPE OF ACTIVITY:	NEW: <input checked="" type="checkbox"/>	FIX: <input type="checkbox"/>	ENHANCE: <input type="checkbox"/>	FUNC. TEST: <input type="checkbox"/>
STORY NUMBER:	1275	PRIORITY:	USER: <input type="checkbox"/>	TECH: <input type="checkbox"/>		
PRIOR REFERENCE:				RISK:	TECH ESTIMATE:	
TASK DESCRIPTION:	SPLIT COLA: When the COLA rate chgs. in the middle of the BI/W Pay Period, we will want to pay the 1 <sup>ST</sup> week of the pay period at the OLD COLA rate and the 2 <sup>ND</sup> week of the Pay Period at the NEW COLA rate. Should occur automatically based on system design.					
NOTES:	For the OT, we will run a mf frame program that will pay or calc the COLA on the 2 <sup>ND</sup> week of OT. The plant currently retransmits the hours data for the 2 <sup>ND</sup> week exclusively so that we can calc COLA. This will come into the Model as a "2144" COLA.					
TASK TRACKING:	Gross Pay Adjustment. Create RM Boundary and Place in DEEntExcessCOLA BIN.					
Date	Status	To Do	Comments			

E. Tramontana - Processo XP - 17-Mar-10 6

# Board per Story Card = Story Board



17-Mar-10 8

# Story Board

## Prima settimana

### To do

Titolo progetto	Autore	Riporto piano progetto	Priorità
Gestione Immagini	Jim	2	1
<b>Storia</b>			
Mostrare le immagini scelte (ridimensionate) ripetendo le proporzioni iniziali delle immagini.			

Scadenza: 1 giorno

Tempo stimato: 15-03-2011

Sviluppatori: 1

Tempo impiegato: 0

## Seconda settimana

### To do

Titolo progetto	Autore	Riporto piano progetto	Priorità
Gestione Immagini	Jim	3	1
<b>Storia</b>			
L'utente potrà selezionare immagini digitando lettere all'interno di una casella di testo.			

Scadenza: 2 giorni

Tempo stimato: 15-03-2011

Sviluppatori: 1

Tempo impiegato: 0

## Terza settimana

### To do

Titolo progetto	Autore	Riporto piano progetto	Priorità
Gestione Immagini	Tim	8	3
<b>Storia</b>			
L'utente potrà continuare ad inserire testo, durante l'aggiornamento delle immagini dovuto alla selezione.			

Scadenza: 1 giorno

Tempo stimato: 15-03-2011

Sviluppatori: 1

Tempo impiegato: 0

## Done

Titolo progetto	Autore	Riporto piano progetto	Priorità
Gestione Immagini	Jack	1	1
<b>Storia</b>			
Mostrare le immagini Gif, png presenti su una cartella del file system su una griglia di 10x6 immagini, indipendentemente dalla risoluzione dello schermo.			

Scadenza: 3 giorni

Tempo stimato: 15-03-2011

Sviluppatori: 1

Tempo impiegato: 0

E. Tramontana - Processo XP - 17-Mar-10 9

# Piccole release

- Rendere ogni release il più piccola possibile
  - Tempo di sviluppo della release 2 settimane
- Effettuare un design semplice e sufficiente per la release corrente
- Piccole release forniscono agli sviluppatori
  - Feedback rapidamente
  - Un senso di: "ho ottenuto qualcosa di valido"
  - Rischio ridotto
  - La fiducia del cliente
  - Possibilità di fare aggiustamenti per requisiti che cambiano

E. Tramontana - Processo XP - 17-Mar-10 11

# Gioco di pianificazione

- Gli utenti (clienti) scrivono le storie (sono i requisiti)
- Gli sviluppatori stimano il tempo per lo sviluppo di ciascuna storia
  - Se le storie sono troppo complesse da stimare, ritornare dal cliente e far dividere le storie
- Gli utenti dividono, fondono e assegnano priorità alle storie
  - Riempiono 3 settimane scegliendo le storie
  - Non preoccuparsi delle dipendenze
- Gli addetti al business prendono decisioni su
  - Date per le release, contesto, priorità dei task
- Pianificare l'intera release (grossolanamente) e la nuova iterazione
  - Non pianificare troppo in avanti
- Per l'attuale release, gli sviluppatori:
  - Dividono ciascuna storia in task, stimano i task, ciascuno si impegna per realizzare un task
  - Vengono svolti prima i task più rischiosi

E. Tramontana - Processo XP - 17-Mar-10 10

# Metafora

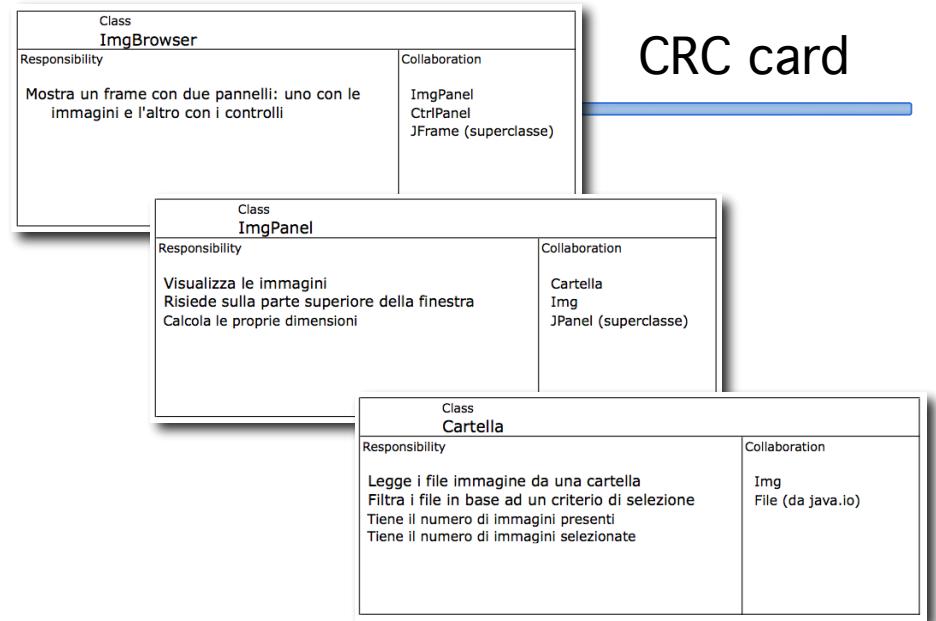
- Guidare il progetto con una singola metafora
  - Es.: "La UI è un desktop"
  - Deve rappresentare l'architettura
    - Rende le discussioni più semplici
  - Il cliente deve essere a suo agio con essa

E. Tramontana - Processo XP - 17-Mar-10 12

# Design Semplice

- Il giusto design per il software si ha quando
  - Passa i test
  - Non ha parti duplicate
  - Esprime ciascuna intenzione importante per i programmatori
  - Ha il numero più piccolo di classi e metodi
- Non preoccuparsi di dover apportare cambiamenti dopo
  - Fare la cosa più semplice che può funzionare
  - Paga quanto usi
- Usare le CRC card (Class Responsibility Collaboration) per documentare il design
  - Permettono di ragionare meglio in termini di oggetti
  - Contribuiscono a fornire una visione complessiva del sistema

E. Tramontana - Processo XP - 17-Mar-10 13



# CRC card

## Testing

- Si testa tutto ciò che potenzialmente può andar male, per tutto il tempo
  - Si eseguono i test più volte al giorno, non appena è stato prodotto del nuovo codice
- I test sono la specifica dei requisiti!
  - Una specifica in formato eseguibile!
- Due tipi di test
  - Test funzionali
  - Unit test

E. Tramontana - Processo XP - 17-Mar-10 15

## Test

- Test funzionali
  - Scritti dall'utente (punto di vista dell'utente)
  - Effettuati da: utenti, sviluppatori e team di testing
  - Automatizzati
  - Eseguiti almeno giornalmente
  - Sono una parte della specifica dei requisiti, quindi documentano i requisiti
- Unit test
  - Scritti dagli sviluppatori (punto di vista del programmatore)
  - Scritti prima della codifica (TDD, Test Driven Development) ed anche dopo la codifica
  - Supportano design, codifica, refactoring e qualità

E. Tramontana - Processo XP - 17-Mar-10 16

# Pair Programming

- Programmatori esperti e motivati
- Ruolo di uno dei partner
  - Usa il mouse e la tastiera
  - Pensa al miglior modo di implementare il metodo
- Ruolo dell'altro
  - L'approccio funzionerà?
  - Pensa ai test
  - Potrebbe essere fatto più semplicemente?
- Scambio dei partner
- Pair programming aiuta la disciplina, sparge la conoscenza sul sistema



E. Tramontana - Processo XP - 17-Mar-10 17

# Possesso del codice collettivo

- Chiunque può aggiungere qualunque codice su qualunque parte del sistema
- Unit test proteggono le funzionalità del sistema
- Chiunque trova un problema lo risolve
- Ciascuno è responsabile per l'intero sistema

E. Tramontana - Processo XP - 17-Mar-10 18

# Integrazione continua

- Integrazione del codice testato ogni poche ore (max un giorno)
- Tutti gli unit test devono essere superati
- Se un test fallisce la coppia che ha prodotto il codice deve ripararlo
- Se non può ripararlo, buttare il codice e ricominciare

# 40 ore a settimana

- Se per te non è possibile fare il lavoro in 40 ore, allora hai troppo lavoro
- 40 ore a settimana ti lasciano “fresco” per risolvere i problemi
- Previene l'inserimento di errori difficili da trovare
- Pianificazioni frequenti evitano a ciascuno di avere troppo lavoro
- Ore extra di lavoro è sintomo di un problema serio

E. Tramontana - Processo XP - 17-Mar-10 19

E. Tramontana - Processo XP - 17-Mar-10 20

## Cliente sul sito

- Scrive i test funzionali
- Stabilisce priorità e fornisce il contesto per le decisioni dei programmatori
- Risponde alle domande
- Porta avanti il suo proprio lavoro
- Se non puoi avere il cliente sul sito, forse il progetto non è così importante?

## Standard di codifica

- Costruzioni complicate (per il design) non sono permesse
  - Mantenere le cose semplici
- Il codice appare uniforme
  - Più facile da leggere
- Usare tutti la stessa convenzione, così non si ha necessità di riformattare il codice, sapere come usare
  - Spazi e Tab per indentazione
  - Posizione parentesi graffe
  - Scelta di nomi classi, metodi, attributi
  - Posizione commenti

Vedere convenzioni per linguaggio Java!

<http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html>

## Refactoring

- Refactoring significa migliorare la struttura del codice senza influenzarne il comportamento
- Fatto in piccoli passi
- Supportato dagli unit test, design semplice e pair programming
- Puntare a codice senza ripetizioni
- Refactoring fatto in coppia dà più coraggio

## In breve

- XP focalizza sul codice
  - Fare solo le cose che sveltiscono la produzione del codice
  - Codifica e test
- XP si orienta sulla gente
  - La conoscenza del sistema è trasferita attraverso la comunicazione tra la gente
- XP è leggero
  - Rimuovere i costi aggiuntivi
  - Creare prodotti di qualità tramite test rigorosi
- I principi di XP non sono nuovi

# Classi E Oggetti

- Ogni oggetto è l'istanza di una classe, e ogni classe ha zero o più istanze. Le classi sono statiche, quindi per esse l'esistenza, la semantica e le relazioni sono fissate prima dell'esecuzione del programma
- La classe per ciascun oggetto è statica, ovvero una volta che l'oggetto è creato la classe è fissata. Gli oggetti sono creati e distrutti durante l'esecuzione di un'applicazione
- Le classi formano il vocabolario del dominio del problema. Gli oggetti insieme interagiscono per soddisfare i requisiti del problema
- Quante istanze ho bisogno per la classe Pagamenti?
  - Potrei avere una istanza per tutta l'applicazione, o una istanza per ciascun file letto
  - Se decido di avere una sola istanza, come posso vietare la creazione di più istanze

1

Prof. Tramontana - Marzo 2019

```
public class MainPagam { // versione 0.1
    public static void main(String[] args) {
        Pagamenti p = new Pagamenti(); // p è una istanza di Pagamenti
        try {
            p.leggiFile("csvfiles", "Importi.csv"); // lettura primo file
            // p contiene tutti i valori letti dal file
        } catch (IOException e) {
        }
        System.out.println("totale: " + p.calcolaSomma());
        System.out.println("max: " + p.calcolaMassimo());
    }
}
```

2

Prof. Tramontana - Marzo 2019

```
public class MainPagam { // versione 0.2
    public static void main(String[] args) {
        Pagamenti p = new Pagamenti(); // p e' l'unica istanza
        try {
            p.leggiFile("csvfiles", "Importi.csv"); // lettura primo file
        } catch (IOException e) {
        }
        System.out.println("file 1 totale: " + p.calcolaSomma());
        System.out.println("file 1 max: " + p.calcolaMassimo());

        try {
            p.leggiFile("csvfiles", "PagMarzo.csv"); // lettura secondo file
            // p adesso contiene tutti i valori letti da entrambi i file
        } catch (IOException e) {
        }
        System.out.println("file 1 e 2 totale: " + p.calcolaSomma());
        System.out.println("file 1 e 2 max: " + p.calcolaMassimo());
    }
}
```

- Poiché il metodo `leggiFile()` della classe `Pagamenti` non cancella i valori in lista, posso leggere più file e inserirli nella stessa lista con varie chiamate a `leggiFile()`
  - Singola responsabilità per il metodo

3

Prof. Tramontana - Marzo 2019

```
public class MainPagam { // versione 0.3
    public static void main(String[] args) {
        Pagamenti p1 = new Pagamenti(); // prima istanza
        Pagamenti p2 = new Pagamenti(); // seconda istanza

        try {
            p1.leggiFile("csvfiles", "Importi.csv"); // lettura primo file
            p2.leggiFile("csvfiles", "PagMarzo.csv"); // lettura secondo file
        } catch (IOException e) {
        }
        System.out.println("file 1 totale: " + p1.calcolaSomma());
        System.out.println("file 1 max: " + p1.calcolaMassimo());
        System.out.println("file 2 totale: " + p2.calcolaSomma());
        System.out.println("file 2 max: " + p2.calcolaMassimo());
    }
}
```

- E' possibile usare varie istanze di `Pagamenti` per mettere valori provenienti da file diversi
- Supponiamo che l'applicazione non debba avere più di una istanza di `Pagamenti`, occorre il Design Pattern Singleton per vietare la creazione di più istanze

4

Prof. Tramontana - Marzo 2019

# Design Pattern Singleton

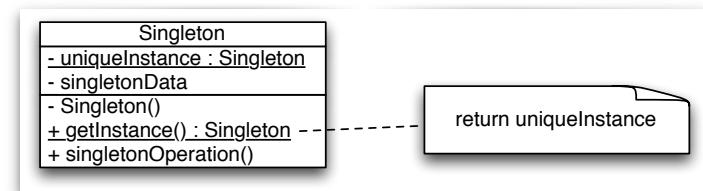
- Intento
  - Assicurare che una classe abbia una sola istanza e fornire un punto di accesso globale all'istanza
- Motivazione
  - Alcune classi dovrebbero avere esattamente una istanza in tutta l'applicazione, es. uno spooler di stampa, un file system, un window manager, una lista clienti, etc.
  - Una variabile globale rende un oggetto accessibile ma non proibisce di avere più oggetti per una classe
  - La classe stessa dovrebbe essere responsabile di tener traccia del suo unico punto di accesso

5

Prof. Tramontana - Marzo 2019

# Singleton

- Soluzione
  - La classe che deve essere un *Singleton* dovrà implementare un'operazione `getInstance()` sulla classe (ovvero, in Java è un metodo static) che ritorna l'unica istanza creata
  - La classe *Singleton* è responsabile per la creazione dell'istanza
  - Il costruttore della classe *Singleton* è privato, così da non permettere la creazione tramite new ad altre classi



6

Prof. Tramontana - Marzo 2019

## Esempio Classe Singleton Fib

```
// Classe Singleton che tiene una lista di
// interi
public class Fib {
    // l'unica istanza e' tenuta da obj
    private static Fib obj = new Fib();
    private int[] x={1, 2, 3, 5, 8, 13, 21,
    34, 55, 89, 144};
    private int i;
    private Fib() {
        i = 3;
    }
    public static Fib getInstance() {
        return obj; // restituisce l'istanza
    }
    public int getValue() {
        if (i<11) i++;
        return x[i-1];
    }
    public void revert() {
        i = 0;
    }
}

public class TestFib {
    public static void main(String[] args) {
        // richiede una istanza di Fib
        Fib f = Fib.getInstance();
        System.out.print("f "+f.getValue());
        System.out.println(" "+f.getValue());

        // richiede una nuova istanza
        Fib f2 = Fib.getInstance();
        System.out.print("f2 "+f2.getValue());
        System.out.println(" "+f2.getValue());

        // Si ha un errore a compile-time con:
        // Fib f3 = (Fib) f2.clone();
        // Fib f4 = new Fib();
    }
}
```

Output dell'esecuzione  
f 5 8  
f2 13 21

7

Prof. Tramontana - Marzo 2019

## Esempio Classe Logs

```
public class Logs {
    private static Logs obj; // Classe Singleton
    private List<String> l; // obj tiene l'istanza
                           // e tiene i dati da registrare

    private Logs() { // il costruttore è privato
        empty();
    }
    public static Logs getInstance() { // restituisce l'unica istanza
        if (obj == null) obj = new Logs(); // crea l'istanza se non presente
        return obj;
    }
    public void record(String s) { // accoda il dato
        l.add(s);
    }
    public String dumpLast() { // restituisce l'ultimo dato
        return l.getLast();
    }
    public String dumpAll() { // restituisce tutti i dati
        String acc = "";
        for (String s : l) // s tiene ciascun elemento in lista, ad ogni passata
            acc = acc.concat(s);
        return acc;
    }
    public void empty() {
        l = new ArrayList<String>();
    }
}
```

8

Prof. Tramontana - Marzo 2019

# Test Per Classe Logs

```
public class TestLogs {  
    private Logs lg = Logs.getInstance();  
  
    public void testSingl() {  
        initLogs();  
        Logs lg2 = Logs.getInstance();  
        lg2.record("uno");  
        lg2.record("due");  
        if (lg.dumpLast().equals("due"))  
            System.out.println("OK test logs singl");  
        else  
            System.out.println("FAILED test logs singl");  
    }  
  
    public void testLast() {  
        initLogs();  
        if (lg.dumpLast().equals("three "))  
            System.out.println("OK test logs last");  
        else  
            System.out.println("FAILED test logs last");  
    }  
  
    public void testAll() {  
        initLogs();  
        if (lg.dumpAll().equals("one two three "))  
            System.out.println("OK test logs all");  
        else  
            System.out.println("FAILED test logs all");  
    }  
}
```

9

```
private void initLogs() {  
    lg.empty();  
    lg.record("one ");  
    lg.record("two ");  
    lg.record("three ");  
}  
  
public static void main(String[] args) {  
    TestLogs tl = new TestLogs();  
    tl.testSingl();  
    tl.testAll();  
    tl.testLast();  
}
```

Output dell'esecuzione  
OK test logs singl  
OK test logs all  
OK test logs last

Prof. Tramontana - Marzo 2019

# Considerazioni

- Grazie al Design Pattern Singleton, la classe Fib non può avere più di una istanza a runtime (lo stesso per la classe Logs)
- Si dice che la classe Fib implementa (o svolge) il ruolo di Singleton
- Esercizio: trasformare la classe Pagamenti in Singleton
- Conseguenze: se dopo aver realizzato Pagamenti come un Singleton, si volessero più istanze di Pagamenti, anziché una sola, il codice delle classi chiamanti rimarrebbe invariato, e la sola classe da modificare è la classe Pagamenti. La variante che permette un numero finito di istanze si chiama Multiton
- Principio delle Conseguenze Locali: un cambiamento in qualche punto del codice non dovrebbe causare problemi in altri punti
- Una variante Multiton di Pagamenti potrebbe associare una istanza a ciascuna cartella, la decisione sull'istanziazione dipende quindi dall'aver già creato (o meno) un'istanza per una data cartella. Implementare come esercizio il Multiton di Pagamenti

Prof. Tramontana - Marzo 2019

# Conseguenze Del Singleton

- La classe che è un *Singleton* ha pieno controllo di come e quando i client accedono al valore della sola istanza
- Evita che esistano variabili globali che tengono la sola istanza condivisa
- Permette di controllare il numero di istanze create in un programma, facilmente ed in un solo punto
- La soluzione è più flessibile rispetto a quella di usare static per tutte le operazioni e le variabili, poiché si può cambiare facilmente il numero di istanze consentite
- L'unico frammento di codice da modificare quando si vuol variare il numero di istanze create è quello della classe che è *Singleton*, mentre usando static si dovrebbero modificare tutte le invocazioni

11

Prof. Tramontana - Marzo 2019

# Riuso Di Classi

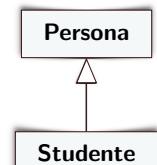
- Spesso si ha bisogno di classi simili
  - Si vuole cioè **riusare** classi esistenti per implementare attributi e metodi leggermente diversi
- **Non** è pratico copiare la classe originaria e modificarne attributi o metodi, si avrebbe una proliferazione di classi e tanto lavoro per il programmatore
- Il riuso delle classi esistenti deve avvenire
  - Senza dover modificare codice esistente (e funzionante)
  - In modo semplice per il programmatore

1

Prof. Tramontana - Marzo 2020

# Ereditarietà Classi

- Attraverso l'ereditarietà è possibile
    - Definire una nuova classe indicando solo cosa ha in più rispetto ad una classe esistente: ovvero attributi e metodi nuovi, e modificando i metodi esistenti
    - Esempio: una classe Persona ha nome e cognome (più vari metodi)
    - La classe Studente dovrebbe avere tutto ciò che Persona ha (attributi e metodi) e nuovi attributi e metodi
    - Studente aggiunge esami, voti, etc.
    - La classe Studente eredita da Persona
- ```
public class Studente extends Persona { ... }
```
- Studente è sottoclasse di Persona e Persona è superclasse di Studente



2

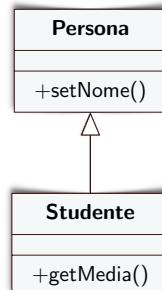
Prof. Tramontana - Marzo 2020

# Ereditarietà Classi

- La sottoclasse
  - Eredita tutti i metodi e gli attributi della superclasse e può usarli come se fossero definiti localmente
  - Aggiunge altri metodi
  - Può ridefinire i metodi della superclasse
  - Non può eliminare metodi o attributi della superclasse
- Esempio: la classe Studente
  - Può usare tutti i metodi della classe Persona, es. setNome()
  - Può aggiungere metodi, es. getMedia()

3

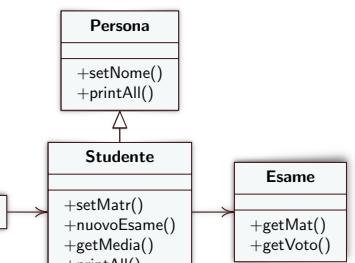
Prof. Tramontana - Marzo 2020



```
public class Persona {
    private String nome, co;
    public void setNome(String n, String c) {
        nome = n; co = c;
    }
    public void printAll() {
        System.out.println(nome + " " + co);
    }
}

public class Studente extends Persona {
    private String matr;
    private List<Esame> esami = new ArrayList<>();
    public void setMatr(String m) { matr = m; }
    public void nuovoEsame(String m, int v) {
        Esame e = new Esame(m, v);
        esami.add(e);
    }
    public float getMedia() {
        if (esami.isEmpty()) return 0;
        float sum = 0;
        for (Esame e : esami) sum += e.getVoto();
        return sum / esami.size();
    }
    public void printAll() {
        super.printAll();
        System.out.println("matr: " + matr);
        for (Esame e : esami)
            System.out.println(e.getMatr() + ": " + e.getVoto());
        System.out.println("media: " + getMedia());
    }
}
```

```
public class Esame {
    private String mat;
    private int voto;
    public Esame(String n,int v){
        mat = n; voto = v;
    }
    public String getMat() {
        return mat;
    }
    public int getVoto() {
        return voto;
    }
}
```



4

Prof. Tramontana - Marzo 2020

# Ereditarietà

- Visibilità
  - Ciò che è `private` è visibile solo alla classe, non alla sottoclasse
  - Ciò che è `public` è visibile a tutti, anche alla sottoclasse
  - Ciò che è `protected` è visibile alle sottoclassi ma non a tutte le altre classi
- Il nome della sottoclasse deve comunicare a quale classe è simile e come è diversa
- Classi che servono come radici di una gerarchia devono avere nomi concisi, altrimenti si può dare un nome più lungo

5

Prof. Tramontana - Marzo 2020

# Interfacce

- In Java una interfaccia riprende il concetto di interfaccia di sistemi orientati ad oggetti
  - Non fornisce una implementazione per i metodi
  - Permette di definire un tipo
  - Elenca le signature dei metodi `public` (senza corpo dei metodi)
  - Posso solo dichiarare i metodi
  - Niente attributi non inizializzati, niente costruttori

```
public interface IAccount {  
    public void setBalance();  
}
```

6

Prof. Tramontana - Marzo 2020

# Classi Astratte

- Una classe astratta è una classe parzialmente implementata. Alcuni metodi sono implementati, altri no (e quest'ultimi sono `abstract`)
- Un metodo `abstract` (senza implementazione) è utile poiché
  - Altri metodi della stessa classe (implementati) possono invocarlo
  - I client si aspettano di poterlo invocare
  - Forza le sottoclassi (concrete) ad implementare il metodo `abstract`
- La classe astratta non può essere istanziata

```
public abstract class Libro {  
    private String autore;  
    public abstract void insert();  
    public String getAutore() {  
        return autore;  
    }  
}
```

7

Prof. Tramontana - Marzo 2020

# Classi Ed Interfacce

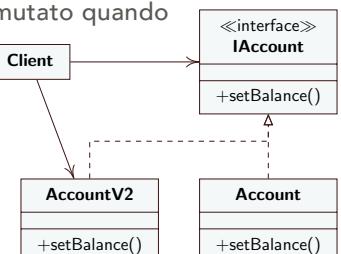
- Una classe può implementare un'interfaccia, ovvero, la classe fornisce un'implementazione dei metodi definiti dall'interfaccia
- Non è possibile istanziare interfacce
- Tramite l'interfaccia i client sanno cosa possono invocare. Per istanziazioni di oggetto e invocazioni di metodo, si può usare una qualsiasi delle implementazioni disponibili per l'interfaccia
- Un client che usa un'interfaccia rimane immutato quando l'implementazione dell'interfaccia cambia

```
public class AccountV2 implements IAccount {  
    public void setBalance() {  
    }  
  
    public class M {  
        public void main(String[] args) {  
            IAccount a = new AccountV2();  
            a.setBalance();  
        }  
    }  
}
```

8

8

Prof. Tramontana - Marzo 2020

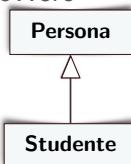


# Compatibilità Fra Tipi

- L'ereditarietà permette di definire una classificazione di tipi. Una sottoclasse è un sottotipo compatibile con la superclasse, ovvero una sottoclasse è anche ciò che è la superclasse
- Esempio, si abbia Studente sottoclasse di Persona
  - Il tipo Studente è compatibile con il tipo Persona
  - La classe Studente fa tutto ciò che fa Persona, ed altre cose oltre quelle che fa Persona
- Una sottoclasse può prendere il posto della superclasse
- Esempio: si può usare un'istanza di Studente al posto di una di Persona. Dove compare p.setNome() con p di tipo Persona posso sostituire s.setNome() con s di tipo Studente
- Attenzione non vale il contrario, non si può usare la superclasse dove si usava la sottoclasse

9

Prof. Tramontana - Marzo 2020



```

public class Persona {
    public void setNome(String n, String c) { ... }
    public void printAll() { ... }
}

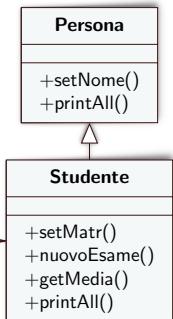
public class Studente extends Persona {
    public void setMatr(String m) { ... }
    public void nuovoEsame(String m, int v) { ... }
    public float getMedia() { ... }
    public void printAll() { ... }
}

public class MainEsami {
    public static void main(String[] args) {
        Studente s = new Studente();
        s.setNome("Alan", "Rossi"); // metodo della superclasse di s
        s.setMatr("M12345");
        s.nuovoEsame("Italiano", 8); // metodo della classe di s
        s.printAll(); // metodo della classe di s

        s.nuovoEsame("Fisica", 7);
        Persona p = s; // p e' dichiarato di tipo Persona
        p.printAll(); // a runtime p punta all'istanza s
    }
}
  
```

10

Prof. Tramontana - Marzo 2020



# Considerazioni

- La classe Studente eredita tutto ciò che ha Persona, inoltre ridefinisce il metodo printAll() (ovvero fa override), quindi modifica il comportamento di printAll() ereditato
  - super.printAll() permette di chiamare printAll() di Persona da Studente, super permette di accedere ai metodi della superclasse
- Nella classe MainEsami, la variabile p è di tipo Persona, ma punta a un'istanza di Studente
  - Chiamando su p il metodo printAll(), sarà eseguito printAll() di Studente
  - Su p non si può chiamare nuovoEsame(), poiché il tipo di p a compile time (Persona) non ha nuovoEsame()
  - Quindi, il compilatore non può far chiamare nuovoEsame() su p (nonostante p punterà a runtime ad un'istanza di Studente)

11

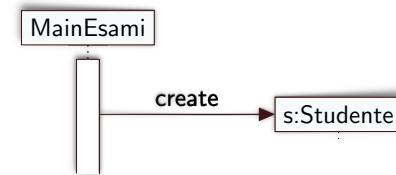
Prof. Tramontana - Marzo 2020

# Diagramma UML Di Sequenza

- Esempio di creazione di un'istanza della classe Studente

```

public class MainEsami {
    public static void main(String[] args) {
        Studente s = new Studente();
    }
}
  
```



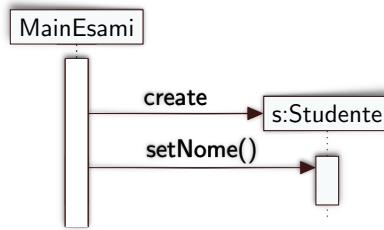
12

Prof. Tramontana - Marzo 2020

# Diagramma UML Di Sequenza

- Esempio di chiamata di metodo su un'istanza di Studente

```
public class MainEsami {
    public static void main(String[] args) {
        Studente s = new Studente();
        s.setNome("Alan", "Rossi"); // chiama metodo di s
    }
}
```



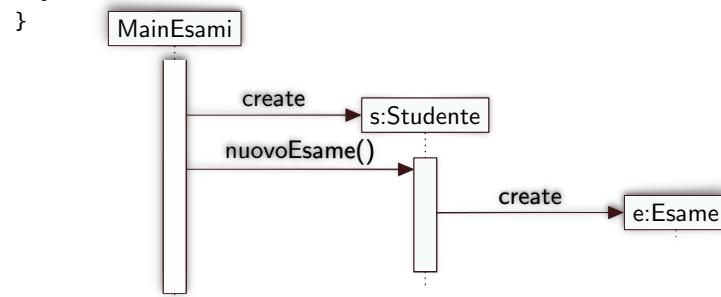
13

Prof. Tramontana - Marzo 2020

# Diagramma UML Di Sequenza

- Esempio di chiamate di metodo a cascata

```
public class MainEsami {
    public static void main(String[] args) {
        Studente s = new Studente();
        s.nuovoEsame("Italiano", 8);
    }
}
```

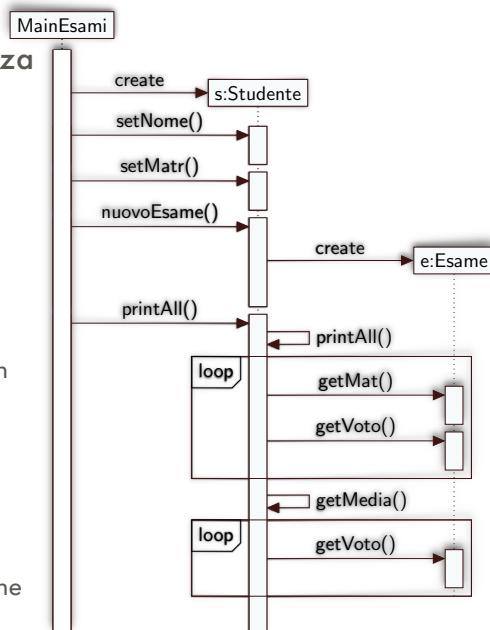


14

Prof. Tramontana - Marzo 2020

## Diagramma UML Di Sequenza

- Mostra interazioni fra oggetti
- L'asse temporale è inteso in verticale verso il basso
- In alto in orizzontale ci sono vari oggetti
- In ciascuna colonna se l'oggetto esiste è indicato con una linea tratteggiata, detta linea della vita, e se è attivo con una barra di attivazione
- Una chiamata di metodo è indicata da una freccia piena che va dalla barra di attivazione di un oggetto ad un altro

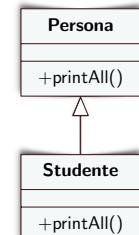


15

Prof. Tramontana - Marzo 2020

# Late Binding E Polimorfismo

```
public void m() {
    Persona p = new Persona();
    Studente s = new Studente();
    Persona px;
    if (i > 10)
        px = p;
    else
        px = s;
    px.printAll();
}
```



- `printAll()` invocato su `px` può assumere il comportamento definito in `Persona` o quello definito in `Studente`
- Il compilatore riconosce che `printAll()` è definito per `px` (qualunque sia l'istanza puntata)
- A runtime si decide quale `printAll()` eseguire, ovvero si ha late binding, ed il comportamento di `printAll()` è polimorfo

16

Prof. Tramontana - Marzo 2020

# Polimorfismo

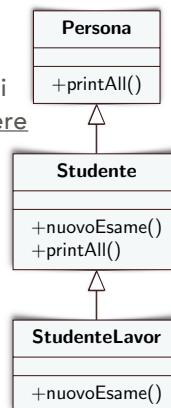
- Nei sistemi ad oggetti possono esistere metodi con lo stesso nome e la stessa signature (in classi diverse)
- Quando si usa l'ereditarietà e sono stati definiti metodi con lo stesso nome, la chiamata ad un metodo può avere effetti diversi, ovvero il comportamento è polimorfo

```
Studente s;  
// ...  
s.nuovoEsame("Maths", 8);  
s.printAll();
```

- La variabile s potrebbe tenere il riferimento ad un'istanza di Studente o di StudenteLavor, quale metodo nuovoEsame() sarà chiamato è deciso a runtime, in base all'istanza. Lo stesso vale quando si ha una variabile di tipo Persona e per il metodo printAll()

17

Prof. Tramontana - Marzo 2020



# Polimorfismo

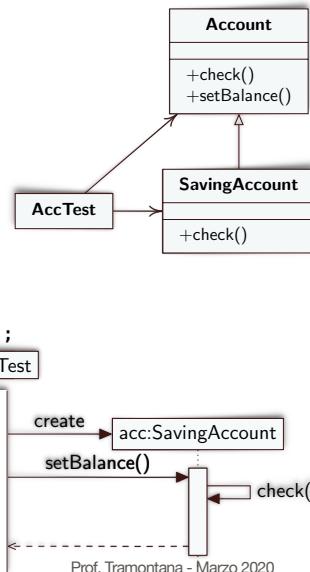
- Il polimorfismo è una caratteristica fondamentale dei sistemi ad oggetti
- Il late binding è tipico dei sistemi in cui esiste il polimorfismo
- Senza polimorfismo dovremmo inserire uno switch sul chiamante per valutare la classe di ciascuna istanza e chiamare il metodo corrispondente a tale classe

18

Prof. Tramontana - Marzo 2020

# Sottoclassi E Dispatch

```
public class Account {  
    protected float balance;  
    public void setBalance(float amount) {  
        System.out.println("in account set-balance");  
        if (check(amount))  
            balance = amount;  
    }  
    public boolean check(float amount) {  
        System.out.println("in account check");  
        return (balance + amount) >= 0;  
    }  
}  
  
public class SavingAccount extends Account {  
    public boolean check(float amount) {  
        System.out.println("in saving-account check");  
        return (balance + amount) >= 1000;  
    }  
}  
  
public class AccTest {  
    public static void main(String[] args) {  
        Account acc = new SavingAccount();  
        acc.setBalance(1234);  
    }  
}
```



19

Prof. Tramontana - Marzo 2020

# Dispatch

- Quale versione di check() è chiamata da setBalance()?
- Quando un metodo (setBalance()) è chiamato su un oggetto, viene controllato il suo tipo a runtime (SavingAccount)
- Si cerca il metodo setBalance() sul tipo a runtime: non trovato
- Se non trovato, si cerca la superclasse: Account
- Se trovato si esegue: Account.setBalance()
- Questo chiama check(), si cerca come prima
- Quindi si cerca prima su SavingAccount: trovato
- Si esegue SavingAccount.check()

20

Prof. Tramontana - Marzo 2020

# Tipi Di Variabili E Tipi A Runtime

- A compile time, si dichiara una variabile di un certo tipo, ed il tipo determina quali operazioni, quali signature, si possono usare. Tale tipo può essere una classe o un'interfaccia
- A runtime, l'oggetto riferito dalla variabile ha una sola implementazione che è sempre una classe, non un'interfaccia. Il tipo a runtime di un oggetto non cambia mai
- Una variabile di un tipo può tenere il riferimento ad un'istanza del sottotipo (la classe Studente è sottotipo di Persona)

```
Persona p = new Studente();
```

- Tramite cast forzo l'istanza p di tipo Persona sulla variabile s del sottotipo Studente

```
Studente s = (Studente) p;
```

- OK per il compilatore, OK a runtime solo se l'istanza p è del tipo di s
- Con s = (Studente) new Persona() si avrebbe a runtime

21

Prof. Tramontana - Marzo 2020

# Dependency Injection

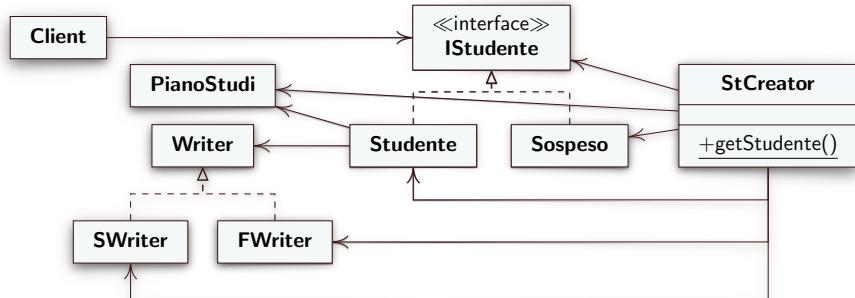
- Il design pattern Factory Method può essere usato per inserire le dipendenze (dependency injection) necessarie alle istanze di ConcreteProduct
- Tramite la Dependency Injection un oggetto (client) riceve altri oggetti da cui dipende, questi altri oggetti sono detti dipendenze
- La tecnica di Dependency Injection permette di **separare la costruzione delle istanze dal loro uso**
- Il client non crea l'istanza di cui ha bisogno
- Le dipendenze sono iniettate al client per mezzo di parametri nel suo costruttore. Questo permette di evitare complicazioni derivanti da metodi setter e da controlli per verificare che le dipendenze non siano null, di conseguenza il codice è più semplice
- L'oggetto che fa Dependency Injection si occupa di connettere (fa wiring di) varie istanze. In un unico posto vediamo le connessioni fra gli oggetti

13

Prof. Tramontana - Anriile 2021

# Esempio

- Si abbiano Writer e PianoStudi che sono dipendenze per Studente
- Studente riceve nel suo costruttore le istanze di Writer e PianoStudi
- Studente conosce solo il tipo Writer non i suoi sottotipi



14

Prof. Tramontana - Anriile 2021

# Design Pattern

- I design pattern sono strutture software (ovvero micro-architetture) per un piccolo numero di classi che descrivono soluzioni di successo per problemi ricorrenti
- Tali micro-architetture specificano le diverse classi ed oggetti coinvolti e le loro interazioni
- Si mira a riusare un insieme di classi, ovvero la soluzione ad un certo problema ricorrente, che spesso è costituita da più di una classe
- Durante la progettazione, le conseguenze sulle classi di varie scelte potrebbero non essere note, e le classi potrebbero diventare difficili da riusare o non esibire alcune proprietà
- Esistono tanti cataloghi di design pattern, per vari contesti
- Sistemi centralizzati, concorrenti, distribuiti, real-time, etc.

1

Prof. Tramontana - Marzo 2020

# Design Pattern

- Un design pattern descrive un problema di progettazione ricorrente che si incontra in specifici contesti e presenta una soluzione collaudata generica ma specializzabile
- Documentano soluzioni già applicate che si sono rivelate di successo per certi problemi e che si sono evolute nel tempo
- Aiutano i principianti ad agire come se fossero esperti
- Supportano gli esperti nella progettazione su grande scala
- Evitano di re-inventare concetti e soluzioni, riducendo il costo
- Forniscono un vocabolario comune e permettono una comprensione dei principi del design
- Analizzano le loro proprietà non-funzionali: ovvero, come una funzionalità è realizzata, es. affidabilità, modificabilità, sicurezza, testabilità, riuso

2

Prof. Tramontana - Marzo 2020

## Descrizione Di Un Pattern

- Un design pattern nomina, astrae ed identifica gli aspetti chiave di un problema di progettazione, le classi e le istanze che vi partecipano, i loro ruoli e come collaborano, ovvero la distribuzione delle responsabilità
- La descrizione include cinque parti fondamentali
- **Nome:** permette di identificare il design pattern con una parola e di lavorare con un alto livello di astrazione, indica lo scopo del pattern
- **Intento:** descrive brevemente le funzionalità e lo scopo
- **Problema** (Motivazione + Applicabilità): descrive il problema a cui il pattern è applicato e le condizioni necessarie per applicarlo
- **Soluzione:** descrive gli elementi (classi) che costituiscono il design pattern, le loro responsabilità e le loro relazioni
- **Conseguenze:** indicano risultati, compromessi, vantaggi e svantaggi nell'uso del design pattern

3

Prof. Tramontana - Marzo 2020

## Descrizione

- Nella sezione Problema della descrizione di un design pattern si parla di forze (come in fisica). Ovvero, le forze sono obiettivi e vincoli, spesso contrastanti, che si incontrano nel contesto di quel design pattern
- Altre parti della descrizione di un design pattern possono essere
- Esempi di utilizzo: illustrano dove il design pattern è stato usato
- Codice: fornisce porzioni di codice che lo implementano

4

Prof. Tramontana - Marzo 2020

# Organizzazione

- I design pattern sono organizzati sul catalogo (libro GoF) in base allo scopo
- Creazionali:** riguardano la creazione di istanze
  - Singleton, Factory Method, Abstract Factory, Builder, Prototype
- Strutturali:** riguardano la scelta della struttura
  - Adapter, Facade, Composite, Decorator, Bridge, Flyweight, Proxy
- Comportamentali:** riguardano la scelta dell'incapsulamento di algoritmi
  - Iterator, Template Method, Mediator, Observer, State, Strategy, Chain of Responsibility, Command, Interpreter, Memento, Visitor

5

Prof. Tramontana - Marzo 2020

# Design Pattern Creazionali

- Permettono di astrarre il processo di creazione oggetti: rendono un sistema indipendente da come i suoi oggetti sono creati, composti, e rappresentati
- Sono importanti se i sistemi evolvono per dipendere più su composizioni di oggetti che su ereditarietà tra classi
  - L'enfasi va dal codificare un insieme fissato di comportamenti verso un più piccolo insieme di comportamenti fondamentali componibili
- Incapsulano conoscenza sulle classi concrete che un sistema usa
- Nascondono come le istanze delle classi sono create e composte

6

Prof. Tramontana - Marzo 2020

# Factory Method

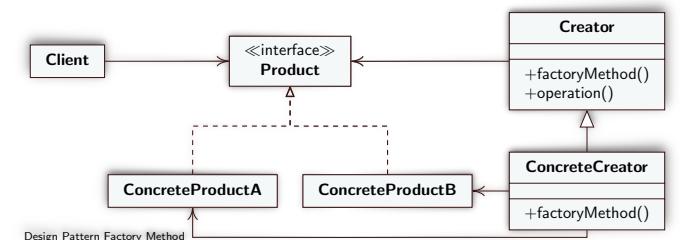
- Intento**
  - Definire una interfaccia per creare un oggetto, ma lasciare che le sottoclassi decidano quale classe istanziare. Factory Method permette ad una classe di rimandare l'istanziazione alle sottoclassi
- Problema**
  - Un framework usa classi astratte per definire e mantenere relazioni tra oggetti. Il framework deve creare oggetti ma conosce solo classi astratte che non può istanziare
  - Un metodo responsabile per l'istanziazione (detto factory, ovvero fabbricatore) incapsula la conoscenza su quale classe creare

7

Prof. Tramontana - Marzo 2020

# Factory Method

- Soluzione**
  - Product è l'interfaccia comune degli oggetti creati da factoryMethod()
  - ConcreteProduct è un'implementazione di Product
  - Creator dichiara il factoryMethod(), quest'ultimo ritorna un oggetto di tipo Product. Creator può avere un'implementazione si default del factoryMethod() che ritorna un certo ConcreteProduct
  - ConcreteCreator implementa il factoryMethod(), o ne fa override, sceglie quale ConcreteProduct istanziare e ritorna tale istanza

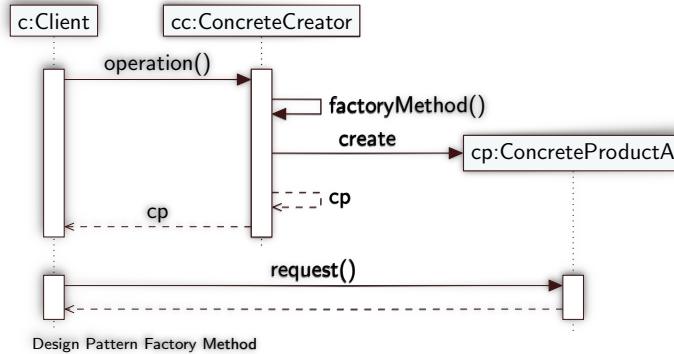


8

Prof. Tramontana - Marzo 2020

# Factory Method

- Soluzione: diagramma UML di sequenza, che illustra le interazioni fra i vari ruoli



9

Prof. Tramontana - Marzo 2020

```

public interface IStudente {
    public void nuovoEsame(String m, int v);
    public float getMedia();
}

public class Studente implements IStudente {
    private List<Esame> esami = new ArrayList<>();
    public void nuovoEsame(String m, int v) {
        Esame e = new Esame(m, v);
        esami.add(e);
    }
    public float getMedia() {
        if (esami.isEmpty()) return 0;
        float sum = 0;
        for (Esame e : esami) sum += e.getVoto();
        return sum / esami.size();
    }
}

public class Sospeso implements IStudente {
    private float media;
    public Sospeso(float m) {
        media = m;
    }
    public void nuovoEsame(String m, int v) {
        System.out.println("Non e' possibile sostenere esami");
    }
    public float getMedia() {
        return media;
    }
}

public interface IStudente {
    public void nuovoEsame(String m, int v);
    public float getMedia();
}

public class Client {
    public void registra() {
        IStudente s =
            StCreator.getStudente();
        s.nuovoEsame("Maths", 8);
    }
}

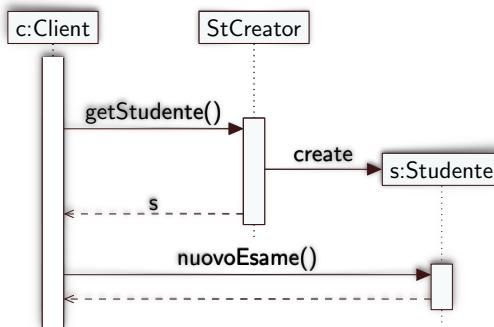
public class StCreator {
    private static float v = 0;
    public static IStudente getStudente() {
        if (v == 0)
            return new Studente();
        return new Sospeso(v);
    }
}
  
```

10

Prof. Tramontana - Marzo 2020

## Esempio Di Factory Method

- Nel precedente esempio di codice, l'interfaccia IStudente svolge il ruolo *Product*, le classi Studente e Sospeso svolgono il ruolo *ConcreteProduct*, e la classe StCreator svolge il ruolo *ConcreteCreator*



11

Prof. Tramontana - Marzo 2020

## Factory Method

- Varianti
  - Il ruolo Creator e ConcreteCreator sono svolti dalla stessa classe
  - Il factoryMethod() è un metodo static
  - Il factoryMethod() ha un parametro che permette al client di suggerire la classe da usare per creare l'istanza
  - Il factoryMethod() usa la *Riflessione Computazionale*, quindi Class.forName() e newInstance(), per eliminare le dipendenze dai ConcreteProduct, la classe istanziata sarà nota a runtime

```

try {
    Class<?> cls = Class.forName("Studente"); // Il nome della classe da istanziare e' una stringa
    Constructor<?> cnstr = cls.getConstructor(new Class[] {});
    return (IStudente) cnstr.newInstance();
} catch (InstantiationException | IllegalAccessException | IllegalArgumentException | InvocationTargetException | NoSuchMethodException | SecurityException e) {
    e.printStackTrace();
}
  
```
- Conseguenze
  - Il codice delle classi dell'applicazione conosce solo l'interfaccia Product e può lavorare con qualsiasi ConcreteProduct. I ConcreteProduct sono facilmente intercambiabili
  - Se si implementa una sottoclasse di Creator per ciascun ConcreteProduct da istanziare si ha una proliferazione di classi

12

Prof. Tramontana - Marzo 2020

# Object Pool

- Un object pool è una deposito di istanze già create, una istanza sarà estratta dal pool quando un client ne fa richiesta
  - Il pool può crescere o può avere dimensioni fisse
    - Dimensioni fisse: se non ci sono oggetti disponibili al momento della richiesta, non ne creo di nuovi
  - Il client restituisce al pool l'istanza usata quando non più utile
- Il design pattern Factory Method può implementare un object pool
  - I client fanno richieste, come visto prima per il Factory Method
  - I client dovranno dire quando l'istanza non è più in uso, quindi riusabile
  - Lo stato dell'istanza da riusare potrebbe dover essere ri-scritto
  - L'object pool dovrebbe essere unico -> uso un Singleton

E. Tramontana Pool, Dependency - 12 May 10 1

# Esempio codice Object Pool

```
import java.util.LinkedList;
// CreatorPool è un ConcreteCreator e implementa un Object Pool
public class CreatorPool extends ShapeCreator {
    private LinkedList<Shape> pool = new LinkedList<Shape>();
    // getShape() è un metodo factory che ritorna un oggetto prelevato dal pool
    public Shape getShape() {
        Shape s;
        if (pool.size() > 0) s = pool.remove();
        else s = new Circle();
        return s;
    }
    // releaseShape() inserisce un oggetto nel pool
    public void releaseShape(Shape s) {
        pool.add(s);
    }
}
```

E. Tramontana Pool, Dependency - 12 May 10 2

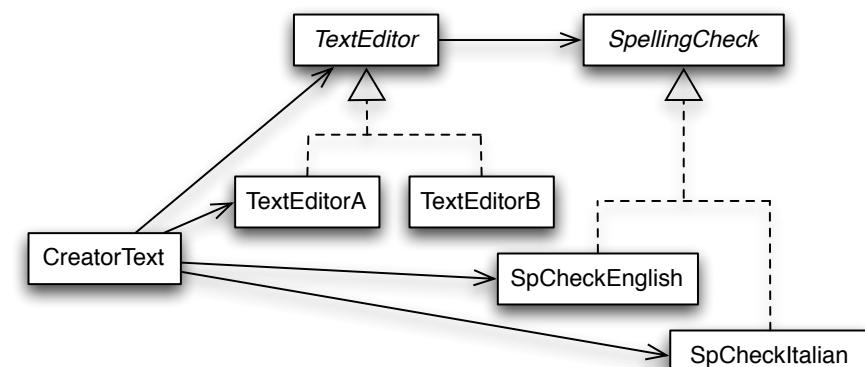
# Dependency Injection

- Il design pattern Factory Method può essere usato per inserire le dipendenze necessarie ad altri oggetti (istanze di ConcreteProduct)
- Dependency injection
  - Una classe C usa un servizio S (ovvero C *dipende* da S)
  - Esistono tante implementazioni di S (ovvero S1, S2), la classe C non deve dipendere dalle implementazioni S1, S2
  - Al momento di creare l'istanza di C, indico all'istanza di C con quale implementazione di S deve operare
- Esempio di dependency
  - Una classe TextEditor usa un servizio SpellingCheck
  - Ci sono tante classi che *implementano* il servizio SpellingCheck, in base alla lingua usata: SpCheckEnglish, SpCheckItalian, etc.
  - TextEditor deve poter essere collegato ad una delle classi che implementano SpellingCheck

E. Tramontana Pool, Dependency - 12 May 10 3

# Diagramma UML

- Esempio di Dependency Injection



E. Tramontana Pool, Dependency - 12 May 10 4

# Esempio Codice Dependency Injection

```
public class TextEditorA implements TextEditor { //TextEditorA è un ConcrProduct
    private SpellingCheck speller;
    public TextEditorA(SpellingCheck sp) { // inserisco la dipen. fornendo sp,
        speller = sp; // istanza del serv., al costruttore
    }
    public void put(String s) {
        if (speller.check(s)) ...
        else ...
    }
}

public class CreatorText { // CreatorText è un ConcreteCreator
    public static TextEditor getEnglishEditor() {
        return new TextEditorA(new SpCheckEnglish());
    }
    public static TextEditor getItalianEditor() {
        return new TextEditorB(new SpCheckItalian());
    }
}
```

E. Tramontana Pool, Dependency - 12 May 10 5

## Note sul codice

- Considerazioni

- TextEditorA e TextEditorB svolgono il ruolo di ConcreteProduct
- L'attributo speller di TextEditorA è inizializzato, attraverso la chiamata al costruttore, con l'opportuna istanza
- Il metodo factory getEnglishEditor() conosce la classe per lo spelling da usare ed inserisce (inject) la dipendenza sull'istanza di TextEditorA
- Un unico metodo factory istanzia e crea le dipendenze
  - Se usassi molteplici classi ConcreteCreator spargerei le decisioni di istanziazione
- All'interno della classe TextEditorA non è fissato l'uso di una classe che implementa SpellingCheck
- Posso sostituire la classe che implementa SpellingCheck facilmente
- Ho ottenuto la composizione di comportamenti piccoli e separati

E. Tramontana Pool, Dependency - 12 May 10 6

# Design Pattern Adapter

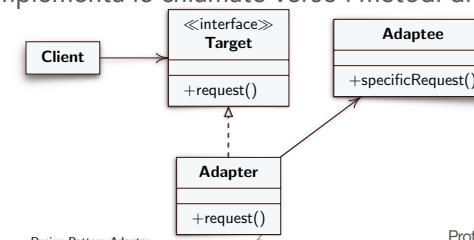
- Intento:** Converte l'interfaccia di una classe in un'altra interfaccia che i client si aspettano. Adapter permette ad alcune classi di interagire, eliminando il problema di interfacce incompatibili
- Problema**
  - Alcune volte una classe di una libreria non può essere usata poiché incompatibile con l'interfaccia che si aspetta l'applicazione. Ovvero nome metodo, parametri, tipo parametri di chiamate all'interno dell'applicazione non sono corrispondenti a quelli offerti da una classe di libreria
  - Non è possibile cambiare l'interfaccia della libreria, poiché non si ha il sorgente (comunque non conviene cambiarla)
  - Non è possibile cambiare l'applicazione, e si può voler cambiare quale metodo invocare, senza renderlo noto al chiamante

1

Prof. Tramontana - Marzo 2019

# Design Pattern Adapter

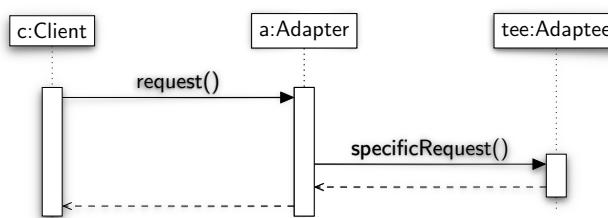
- Soluzione Object Adapter**
  - Target** è l'interfaccia che il chiamante si aspetta
  - Adaptee** è l'oggetto di libreria
  - Adapter** converte, ovvero adatta, la chiamata che fa una classe client all'interfaccia della classe di libreria. Il chiamante usa l'Adapter come se fosse l'oggetto di libreria. Adapter tiene il riferimento all'oggetto di libreria (Adaptee) e sa come invocarlo, ovvero implementa le chiamate verso i metodi di Adaptee



Prof. Tramontana - Marzo 2019

# Design Pattern Adapter

- Diagramma di sequenza della soluzione Object Adapter



3

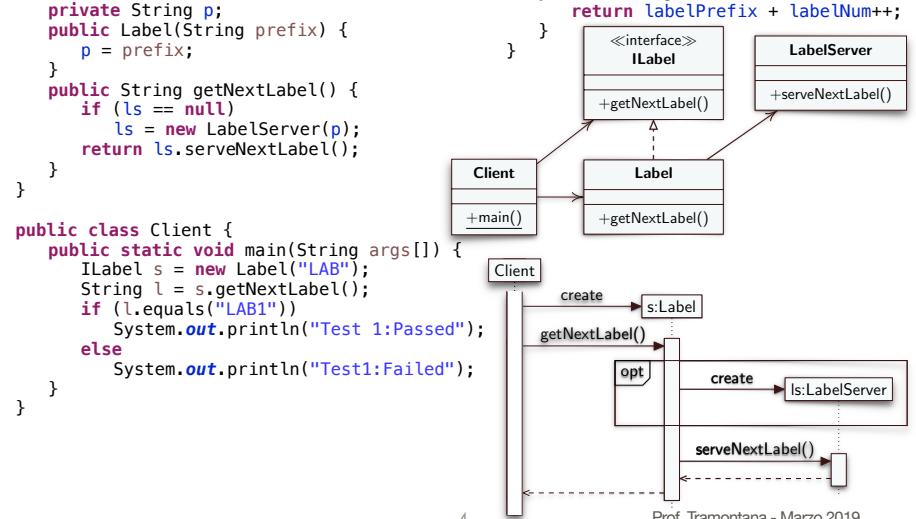
Prof. Tramontana - Marzo 2019

```

public interface ILabel { // Target
    public String getNextLabel();
}

// Adapter
public class Label implements ILabel {
    private LabelServer ls;
    private String p;
    public Label(String prefix) {
        p = prefix;
    }
    public String getNextLabel() {
        if (ls == null)
            ls = new LabelServer(p);
        return ls.serveNextLabel();
    }
}

public class Client {
    public static void main(String args[]) {
        ILabel s = new Label("LAB");
        String l = s.getNextLabel();
        if (l.equals("LAB1"))
            System.out.println("Test 1:Passed");
        else
            System.out.println("Test1:Failed");
    }
}
  
```



4

Prof. Tramontana - Marzo 2019

# Design Pattern Adapter

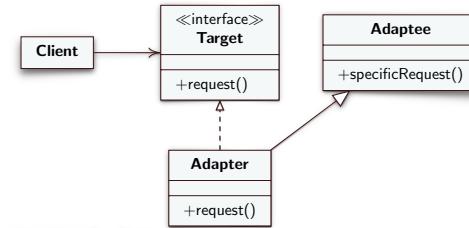
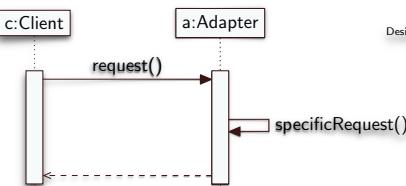
- Soluzione Class Adapter

- Adapter è sottoclasse di Adaptee

```
public class Label extends LabelServer implements ILabel { // Adapter

    public Label(String prefix) {
        super(prefix);
    }

    public String getNextLabel() {
        return serveNextLabel();
    }
}
```



5

Prof. Tramontana - Marzo 2019

# Design Pattern Adapter

- Variante Adapter a due vie

- Definizione: la classe Adapter fornisce l'interfaccia di Target e l'interfaccia di Adaptee. Realizzazione: la soluzione Class Adapter è un Adapter a due vie

- Conseguenze del design pattern Adapter

- Client e classe di libreria Adaptee rimangono indipendenti. L'Adapter può cambiare il comportamento dell'Adaptee
- Può aggiungere test di precondizioni e postcondizioni [Precondizioni: cosa si deve soddisfare prima di eseguire. Postcondizioni: cosa è verificato se tutto è andato bene]
- L'Object Adapter può implementare la tecnica di Lazy Initialization
- Il design pattern Adapter aggiunge un livello di indirezione. Ogni invocazione del client ne scatena un'altra fatta dall'Adapter. Possibile rallentamento (trascurabile), e codice più difficile da comprendere

6

Prof. Tramontana - Marzo 2019

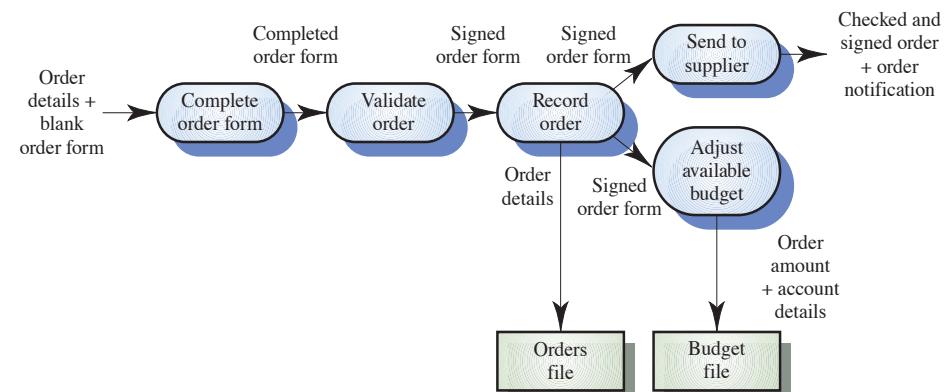
# Modelli di comportamento

- Sono usati per descrivere il comportamento globale del sistema
  - **Data processing model** (ovvero Data Flow Diagram, DFD)
    - Mostrano i passi per l'elaborazione di dati che attraversano il sistema
    - Notazione intuitiva comprensibile ai clienti
    - Mostrano lo scambio di informazioni tra sistemi e sottosistemi
    - Simili al diagramma delle attività UML
  - **State machine model** (in UML sono i diagrammi degli Stati)
    - Modellano il comportamento in risposta a eventi interni o esterni
    - Mostrano **stati** del sistema come **nodi** ed **eventi** come **archi** tra i nodi
    - Quando un evento si verifica, il sistema passa da uno stato a un altro
    - Utili per sistemi real-time, poiché spesso pilotati da eventi
- Entrambi sono richiesti per ottenere la descrizione del sistema

Riferimenti  
Pressman, capitoli 6.5.2, 8.5.2,  
8.5.3, 8.6, 8.8  
Sommerville, capitoli 7.1, 7.2

E. Tramontana - UML Attività e Stati - 14 Apr 10 1

# Esempio Modello DFD per ordini



- Rettangoli arrotondati: passi di elaborazione
- Frecce: flussi
- Rettangoli: archivi o sorgenti dati

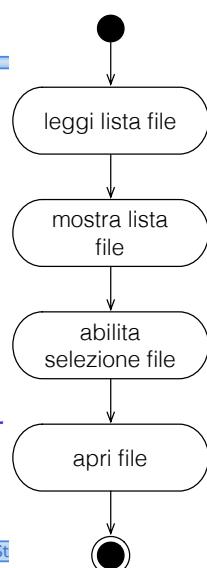
E. Tramontana - UML Attività e Stati - 14 Apr 10 2

# Diagramma delle attività

- Utilizzato per la modellazione di processi e workflow
- Mostra dettagli di funzionamento interno del sistema software da realizzare
- È una vista sull'esecuzione delle attività
- Mostra dipendenze tra attività (o passi)

Attività: Apri file da browser

E. Tramontana - UML Attività e Stati - 14 Apr 10 3

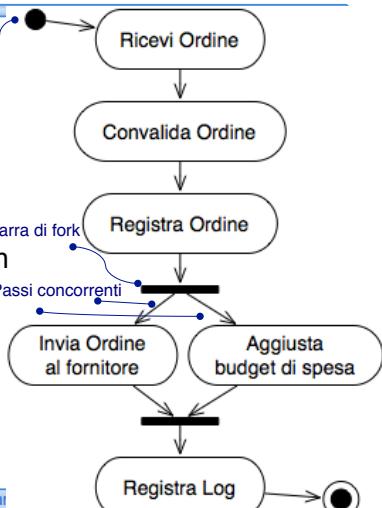


# Diagramma attività UML per ordini

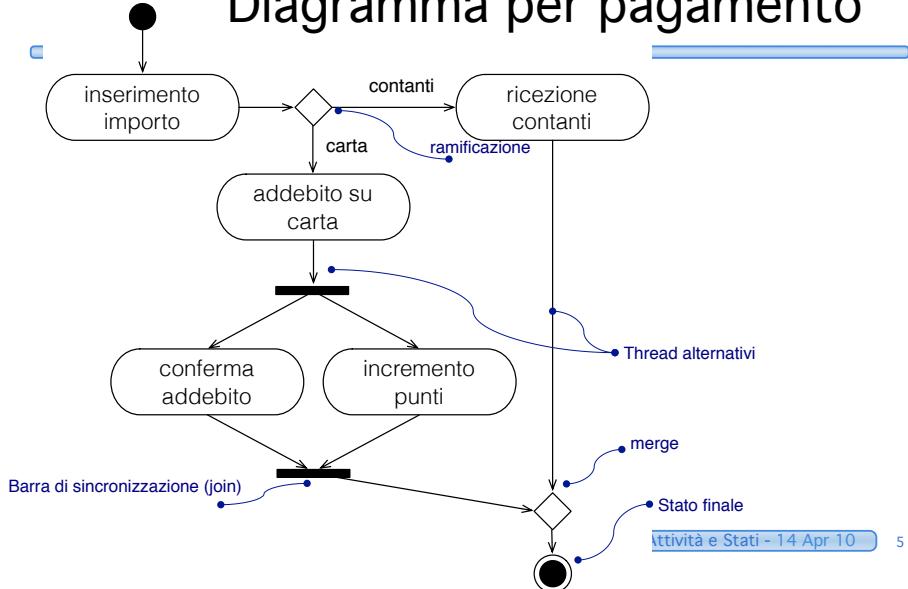
Notazione grafica per diagrammi UML delle attività

- Rettangoli arrotondati: passi di elaborazione
- Frecce continue: flussi
- Barre: sincronizzazione per fork o per join
- Rettangoli: oggetti o dati in input o in output
- Rombi: ramificazioni condizionali o merge
- Cerchi pieni: stati iniziale e finale (con circonferenza)

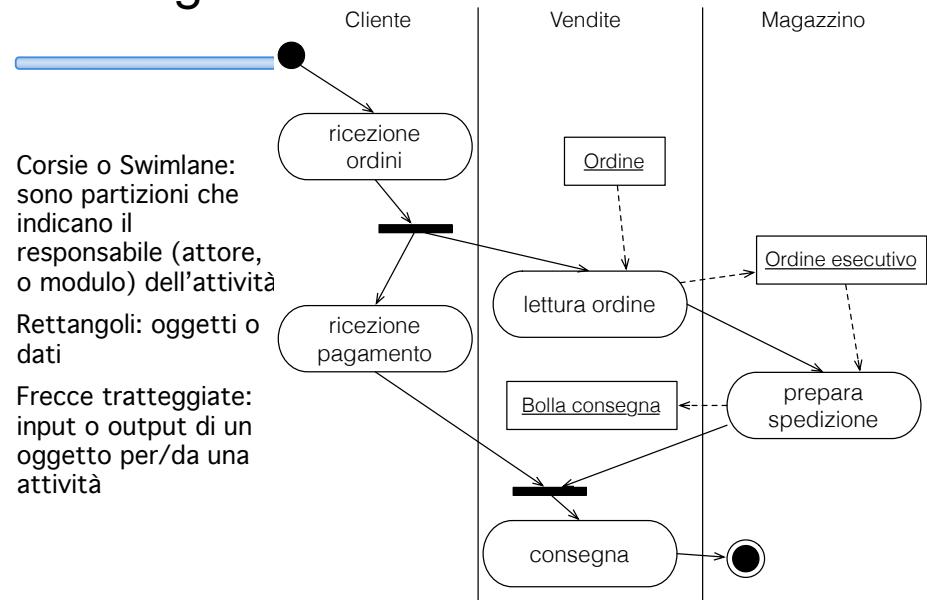
E. Tramontana - UML Attività e Stati - 14 Apr 10 4



## Diagramma per pagamento



## Diagramma attività con Corsie

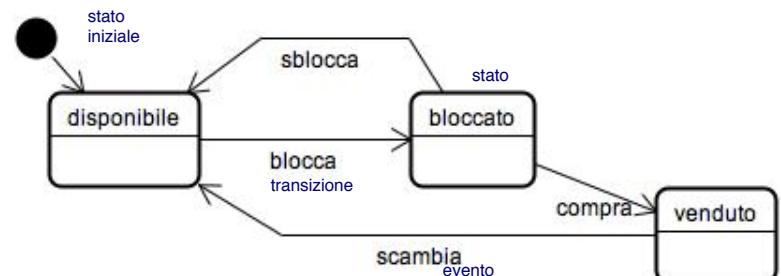


## Diagrammi attività

- I diagrammi delle attività
  - Non dicono quali sono gli oggetti che svolgono le attività
  - Sono il punto iniziale della progettazione
  - Vanno ri-elaborati per arrivare ad assegnare una o più operazioni ad una classe che le implementa
  - Possono essere usati come punto di partenza per ottenere i diagrammi UML di collaborazione fra oggetti

## Diagramma UML degli Stati

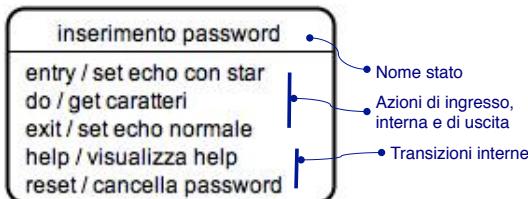
- Diagramma UML degli stati che mostra la vita di un biglietto per uno spettacolo



- Rettangoli con angoli arrotondati: stati
- Frecce: transizioni tra stati
- Cerchi pieni: stati iniziale e finale (con circonferenza)

## Stato

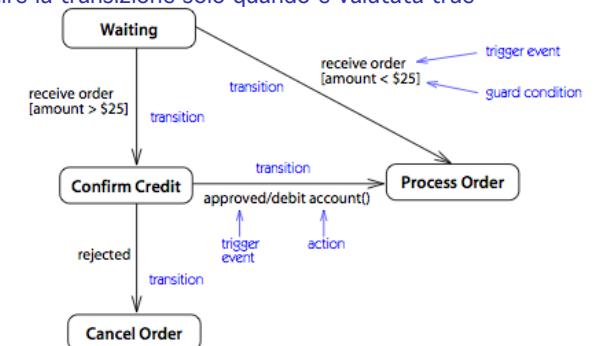
- Descrive un intervallo di tempo durante la vita di un oggetto
- È caratterizzato da
  - Valori di oggetti, o
  - Intervallo in cui un oggetto aspetta certi eventi, o
  - Intervallo in cui un oggetto fa certe azioni



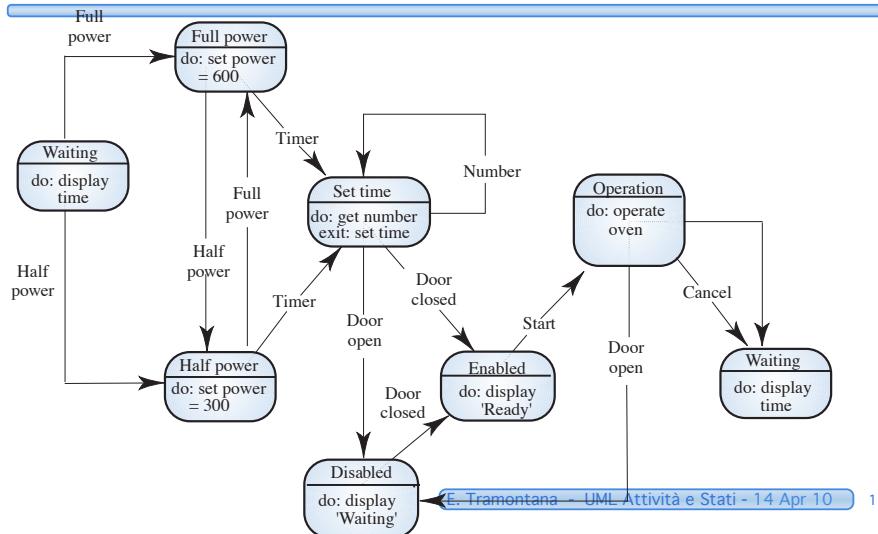
E. Tramontana - UML Attività e Stati - 14 Apr 10 9

## Transizione

- Permette di lasciare uno stato in risposta ad un certo evento
- Un oggetto gestisce un solo evento alla volta
- È caratterizzata da: event-trigger [guard] / action
  - Evento di inizio (trigger)
  - Condizione di guardia (espressione boolean), valutata quando l'evento avviene e che fa avvenire la transizione solo quando è valutata true
  - una azione
  - uno stato target



## Es. diagramma degli Stati per forno



11

## Descrizione stati per forno

| State      | Description                                                                                                                                                                                                                  |
|------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Waiting    | The oven is waiting for input. The display shows the current time.                                                                                                                                                           |
| Half power | The oven power is set to 300 watts. The display shows 'Half power'.                                                                                                                                                          |
| Full power | The oven power is set to 600 watts. The display shows 'Full power'.                                                                                                                                                          |
| Set time   | The cooking time is set to the user's input value. The display shows the cooking time selected and is updated as the time is set.                                                                                            |
| Disabled   | Oven operation is disabled for safety. Interior oven light is on. Display shows 'Not ready'.                                                                                                                                 |
| Enabled    | Oven operation is enabled. Interior oven light is off. Display shows 'Ready to cook'.                                                                                                                                        |
| Operation  | Oven in operation. Interior oven light is on. Display shows the timer countdown. On completion of cooking, the buzzer is sounded for 5 seconds. Oven light is on. Display shows 'Cooking complete' while buzzer is sounding. |

E. Tramontana - UML Attività e Stati - 14 Apr 10 12

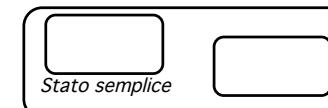
# Descrizione stimoli (eventi) per forno

| Stimulus    | Description                                   |
|-------------|-----------------------------------------------|
| Half power  | The user has pressed the half power button    |
| Full power  | The user has pressed the full power button    |
| Timer       | The user has pressed one of the timer buttons |
| Number      | The user has pressed a numeric key            |
| Door open   | The oven door switch is not closed            |
| Door closed | The oven door switch is closed                |
| Start       | The user has pressed the start button         |
| Cancel      | The user has pressed the cancel button        |

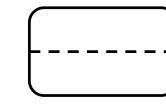
E. Tramontana - UML Attività e Stati - 14 Apr 10 13

# Stati composti

- Uno stato composto è uno stato che consiste di vari sottostati sequenziali o concorrenti
  - Uno stato semplice non consiste di sottostati
- Solo uno dei sottostati sequenziali può essere attivo in un certo momento
- Lo stato esterno rappresenta la condizione di essere in uno qualsiasi degli stati interni
- Una transizione verso o da uno stato composto può invocare varie azioni di entry (dalla più esterna) o exit (dalla più interna)



Stato composto sequenziale

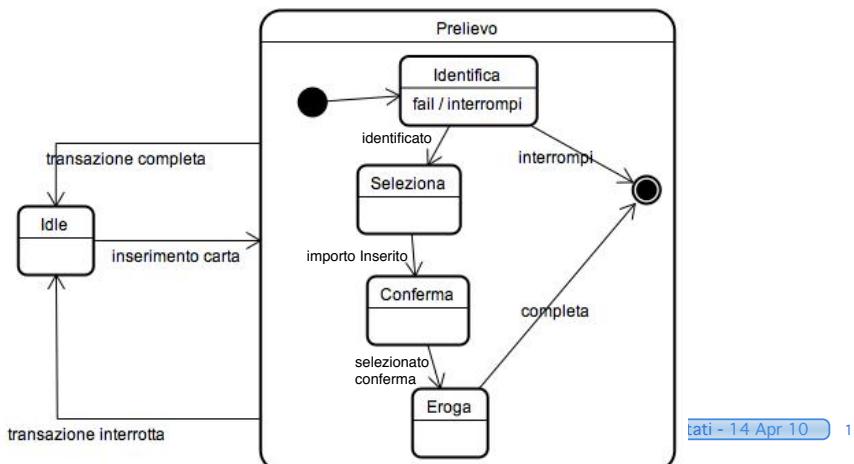


Stato composto concorrente

E. Tramontana - UML Attività e Stati - 14 Apr 10 14

# Diagramma UML degli Stati per ATM

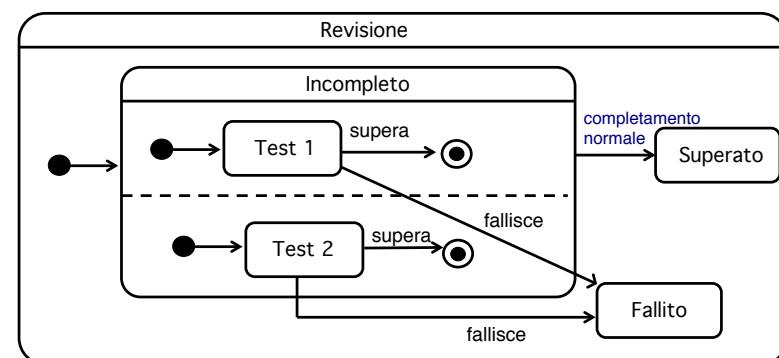
- Stato composto con sottostati sequenziali



E. Tramontana - UML Attività e Stati - 14 Apr 10 15

# Diagramma degli Stati per Revisioni

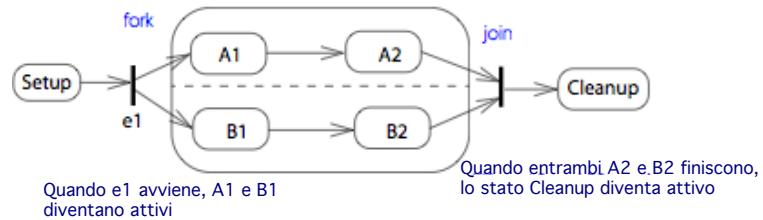
- Revisione è uno stato composto con sottostati sequenziali
  - I sottostati di Revisione sono: Incompleto, Superato e Fallito
- Incompleto è uno stato composto con sottostati concorrenti
  - I sottostati di Incompleto sono: Test 1 e Test 2



E. Tramontana - UML Attività e Stati - 14 Apr 10 16

# Diagramma degli Stati con fork e join

- Stato composto concorrente



# Modelli ad oggetti

- Descrivono il sistema in termini di classi (OOP)
- Una classe ha attributi ed operazioni comuni ad un set di oggetti
- Vari modelli (e diagrammi) ad oggetti possono essere prodotti
  - Di ereditarietà, aggregazione, interazione
- Pro del modello ad oggetti
  - Mappa naturalmente entità del mondo reale
  - Classi che rappresentano entità del dominio sono *riusabili*
- Contro
  - Entità astratte sono più difficilmente modellabili
  - L'identificazione di classi è un processo difficile che richiede una comprensione profonda del dominio applicativo

E. Tramontana - Diagrammi Classi - 26-Apr-15 1

# Identificazione classi

- Dall'elenco dei requisiti
  - Analisi grammaticale del testo
    - Nomi --> classi o attributi
    - Verbi --> operazioni
  - Individuare oggetti fisici
    - Questi suggeriscono classi corrispondenti
  - Raggruppare in modo coeso operazioni tra loro e dati tra loro
    - Questi gruppi suggeriranno delle classi

E. Tramontana - Diagrammi Classi - 26-Apr-15 2

## Esempio: ‘Gestione Ordini’

- Requisiti (frammenti)
  - ... dovrà essere possibile cercare un cliente ed avere mostrati i dati anagrafici del cliente trovato
  - ... la scheda cliente dovrà mostrare tutti i dati anagrafici ed un elenco di fornitori da cui il cliente ha già acquistato
  - ... su richiesta dell'utente dovrà essere calcolato l'importo complessivo degli ordini fatti dal cliente nell'intervallo di tempo selezionato
  - ... per ciascun ordine dovranno essere mostrati: nome fornitore, nome cliente, linea di appartenenza dei prodotti acquistati, importo complessivo
  - ... il report mensile dovrà contenere per ciascun cliente: la provincia di appartenenza e il totale ordinato per ciascun fornitore

E. Tramontana - Diagrammi Classi - 26-Apr-15 3

## Esempio: ‘Gestione Ordini’

- Requisiti
  - ... dovrà essere possibile **cercare** un cliente ed avere **mostrati** i dati anagrafici del **cliente** trovato
  - ... la **scheda cliente** dovrà **mostrare** tutti i **dati anagrafici** ed un elenco di fornitori da cui il **cliente** ha già acquistato
  - ... su richiesta dell'utente dovrà essere **calcolato** l'importo **complessivo** degli **ordini** fatti dal **cliente** nell'intervallo di **tempo** **selezionato**
  - ... per ciascun **ordine** dovranno essere **mostrati**: nome **fornitore**, nome **cliente**, linea di appartenenza **dei prodotti** acquistati, **importo complessivo**
  - ... il **report mensile** dovrà contenere per ciascun **cliente**: la **provincia di appartenenza** e il **totale ordinato** per ciascun **fornitore**

E. Tramontana - Diagrammi Classi - 26-Apr-15 4

# Identificazione classi

- Classi (in verde)
  - Cliente, Fornitore, Ordine, Prodotto, ReportMensile, SchedaCliente
- Attributi (in marrone)
  - Dati anagrafici cliente, nome cliente, provincia cliente
  - Linea appartenenza prodotti
  - Importo ordine
  - Nome fornitore
- Metodi (in arancio)
  - Cercare un cliente
  - Mostrare dati anagrafici e fornitori per un cliente
  - Calcolare totale ordini per un cliente
  - Selezionare ordini in un intervallo temporale
  - Calcolare totale ordini per un cliente per ciascun fornitore per mese

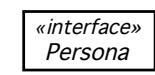
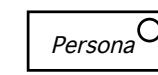
E. Tramontana - Diagrammi Classi - 26-Apr-15 5

# Notazione UML per classi e interfacce

- Esistono varie notazioni per la classe e l'interfaccia (Java)
- Le notazioni indicano
  - Nome classe; nome classe e attributi; nome classe, attributi e metodi
  - Per la visibilità di attributi e metodi: + public, # protected, – private
  - I nomi delle interfacce sono in corsivo
- Uno stereotipo (es. «interface») indica una variazione di un elemento UML, che ha tutte le proprietà dell'elemento di partenza

## Interfaccia

```
public interface Persona {  
    public String getName();  
    public boolean isEmpty();  
}
```



E. Tramontana - Diagrammi Classi - 26-Apr-15 6

# Notazione UML per classi e interfacce

- Forma degli attributi
  - visibilità nome: tipo es. – prezzo : int
  - visibilità nome: tipo = valore iniziale es. – prezzo : int = 10
  - visib nome[molteplicità]: tipo es. – prezzo[5] : int
- Forma dei metodi
  - visibilità nome(par: tipo): tipo di ritorno es. + getCosto() : int
  - I metodi statici sono sottolineati

## Classe

```
public class Prodotto {  
    private String nome;  
    private int costo;  
    public int getCosto() {  
        return costo;  
    }  
}
```



E. Tramontana - Diagrammi Classi - 26-Apr-15 7

# Diagramma UML delle classi

- Il diagramma delle classi mostra le classi, le loro caratteristiche e le loro relazioni (ereditarietà, implementazione, associazione, uso)
- Una associazione descrive una connessione tra istanze delle classi e specifica
  - Molteplicità: quante istanze di una classe possono essere in relazione con una istanza dell'altra classe (\* indica illimitate)
    - Se indicato come attributo, es. nome[0..1] : Persona
  - Nome ruolo: usato per attraversare l'associazione (è il nome dell'attributo all'interno della classe di partenza)
  - Navigabilità: verso di attraversamento
  - Nome: spiega l'associazione e il verso



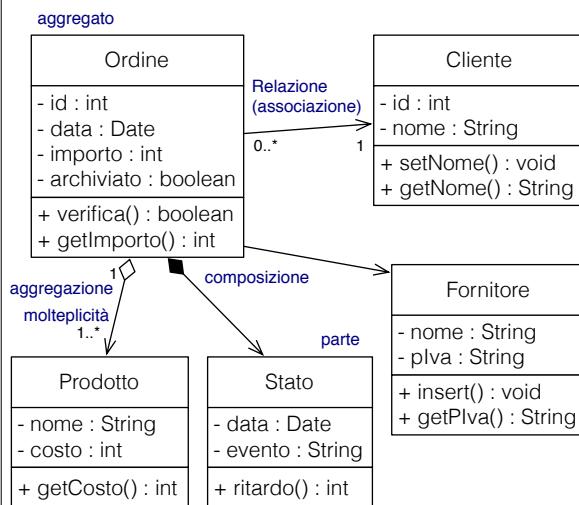
E. Tramontana - Diagrammi Classi - 26-Apr-15 8

## Diagramma delle classi per ‘Ordini’

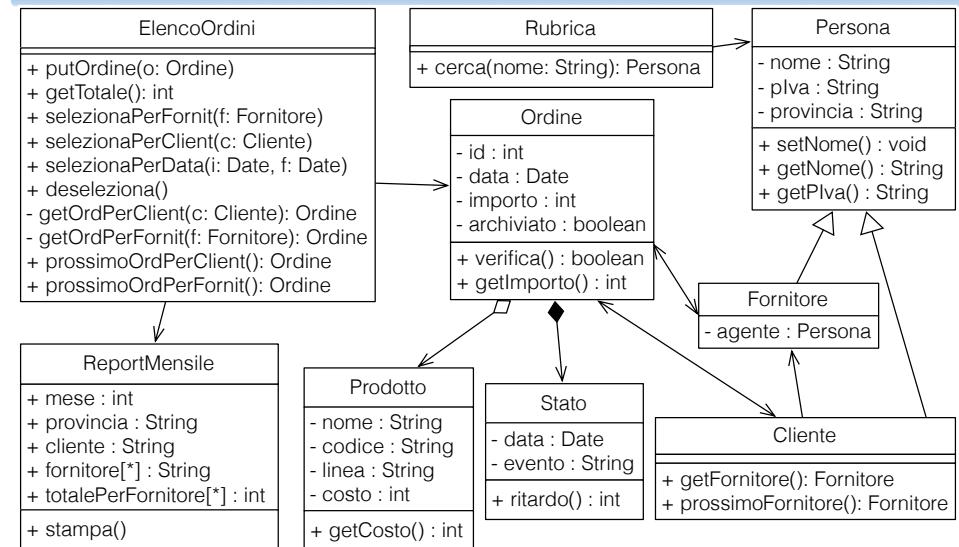
```
import java.util.Date;
public class Ordine {
    private int id;
    private Date data;
    private int importo;
    private boolean archiviato;

    private Cliente cli;
    private Fornitore forn;
    private Stato[] s;
    private Prodotto[] prod;

    public boolean verifica() {
        // implementazione
    }
    public int getImporto() {
        // implementazione
    }
}
```



## Diagramma delle classi (migliorato)



## Considerazioni su requisiti e progettazione

- L’analisi grammaticale sui requisiti non ha evidenziato le classi Persona, Rubrica, ElencoOrdini, e Stato
  - Il progettista deve capire ciò che occorre
- ‘Cerca un cliente’ è realizzato dal metodo `cerca()` di Rubrica
  - Rubrica può contenere istanze di Fornitore o Cliente
  - Persona può essere una interfaccia
- ‘L’elenco dei fornitori di un cliente’ è realizzato dai metodi `getFornitore()` e `prossimoFornitore()` di Cliente
  - La classe cliente tiene una lista di fornitori
- ‘Calcolare totale ordini’ è nel metodo `getTotale()` d’ElencoOrdini
  - L’insieme degli ordini su cui calcolare il totale è generato dai metodi `selezionaPerFornit()`, `selezionaPerClient()`, `selezionaPerData()`

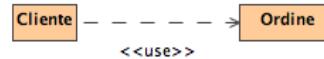
## Considerazioni

- Modularità
  - I metodi `selezionaPerFornit()`, `selezionaPerClient()`, `selezionaPerData()` permettono ciascuno di estrarre una parte degli ordini secondo criteri diversi, quindi si possono usare separatamente, e sono metodi brevi
- Generalizzazione
  - Si possono richiamare i suddetti metodi separatamente per ottenere selezioni di ordini (e totali) differenti rispetto al requisito
  - `prossimoXyz()` permette di scorrere la lista correntemente selezionata dall’esterno della classe ElencoOrdini [vedi classi Java `LinkedList`,  `StringTokenizer`, etc.]
- Miglioramenti futuri [vedi lezioni successive]
  - Usare design pattern *Factory Method* per Cliente e Fornitore
  - Usare design pattern *Composite* per Ordine e Prodotto
  - Usare design pattern *Observer* per ReportMensile e ElencoOrdini

# Costrutti di estensibilità

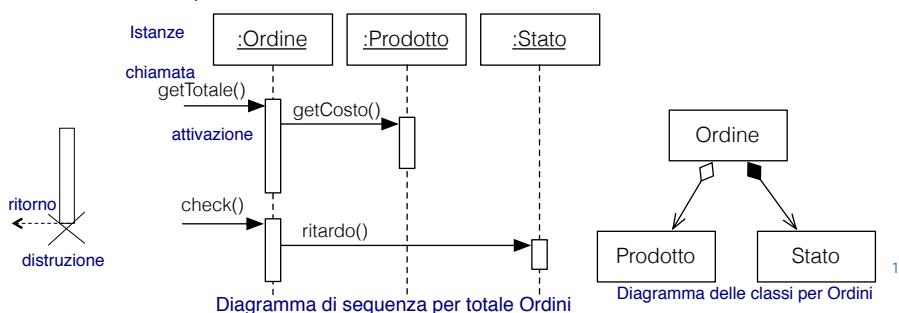
- Vincoli (Constraints)
  - Si usano per indicare condizioni o restrizioni e sono rappresentati da espressioni entro parentesi graffe
  - Es. accanto ad un attributo: {il valore è multiplo di 10}
- Stereotipi
  - Si usano per definire nuovi elementi o per specificare tipi di relazioni sono rappresentati da testo entro «»
- Dipendenze e stereotipi predefiniti
  - Associazioni (o dipendenze) tra classi: «use», «call», «instantiate», «destroy»
  - «use» indica che un elemento (Ordine) è richiesto per il corretto funzionamento di un altro (Cliente)
    - Es. necessario a compile time perché è un parametro di un metodo
  - Generalizzazioni tra classi: «implements»
- Tagged Value
  - Coppia di stringhe che indica un dato

E. Tramontana - Diagrammi Classi - 26-Apr-15 13  
ed il suo valore entro {} Es. dentro una classe: {nome=John}



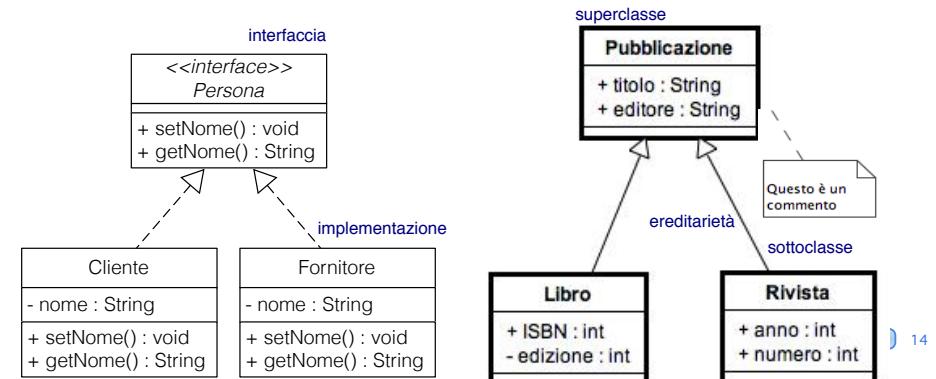
# Diagramma UML di sequenza

- Mostra interazioni tra oggetti
  - L'asse temporale è inteso in verticale
  - In alto in orizzontale ci sono i vari oggetti che ne prendono parte
    - In ciascuna colonna verticale, se l'oggetto che partecipa esiste è indicato con una linea tratteggiata, se è attivo con una barra (di attivazione)
  - Un messaggio è una freccia dalla barra di attivazione di un oggetto ad un altro
    - Frecce piene indicano comunicazione sincrona, viceversa vuote asincrona



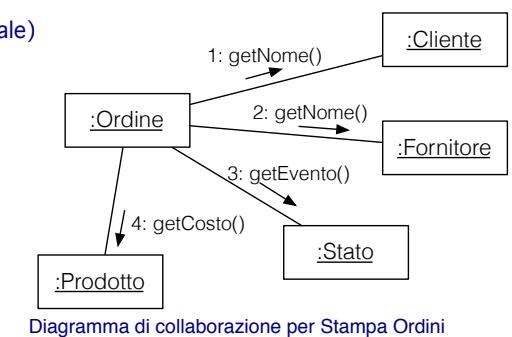
# Diagramma UML di ereditarietà

- Organizza le classi in una gerarchia
  - Le classi in alto nella gerarchia (superclassi) mostrano le proprietà comuni delle classi in basso (sottoclassi)
  - Le classi ereditano gli attributi e i servizi da una o più super-classi



# Diagramma UML di collaborazione

- Mostra interazioni tra oggetti
  - Il flusso dei messaggi è indicato da frecce accanto alle associazioni fra istanze che partecipano all'interazione
  - I messaggi sono mostrati da etichette sulle frecce ed hanno
    - Un numero sequenziale che indica l'ordine temporale con cui avvengono
    - Il metodo chiamato
    - Un valore di ritorno (opzionale)



# Design Pattern Facade

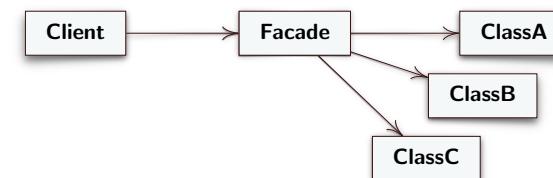
- **Intento:** Fornire un'interfaccia unificata al posto di un insieme di interfacce in un sottosistema (consistente di un insieme di classi). Definire un'interfaccia di alto livello (semplificata) che rende il sottosistema più facile da usare
- **Problema**
  - Spesso si hanno tante classi che svolgono funzioni correlate e l'insieme delle interfacce può essere complesso
  - Può essere difficile capire qual è l'interfaccia essenziale ai client per l'insieme di classi
  - Si vogliono ridurre le comunicazioni e le dipendenze dirette fra i client ed il sottosistema

1

Prof. Tramontana - Aprile 2019

# Design Pattern Facade

- **Soluzione**
  - **Facade** fornisce un'unica interfaccia semplificata ai client e nasconde gli oggetti del sottosistema, questo riduce la complessità dell'interfaccia e quindi delle chiamate. **Facade** invoca i metodi degli oggetti che nasconde
  - Client interagisce solo con l'oggetto Facade



2

Prof. Tramontana - Aprile 2019

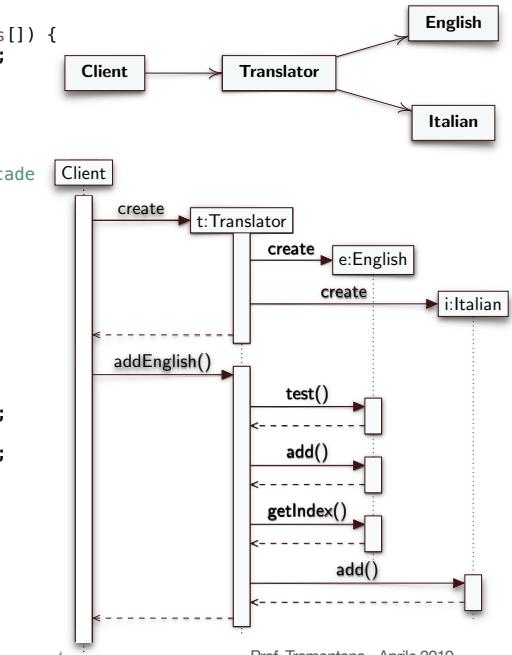
# Design Pattern Facade

- **Conseguenze**
  - Nasconde ai client l'implementazione del sottosistema
  - Promuove l'accoppiamento debole tra sottosistema e client
  - Riduce le dipendenze di compilazione in sistemi grandi. Se si cambia una classe del sottosistema, si può ricompilare la parte di sottosistema fino al facade, quindi non i vari client
  - Non previene l'uso di client più complessi, quando occorre, che accedono ad oggetti del sottosistema
- **Implementazione**
  - Per rendere gli oggetti del sottosistema non accessibili al client le corrispondenti classi possono essere annidate dentro la classe Facade

3

Prof. Tramontana - Aprile 2019

```
public class Client {  
    public static void main(String args[]) {  
        Translator t = new Translator();  
        t.addEnglish("Hello");  
        t.multiPrinting();  
    }  
  
    public class Translator { // Ruolo Facade  
        private English e = new English();  
        private Italian i = new Italian();  
  
        public void addEnglish(String s) {  
            if (e.test(s)) {  
                e.add(s);  
                i.add(e.getIndex(s));  
            }  
        }  
  
        public void multiPrinting() {  
            System.out.print("Italiano: ");  
            i.printText();  
            System.out.print("English: ");  
            e.printText();  
        }  
    }  
}
```



4

Prof. Tramontana - Aprile 2019

```
public class English {
    private String text = " ";
    private List<String> d =
        Arrays.asList("Alright", "Hello",
                     "Understood", "Yes");

    public boolean test(String s) {
        return d.contains(s);
    }

    public void add(String s) {
        text = text + " " + s;
    }

    public String getText() {
        return text;
    }

    public int getIndex(String s) {
        return d.indexOf(s);
    }

    public void printText() {
        System.out.println(text);
    }
}
```

```
public class Italian {
    private String text = " ";
    private List<String> d =
        Arrays.asList("Va bene", "Ciao",
                     "Capito", "Si");

    public void add(int i) {
        text = text + " " + d.get(i);
    }

    public void printText() {
        System.out.println(text);
    }
}
```

# Design Pattern State

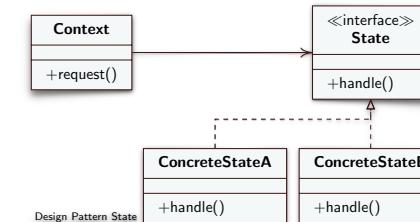
- **Intento:** Permettere ad un oggetto di alterare il suo comportamento quando il suo stato interno cambia. Far sembrare che l'oggetto abbia cambiato la sua classe
- **Problema**
  - Il comportamento di un oggetto dipende dal suo stato e il comportamento deve cambiare a run-time in base al suo stato
  - Le operazioni da svolgere hanno vari grandi rami condizionali che dipendono dallo stato
  - Lo stato è spesso rappresentato dal valore di una o più variabili enumerative costanti
  - Spesso varie operazioni contengono la stessa struttura condizionale

1

Prof. Tramontana - Aprile 2019

# Design Pattern State

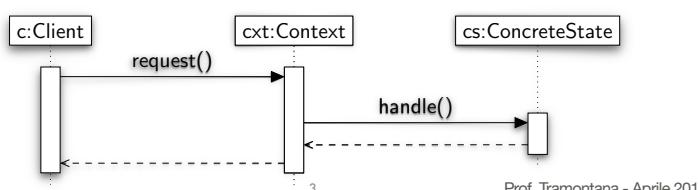
- **Soluzione**
  - Inserire ogni ramo condizionale in una classe separata
  - **Context** definisce l'interfaccia che interessa ai client, e mantiene un'istanza di una classe **ConcreteState** che definisce lo stato corrente
  - **State** definisce un'interfaccia che incapsula il comportamento associato ad un particolare stato del Context
  - **ConcreteState** sono le sottoclassi che implementano ciascuna il comportamento associato ad uno stato del Context



Prof. Tramontana - Aprile 2019

# Design Pattern State

- **Collaborazioni**
  - Il **Context** passa le richieste dipendenti da un certo stato all'oggetto **ConcreteState** corrente
  - Un **Context** può passare se stesso come argomento all'oggetto **ConcreteState** per farlo accedere al contesto se necessario
  - Il **Context** è l'interfaccia per le classi client
  - Il **Context** o i **ConcreteState** decidono quale stato è il successivo ed in quali circostanze



Prof. Tramontana - Aprile 2019

# Design Pattern State

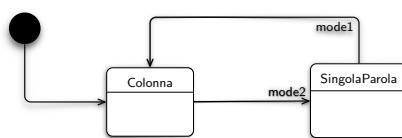
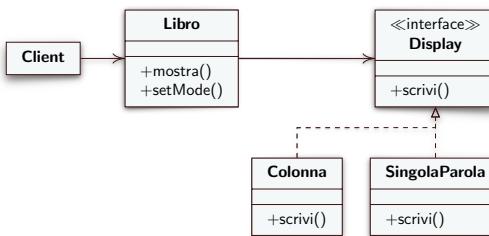
- **Conseguenze**
  - Il comportamento associato ad uno stato è localizzato in una sola classe (**ConcreteState**) e si partiziona il comportamento di stati differenti. Per tale motivo, si posso aggiungere nuovi stati e transizioni facilmente, creando nuove sottoclassi. Incapsulare le azioni di uno stato in una classe impone una struttura e rende più chiaro lo scopo del codice
  - La logica che gestisce il cambiamento di stato è separata dai vari comportamenti ed è in una sola classe (**Context**), anziché (con istruzioni if o switch) sulla classe che implementa i comportamenti. Tale separazione aiuta ad evitare stati inconsistenti, poiché i cambiamenti di stato vengono decisi da una sola classe e non da tante
  - Il numero di classi totale è maggiore, le classi sono più semplici

4

Prof. Tramontana - Aprile 2019

# Esempio

- Si vogliono avere vari modi per scrivere il testo di un libro su un display: in modalità una colonna, due colonne, o una singola parola per volta



Prof. Tramontana - Aprile 2019

```

public class Libro { // Context
    private String testo = "Darwin's _Origin of Species_ persuaded the world that the "
    + "difference between different species of animals and plants is not the fixed "
    + "immutable difference that it appears to be";
    private List<String> lista = Arrays.asList(testo.split("[\\s]+"));
    public Display mode = new Colonna();
    public void mostra() {
        mode.scrivi(lista);
    }
    public void setMode(int x) {
        switch (x) {
            case 1: mode = new Colonna(); break;
            case 2: mode = new SingolaParola(); break;
        }
    }
}

public interface Display { // State
    public void scrivi(List<String> testo);
}

public class Colonna implements Display { // ConcreteState
    private final int numCar = 38;
    private final int numRighe = 12;
    public void scrivi(List<String> testo) {
        int riga = 0;
        int col = 0;
        for (String p : testo) {
            if (col + p.length() > numCar) {
                System.out.println();
                riga++;
                col = 0;
            }
            if (riga == numRighe) break;
            System.out.print(p + " ");
            col += p.length() + 1;
        }
    }
}

public class Client {
    public static void main(String[] args) {
        Libro l = new Libro();
        l.mostra();
        l.setMode(2);
        l.mostra();
    }
}
  
```

6

Prof. Tramontana - Aprile 2019

```

public class SingolaParola implements Display { // ConcreteState
    private int maxLung;

    public void scrivi(List<String> testo) {
        System.out.println();
        mettiSpazi(30);
        trovaMaxLung(testo);
        for (String p : testo) {
            int numSpazi = (maxLung - p.length()) / 2;
            mettiSpazi(numSpazi);
            System.out.print(p);
            if (p.length() % 2 == 1) numSpazi++;
            mettiSpazi(numSpazi);
            aspetta();
            cancellaRiga();
        }
        System.out.println();
    }

    private void mettiSpazi(int n) {
        for (int i = 0; i < n; i++) System.out.print(" ");
    }

    private void cancellaRiga() {
        for (int i = 0; i < maxLung; i++) System.out.print("\b");
    }

    private void trovaMaxLung(List<String> testo) {
        for (String p : testo) if (maxLung < p.length()) maxLung = p.length();
    }

    private static void aspetta() {
        try {
            Thread.sleep(300);
        } catch (InterruptedException e) { }
    }
}
  
```

7

Prof. Tramontana - Aprile 2019

```

public class LibroPrimaDiState {
    private String testo = "...";
    private List<String> lista = Arrays.asList(testo.split("[\\s]+"));
    private int mode = 2;

    public void mostra() {
        switch (mode) {
            case 1:
                // vedi metodo scrivi della classe SingolaParola
                break;
            case 2:
                // vedi metodo scrivi della classe Colonna
                break;
        }
    }

    public void setMode(int x) {
        mode = x;
    }
}
  
```

8

Prof. Tramontana - Aprile 2019

# Requisiti

- Il sistema software dovrà fornire la possibilità di prenotare e acquistare un biglietto (per un viaggio). Si potrà annullare la prenotazione, ma non l'acquisto. Ogni biglietto ha un codice, un prezzo, una data di acquisto, il nome dell'intestatario (e i dettagli del viaggio). Per la prenotazione si dovrà dare il nome dell'intestatario.

# Progettazione

- Il sistema software dovrà fornire la possibilità di **prenotare** e **acquistare** un **biglietto** (per un viaggio). Si potrà **annullare** la prenotazione, ma non l'acquisto. Ogni biglietto ha un **codice**, un **prezzo**, una **data** di acquisto, il **nome** dell'intestatario (e i dettagli del viaggio). Per la prenotazione si dovrà dare il nome dell'intestatario.
- Classi:** Biglietto
- Attributi:** codice, prezzo, data, nome
- Operazioni:** prenota, acquista, annulla
- La classe Biglietto si può trovare in uno degli stati: disponibile, bloccato (ovvero prenotato), venduto

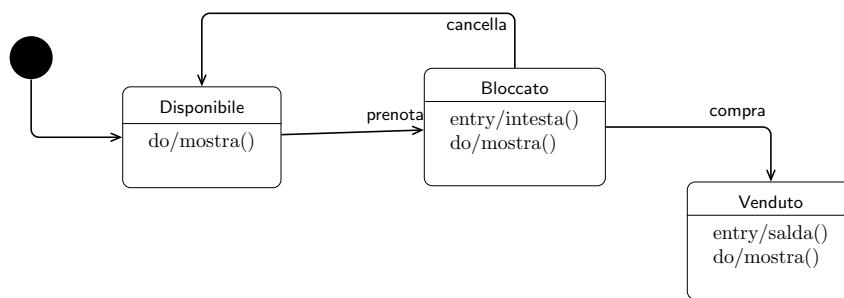
1

Prof. Tramontana - Aprile 2019

2

Prof. Tramontana - Aprile 2019

# Diagramma Degli Stati



- Quindi, la classe Biglietto dovrà implementare i metodi: prenota, cancella, compra, mostra.
- Ciascuna operazione controllerà lo stato in cui si trova il biglietto prima di eseguire le azioni necessarie

3

Prof. Tramontana - Aprile 2019

```
// Codice che implementa i suddetti requisiti (prima versione)

public class Biglietto {
    private String codice = "XYZ", nome;
    private int prezzo = 100;
    private enum StatoBiglietto { DISP, BLOC, VEND }
    private StatoBiglietto stato = StatoBiglietto.DISP;

    // ogni operazione deve controllare in che stato si trova il biglietto
    public void prenota(String s) {
        switch (stato) {
            case DISP:
                System.out.println("Cambia stato da Disponibile a Bloccato");
                nome = s;
                System.out.println("Inserito nuovo intestatario");
                stato = StatoBiglietto.BLOC;
                break;
            case BLOC:
                nome = s;
                System.out.println("Inserito nuovo intestatario");
                break;
            case VEND:
                System.out.println("Non puo' cambiare il nome nello stato Venduto");
                break;
        }
    }

    public void cancella() {
        if (stato == StatoBiglietto.BLOC) {
            System.out.println("Cambia stato da Bloccato a Disponibile");
            nome = null;
            System.out.println("Biglietto annullato");
            stato = StatoBiglietto.DISP;
        }
    }

    public void compra() {
        if (stato == StatoBiglietto.DISP) {
            System.out.println("Cambia stato da Disponibile a Bloccato");
            nome = null;
            System.out.println("Inserito nuovo intestatario");
            stato = StatoBiglietto.BLOC;
        }
    }

    public void mostra() {
        System.out.println("Nome: " + nome);
        System.out.println("Prezzo: " + prezzo);
        System.out.println("Stato: " + stato.name());
    }
}
```

4

Prof. Tramontana - Aprile 2019

```

public void cancella() {
    switch (stato) {
        case DISP:
            System.out.println("Lo stato era gia' Disponibile");
            break;
        case BLOC:
            System.out.println("Cambia stato da Bloccato a Disponibile");
            nome = "";
            stato = StatoBiglietto.DISP;
            break;
        case VEND:
            System.out.println("Non puo' cambiare stato da Venduto a Disponibile");
            break;
    }
}

public void mostra() {
    System.out.println("Prezzo: " + prezzo + " codice: " + codice);
    if (stato == StatoBiglietto.BLOC || stato == StatoBiglietto.VEND)
        System.out.println("Nome: " + nome);
}

```

5

Prof. Tramontana - Aprile 2019

```

public void compra() {
    switch (stato) {
        case DISP:
            System.out.println("Non si puo' pagare, bisogna prima intestarlo");
            break;
        case BLOC:
            System.out.println("Cambia stato da Bloccato a Venduto");
            stato = StatoBiglietto.VEND;
            System.out.println("Pagamento effettuato");
            break;
        case VEND:
            System.out.println("Il biglietto era gia' stato venduto");
            break;
    }
}

```

6

Prof. Tramontana - Aprile 2019

```

public class Client {
    private Biglietto b = new Biglietto();
    public static void main(String[] args) {
        usaBiglietto();
    }

    private static void usaBiglietto() {
        b.prenota("Mario Tokoro");
        b.mostra();
        b.compra();
        b.mostra();
    }

    private static void nonUsaOk() {
        b.compra();
        b.cancella();
        b.prenota("Mario Biondi");
    }
}

Output dell'esecuzione di MainBiglietto
Prezzo: 100 codice: XYZ
Cambia stato da Disponibile a Bloccato
Inserito nuovo intestatario
Prezzo: 100 codice: XYZ
Nome: Mario Tokoro
Cambia stato da Bloccato a Venduto
Pagamento effettuato
Prezzo: 100 codice: XYZ
Nome: Mario Tokoro
Il biglietto era gia' stato venduto
Non puo' cambiare stato da Venduto a Disponibile
Non puo' cambiare il nome nello stato Venduto

```

7

Prof. Tramontana - Aprile 2019

## Analisi Del Codice

- La classe ha circa 70LOC, metodo più lungo 15LOC, solo 32 linee con ";"
- Ogni metodo ha vari rami condizionali, uno per ogni stato. La logica condizionale rende il codice difficile da modificare
- Il comportamento in ciascuno stato non è ben separato, poiché lo stesso metodo implementa più comportamenti
- Si può arrivare a un design e un codice più semplice, e che separa i comportamenti? Sì, tramite indirettezze
- Le condizioni possono essere trasformate in messaggi, questo riduce i duplicati, aggiunge chiarezza e aumenta la flessibilità del codice
- La tecnica di refactoring [Replace Conditional with Polymorphism](#) (ovvero Sostituisci i rami condizionali con il polimorfismo), indica come fare
- Ovvero, si tratta del design pattern ... State, ovvero [Replace Type Code with State](#)

8

Prof. Tramontana - Aprile 2019

```

// StatoBiglietto e' uno State
public interface StatoBiglietto {
    public void mostra();
    public StatoBiglietto intesta(String s);
    public StatoBiglietto paga();
    public StatoBiglietto cancella();
}

// Disponibile e' un ConcreteState
public class Disponibile implements StatoBiglietto {
    @Override public void mostra() { }

    @Override public StatoBiglietto intesta(String s) {
        System.out.println("DISP Cambia stato da Disponibile a Bloccato");
        return new Bloccato().intesta(s);
    }

    @Override public StatoBiglietto paga() {
        System.out.println("DISP Non si puo' pagare, bisogna prima intestarlo");
        return this;
    }

    @Override public StatoBiglietto cancella() {
        System.out.println("DISP Lo stato era gia' Disponibile");
        return this;
    }
}

```

9

Prof. Tramontana - Aprile 2019

```

// Bloccato e' un ConcreteState
public class Bloccato implements StatoBiglietto {
    private String nome;

    @Override public void mostra() {
        System.out.println("BLOC Nome: " + nome);
    }

    @Override public StatoBiglietto intesta(String s) {
        System.out.println("BLOC Inserito nuovo intestatario");
        nome = s;
        return this;
    }

    @Override public StatoBiglietto paga() {
        System.out.println("BLOC Cambia stato da Bloccato a Venduto");
        return new Venduto(nome).paga();
    }

    @Override public StatoBiglietto cancella() {
        System.out.println("BLOC Cambia stato da Bloccato a Disponibile");
        return new Disponibile();
    }
}

```

10

Prof. Tramontana - Aprile 2019

```

import java.time.LocalDateTime;
public class Venduto implements StatoBiglietto { // Venduto e' un ConcreteState
    private final String nome;
    private LocalDateTime dataPagam;

    public Venduto(String n) { nome = n; }

    @Override public void mostra() {
        System.out.println("VEND Nome: " + nome);
    }

    @Override public StatoBiglietto intesta(String s) {
        System.out.println("VEND Non puo' cambiare il nome nello stato Venduto");
        return this;
    }

    @Override public StatoBiglietto paga() {
        if (dataPagam == null) {
            dataPagam = LocalDateTime.now();
            System.out.println("VEND Pagamento effettuato");
        } else
            System.out.println("VEND Il biglietto era gia' stato pagato");
        return this;
    }

    @Override public StatoBiglietto cancella() {
        System.out.println("VEND Non puo' cambiare stato da Venduto a Disponibile");
        return this;
    }
}

```

11

Prof. Tramontana - Aprile 2019

```

// Biglietto e' un Context
public class Biglietto {
    private String codice = "XYZ";
    private int prezzo = 100;

    private StatoBiglietto sb = new Disponibile();

    public void mostra() {
        System.out.println("Prezzo: " + prezzo + " codice: " + codice);
        sb.mostra();
    }

    public void prenota(String s) {
        sb = sb.intesta(s);
    }

    public void cancella() {
        sb = sb.cancella();
    }

    public void compra() {
        sb = sb.paga();
    }
}

```

12

Prof. Tramontana - Aprile 2019

```

public class Client {
    private static Biglietto b = new Biglietto();
    public static void main(String[] args) {
        usaBiglietto();
    }

    private static void usaBiglietto() {
        b.prenota("Mario Tokoro");
        b.mostra();
        b.compra();
        b.mostra();
    }

    private static void nonUsaOk() {
        b.compra();
        b.cancella();
        b.prenota("Mario Biondi");
    }
}

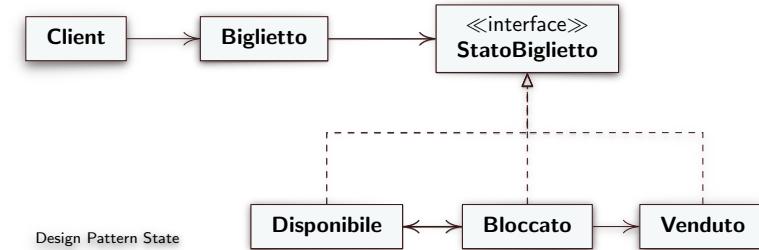
Output dell'esecuzione di MainBiglietto
Prezzo: 100 codice: XYZ
DISP Cambia stato da Disponibile a Bloccato
BLOC Inserito nuovo intestatario
Prezzo: 100 codice: XYZ
BLOC Nome: Mario Tokoro
BLOC Cambia stato da Bloccato a Venduto
VEND Pagamento effettuato
Prezzo: 100 codice: XYZ
VEND Nome: Mario Tokoro

VEND Il biglietto era già stato venduto
VEND Non puo' cambiare stato da Venduto a Disponibile
VEND Non puo' cambiare il nome nello stato Venduto

```

13

Prof. Tramontana - Aprile 2019



14

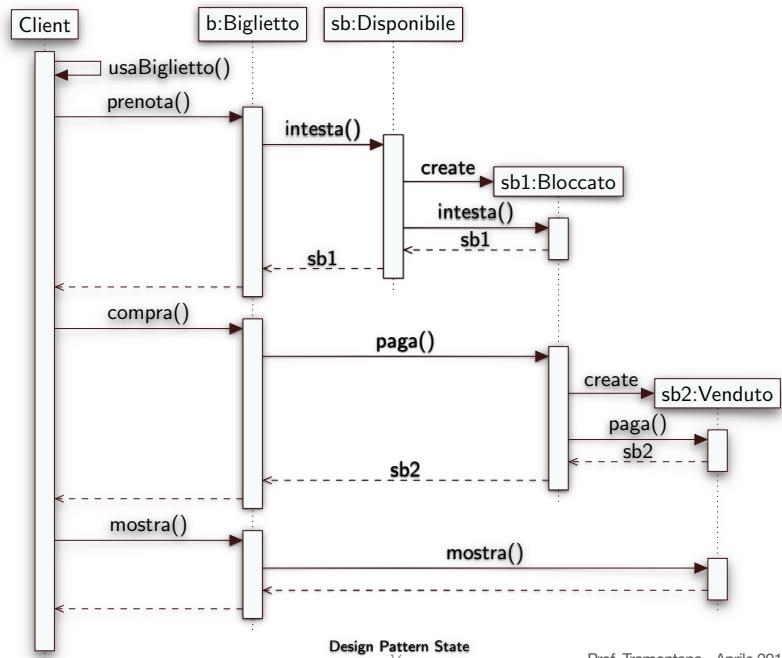
Prof. Tramontana - Aprile 2019

## Conseguenze

- Sono state eliminate le istruzioni condizionali: i metodi non devono controllare in quale stato si trovano, poiché la classe si riferisce ad un singolo stato. Non si ha codice duplicato per i test condizionali su ciascun metodo
- Ogni metodo è più semplice da comprendere e modificare
- Ciascuno stato avvia, quando occorre, una transizione, questo ha permesso di eliminare l'avvio delle transizioni da Context, e quindi gli switch su esso
- L'interfaccia usata dai ConcreteState permette di ritornare il riferimento a un nuovo state (è detta fluent)
- Il codice per ciascuno stato può implementare altre attività senza complicarsi
- La presenza di switch è un sintomo che suggerisce di usare il polimorfismo
- Le LOC sono 2 o 3 per metodo, ci sono 4 classi, e 1 interfaccia
- Totale LOC 140 circa (compresi commenti e linee vuote), solo 53 linee con ;"

15

Prof. Tramontana - Aprile 2019



16

Prof. Tramontana - Aprile 2019

# Observer

- **Intento:** Definire una dipendenza uno a molti fra oggetti, così che quando un oggetto cambia stato tutti i suoi oggetti dipendenti sono notificati e aggiornati automaticamente
- Conosciuto anche come **Publish-Subscribe**
- **Motivazioni**

- Un sistema che è stato partizionato in un insieme di classi che cooperano deve mantenere la **consistenza** fra oggetti che hanno relazioni
- Es. uno strumento di **presentazione** è separato dai **dati** dell'applicazione. Le classi che tengono i dati e quelle di presentazione sono separate e riusabili. Uno spreadsheet e un diagramma sono dipendenti dall'oggetto che contiene i dati e dovrebbero essere notificati dei cambiamenti dei dati

1

E. Tramontana — Maggio-2020

# Observer

- Il design pattern Observer descrive come stabilire relazioni fra oggetti dipendenti
- Gli oggetti chiave sono **Subject** e **Observer**
- Un **Subject** può avere tanti **Observer** che dipendono da esso e gli **Observer** sono notificati quando lo stato del **Subject** cambia
- **Applicabilità**
  - Un'astrazione ha due aspetti (es. dati e presentazione), ciascuno dipendente dall'altro. Incapsulare questi aspetti in oggetti separati permette di riusarli indipendentemente
  - Un cambiamento su un oggetto richiede il cambiamento di altri, non si conosce quanti oggetti sarà necessario cambiare
  - Un oggetto deve notificare altri oggetti senza fare assunzioni su chi sono tali oggetti, quindi gli oggetti non devono essere strettamente accoppiati

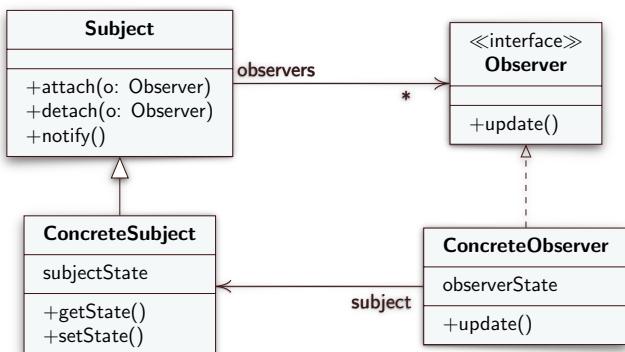
2

E. Tramontana — Maggio-2020

# Observer

## • Soluzione

- Diagramma delle classi



3

E. Tramontana — Maggio-2020

# Observer

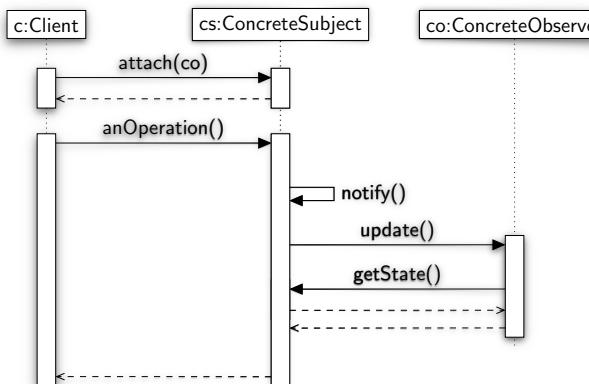
- **Partecipanti**
  - **Subject** conosce i suoi osservatori, un qualsiasi numero di oggetti **Observer** può osservare un **Subject**. Implementa le operazioni per aggiungere e togliere oggetti **Observer** e per notificarli
  - **Observer** definisce una interfaccia (`operazione update()`) comune a tutti gli oggetti che necessitano la notifica
  - **ConcreteSubject** tiene lo stato che interessa agli oggetti **ConcreteObserver**. Notifica i suoi osservatori quando il suo stato cambia. Eredita da **Subject**
  - **ConcreteObserver** tiene un riferimento all'oggetto **ConcreteSubject**, e tiene lo stato che deve rimanere consistente con quello del **Subject**. Implementa **Observer** per ricevere notifiche dei cambiamenti del **Subject**. Dopo la notifica può interrogare il **subject** per ottenere il nuovo dato

4

E. Tramontana — Maggio-2020

# Observer

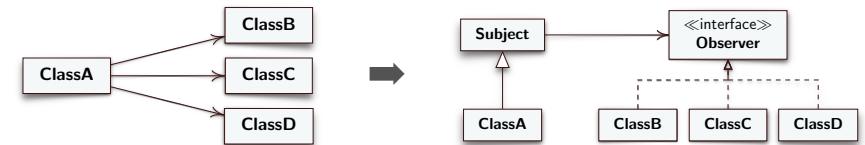
- Quando lo stato dell'oggetto ConcreteSubject cambia, l'oggetto ConcreteSubject chiama `notify()` che chiamerà `update()` sugli oggetti ConcreteObserver per aggiornarli



5

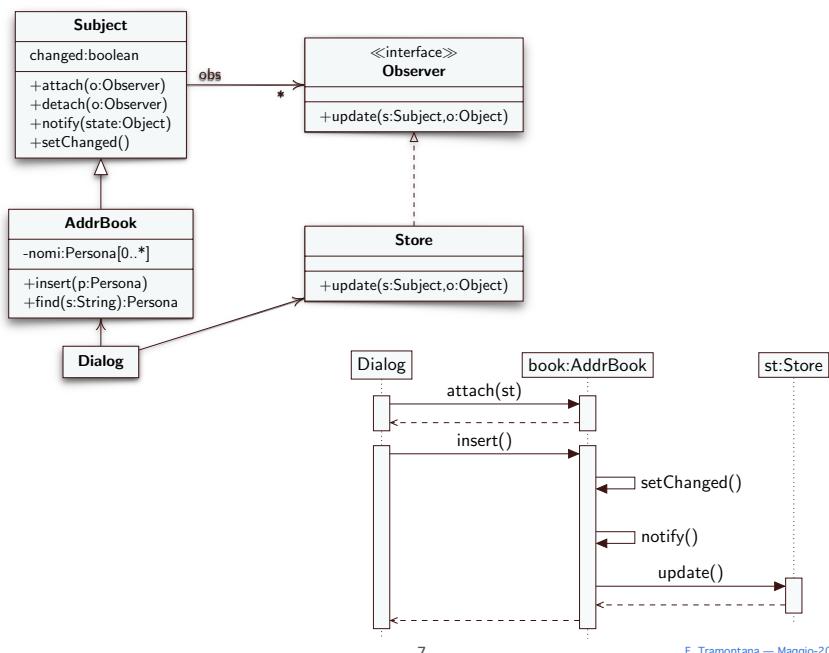
E. Tramontana — Maggio-2020

# Prima e Dopo l'Uso di Observer



6

E. Tramontana — Maggio-2020



7

E. Tramontana — Maggio-2020

```

public class Subject {
    private List<Observer> obs = new ArrayList<>();
    private boolean changed = false;
    public void notify(Object state) {
        if (!changed) return;
        for (Observer o : obs) o.update(this, state);
        changed = false;
    }
    public void setChanged() {
        changed = true;
    }
    public void attach(Observer o) {
        obs.add(o);
    }
    public void detach(Observer o) {
        obs.remove(o);
    }
}

public interface Observer {
    public void update(Subject s, Object o);
}

public class Store implements Observer {
    @Override
    public void update(Subject s, Object o) {
        List<Persona> l = (List<Persona>) o;
        String nom;
        try (FileWriter f = new FileWriter("nomi.txt")) {
            for (Persona p : l) {
                nom = p.getNome() + "\t" + p.getCognome() +
                    "\t" + p.getTelefono();
                f.write(nom + "\n");
            }
        } catch (IOException e) { }
    }
}

public class AddrBook extends Subject {
    private List<Persona> nomi = new ArrayList<>();

    public void insert(Persona p) {
        if (nomi.contains(p)) return;
        nomi.add(p);
        setChanged(); // la prossima notifica avverrà
        notify(nomi); // notifica i ConcreteObserver
    }

    public Persona find(String cognome) {
        for (Persona p : nomi)
            if (p.getCognome().equals(cognome)) return p;
        System.out.println("AddrBook.find: NOT found");
        return null;
    }
}

public class Dialog {
    private static final AddrBook book =
        new AddrBook();
    private static final Store st = new Store();
    private static final Persona p1 =
        new Persona("Oliver", "Stone", "012345", "NY");

    public static void main(String[] args) {
        book.attach(st);
        book.insert(p1);
    }
}

```

The code provides the implementation of the `Subject`, `Observer`, `Store`, `AddrBook`, and `Dialog` classes. The `Subject` maintains a list of observers and a `changed` flag. The `Observer` interface defines the `update` method. The `Store` class implements the `Observer` interface and handles the file output. The `AddrBook` class extends `Subject` and implements the `insert` and `find` methods. The `Dialog` class is used to demonstrate the usage of the pattern.

8

E. Tramontana — Maggio-2020

# Observer

- Conseguenze

- Il Subject conosce solo la classe Observer e non ha bisogno di conoscere le classi ConcreteObserver. ConcreteSubject e ConcreteObserver non sono accoppiati quindi più facili da riusare e modificare
- La notifica inviata da un ConcreteSubject è mandata a tutti gli oggetti che si sono iscritti, il ConcreteSubject non sa quanti sono. Gli osservatori possono essere rimossi in qualunque momento.  
L'osservatore sceglie se gestire o ignorare la notifica
- L'aggiornamento da parte del Subject può far avviare tante operazioni sugli Observer e altri oggetti per gli aggiornamenti. La notifica non dice agli Observer cosa è cambiato nel ConcreteSubject, è un evento che indica il completamento di un'operazione del ConcreteSubject

9

E. Tramontana — Maggio-2020

## Avvertenze

- Con tanti Subject e pochi Observer, anziché tenere riferimenti a Observer nei Subject, si può usare una sola tabella associativa (in comune fra i Subject) per ridurre lo spazio impegnato
- Un Observer potrebbe osservare più oggetti, per sapere chi ha mandato la notifica, come parametro di update(), si manda il riferimento al Subject
- Subject chiama notify() dopo un cambiamento, oppure aspetta un certo numero di cambiamenti, in modo da evitare continue notifiche agli Observer
- Per mantenere la consistenza fra Observer e Subject, un cambiamento dell'Observer deve essere comunicato al Subject (con setState())
- Quando si vuol eliminare un Subject, gli Observer dovrebbero essere avvisati per cancellare il loro riferimento al Subject
- Il Subject può passare ulteriori informazioni quando chiama update(), modello push; anziché aspettare che l'Observer legga lo stato, modello pull
- Gli Observer che si registrano possono specificare gli eventi di interesse, in modo che con l'update() si manda solo ciò che è cambiato

E. Tramontana — Maggio-2020

# Observer In Java

- Il problema affrontato dal design pattern Observer è così comune che la sua soluzione è fornita nella libreria java.util, con i tipi Observable e Observer
- Observable svolge il ruolo di Subject quindi tiene traccia di tutti gli oggetti ConcreteObserver che vogliono essere informati di un cambiamento. Observable notifica il cambiamento di stato quando il metodo `notifyObservers()` viene chiamato
- La classe Observable ha una variabile (flag) che indica se lo stato è cambiato dalla precedente notifica, è impostato dal metodo `setChanged()`. La chiamata a `setChanged()` è da effettuare all'interno dei metodi della classe ConcreteSubject secondo la logica desiderata
- Observer è una interfaccia che ha solo il metodo `update()` e può avere un argomento che indica quale oggetto ha causato l'aggiornamento

11

E. Tramontana — Maggio-2020

## Reactive Streams

- Quando il ConcreteSubject chiama `notify()`, questo chiama `update()` che esegue sul thread del chiamante, costringendo il chiamante ad aspettare l'esecuzione di `update()` di ciascun ConcreteObserver
- Più recentemente, con i Reactive Streams, si è cercato di risolvere il problema del passaggio di un insieme di item da un **publisher** a un **subscriber** senza bloccare il publisher e senza inondare il subscriber
- Con la versione 9 di Java (settembre 2017), Observable e Observer sono disponibili per compatibilità con versioni precedenti, ma sconsigliati
- Java 9 fornisce nella libreria `java.util.concurrent` le interfacce `Publisher<T>`, `Subscriber<T>`, `Subscription`, e la classe `SubmissionPublisher`. Quest'ultima implementa un Publisher dedicando un thread per mandare ciascun messaggio ai Subscriber

12

E. Tramontana — Maggio-2020

# Publisher Subscriber Java 9

- Publisher è l'astrazione per fornire item, definisce il metodo `subscribe()` che prende in input `Subscriber<T>`
- SubmissionPublisher implementa Publisher e invia item ai Subscriber in maniera asincrona. Quando sul SubmissionPublisher è chiamato il metodo `submit()`, esso esegue in un thread dedicato (asincrono) la chiamata al metodo `onNext()` dei Subscriber, e si blocca se il subscriber non può ricevere l'item
- Subscriber è usato per ricevere item, definisce i metodi `onComplete()`, `onError()`, `onNext()`, `onSubscribe()`
- Subscription definisce il collegamento fra Publisher e Subscriber; i metodi `cancel()` e `request(n)` permettono al Subscriber di fermare l'invio di messaggi e di richiedere l'invio dei prossimi n messaggi

13

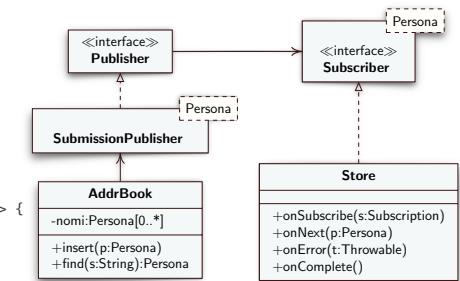
E. Tramontana — Maggio-2020

```
public class AddrBook {
    private List<Persona> nomi = new ArrayList<>();
    private SubmissionPublisher<Persona> publ = new SubmissionPublisher<>();
    public void attach(Subscriber<Persona> s) {
        publ.subscribe(s);
    }
    public boolean insert(Persona p) {
        if (nomi.contains(p)) return false;
        nomi.add(p);
        publ.submit(p);
        return true;
    }
}

public class Store implements Subscriber<Persona> {
    private Subscription sub;
    @Override
    public void onSubscribe(Subscription s) {
        sub = s;
        sub.request(1);
    }
    @Override
    public void onNext(Persona p) {
        String nom = p.getNome() + "\t" + p.getCognome();
        System.out.println("Store onNext: "+nom);
        sub.request(1);
    }
    @Override
    public void onError(Throwable throwable) {
        System.out.println("In Store: errore");
    }
    @Override
    public void onComplete() {
        System.out.println("In Store: completato");
    }
}
```

14

E. Tramontana — Maggio-2020



# Model View Controller (MVC)

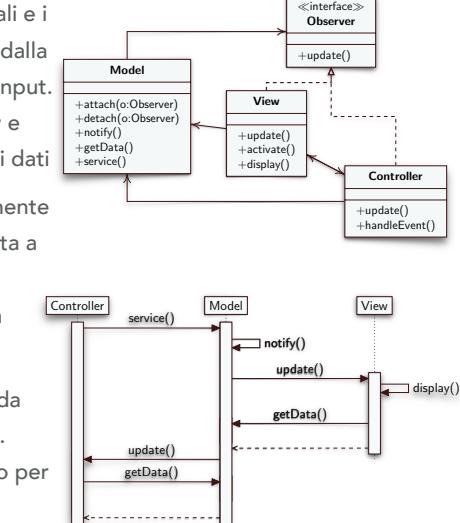
- E' considerato un pattern architettonale per le applicazioni interattive, che individua tre componenti: **Model** per funzionalità principali e dati; **View** per mostrare i dati; **Controller** per prendere gli input dell'utente
- Motivazioni
  - Le interfacce utente possono cambiare, poiché funzionalità, dispositivi o piattaforme cambiano
  - Le stesse informazioni sono presentate in finestre differenti (per es. sotto forma di grafici diversi)
  - Le visualizzazioni devono subito adeguarsi alle manipolazioni sui dati
  - I cambiamenti all'interfaccia utente dovrebbero essere facili
  - Il supporto ai diversi modi di visualizzazione non dovrebbe avere a che fare con le funzionalità principali

15

E. Tramontana — Maggio-2020

# Model View Controller (MVC)

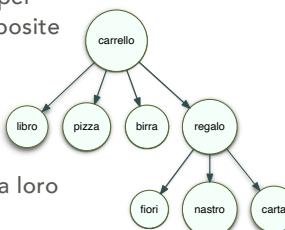
- Model** incapsula le funzionalità principali e i dati dell'applicazione. E' indipendente dalla rappresentazione degli output e dagli input. Registra View e Controller. Avvisa View e Controller registrati dei cambiamenti di dati
- View** mostra i dati all'utente. Generalmente ci sono tante View, ogni View è associata a un Controller. View inizializza il proprio Controller, e mostra i dati che legge da Model
- Controller** riceve gli input dell'utente (da mouse e tastiera) sotto forma di eventi. Traduce gli eventi in richieste di servizio per Model o avvisa View



E. Tramontana — Maggio-2020

# Composite

- **Intento:** Comporre oggetti in strutture ad albero per rappresentare gerarchie di parti o del tutto. Composite permette ai client di trattare oggetti singoli e composizioni di oggetti uniformemente



- **Motivazione**

- E' necessario **raggruppare** elementi semplici tra loro per formare elementi più grandi
- Se nell'implementazione c'è **distinzione** tra classi per elementi semplici e classi per contenitori di questi elementi semplici, il codice che usa queste classi deve trattarli in modo differente. Questa distinzione rende il codice più complicato
- Composite permette di descrivere una composizione **ricorsiva**, in modo che i client non debbano fare distinzione tra tipi di elementi. I client tratteranno tutti gli oggetti della struttura uniformemente

1

Prof. Tramontana - Maggio 2020

# Esempi

- **Esempio 1: File**

- Le operazioni su disco permettono la gestione di file (immagini, testo, etc.) e la gestione di cartelle. Esempi di operazioni: creazione, lettura, scrittura, esecuzione

- Le classi client devono poter fare operazioni su file e **cartelle**, senza distinguere fra essi

- Una cartella deve poter contenere al suo interno sia file che altre cartelle, queste a sua volta contengono altri elementi, e così via

- **Esempio 2: Figure**

- In un editor di figure ho sia figure semplici, es. Linea, Box, che figure composte, ovvero **raggruppamenti** di figure semplici e composte

- Le classi client devono poter trattare allo stesso modo sia le figure semplici che quelle composte, quando avviano operazioni come disegna, sposta, etc.

2

Prof. Tramontana - Maggio 2020

# Composite

- **Soluzione**

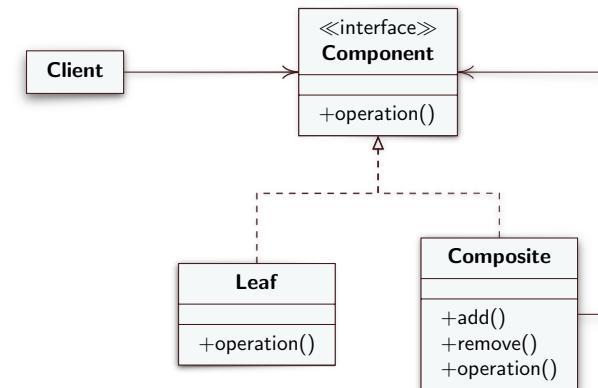
- **Component:** interfaccia (o classe abstract) che rappresenta elementi semplici e non; dichiara le operazioni degli oggetti della composizione; implementa le operazioni comuni alle sottoclassi, in modo appropriato; dichiara le operazioni per l'accesso e la gestione degli elementi semplici; può definire un'operazione che permette ad un elemento di accedere all'oggetto padre nella struttura ricorsiva
- **Leaf:** classe che rappresenta elementi semplici (detti child/children); è sottoclasse di Component; implementa il comportamento degli oggetti semplici
- **Composite:** classe che rappresenta elementi contenitori; è sottoclasse di Component; definisce il comportamento per l'aggregato di elementi child; tiene il riferimento a ciascuno degli elementi child; implementa operazioni per gestire elementi child

3

Prof. Tramontana - Maggio 2020

# Composite

- **Soluzione**



4

Prof. Tramontana - Maggio 2020

# Composite

- Collaborazioni
  - I client usano l'interfaccia di Component per interagire con elementi della struttura composita. Se il ricevente del messaggio è Leaf, la richiesta è gestita direttamente. Se il ricevente è un Composite, questo invia la richiesta ai suoi child e possibilmente avvia operazioni addizionali prima e dopo
- Conseguenze
  - Elementi semplici possono essere composti in elementi più complessi, questi possono essere composti, e così via (ovvero composizione ricorsiva)
  - Un client che si aspetta un elemento semplice può riceverne uno composto
  - I client sono semplici, trattano strutture composte e semplici uniformemente. I client non devono sapere se trattano con un Leaf o un Composite
  - Nuovi tipi di elementi (Leaf o Composite) possono essere aggiunti e potranno funzionare con la struttura ed i client esistenti
  - Non è possibile a design time vincolare il Composite solo su certi componenti Leaf (in base al tipo), invece dovranno essere fatti dei controlli a runtime

5

Prof. Tramontana - Maggio 2020

# Composite

- Ulteriori dettagli
  - Per facilitare la navigazione degli oggetti, gli elementi child mantengono il riferimento all'oggetto che li contiene. Il riferimento è inserito in Component, mentre Leaf e Composite possono implementare le operazioni per gestirlo
  - Per avere client che non distinguono se trattano con Leaf o Composite (è uno degli obiettivi), la classe Component dovrebbe definire quante più operazioni in comune possibile. Le classi Leaf e Composite faranno override delle operazioni
  - Dove dichiarare le operazioni di gestione di child, add() e remove()?
    - Se dichiarate in Component si ha trasparenza, ma per le classi Leaf tali operazioni non hanno significato. La sicurezza nell'uso è quindi compromessa, poiché i client potrebbero avvarle su Leaf. Se i client avviano un'operazione add() su Leaf e si ignora la richiesta, questo potrebbe indicare un difetto del programma
    - Se dichiarate in Composite si ha sicurezza, poiché a compile time si verifica che tali operazioni non possono essere chiamate su Leaf, ma si perde la trasparenza

6

Prof. Tramontana - Maggio 2020

# Composite

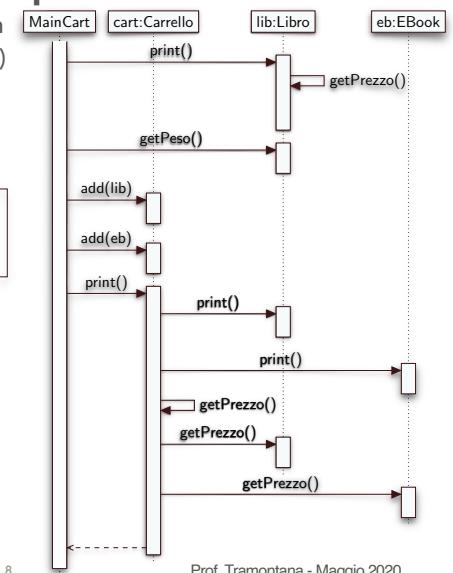
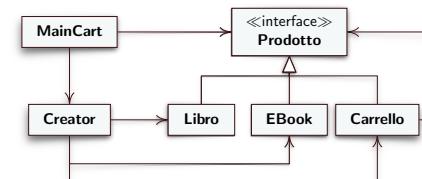
- Si può inserire una operazione getComposite() in Component che ritorna null e che è ridefinita in Composite per ritornare il riferimento a se stesso. I client dovrebbero comunque distinguere il tipo di risultato e fare operazioni differenti, niente trasparenza
- La lista che contiene elementi child deve essere definita in Composite, altrimenti se fosse definita in Component si sprecherebbe spazio, poiché ogni Leaf avrebbe tale variabile anche se non deve usarla mai
- L'ordinamento di child per un Composite potrebbe essere importante e va tenuto in considerazione su certe implementazioni
- Il Composite potrebbe implementare una **cache**, per ottimizzare le prestazioni, è utile quando si implementa un'operazione che deve ricercare tra tutti i suoi componenti child. I componenti child devono poter accedere ad un'operazione che permette di invalidare la cache del Composite

7

Prof. Tramontana - Maggio 2020

# Esempio

- Si vuol gestire un prodotto e un insieme di prodotti (nel carrello) allo stesso modo



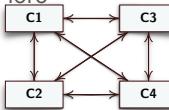
8

Prof. Tramontana - Maggio 2020

# Mediator

- Intento

- Definire un oggetto che incapsula come un gruppo di oggetti interagisce. Il Mediator promuove il lasco accoppiamento fra oggetti poiché evita che essi interagiscano direttamente, e permette di modificare le loro interazioni indipendentemente da essi



- Motivazione

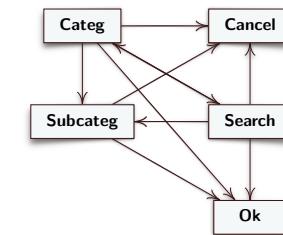
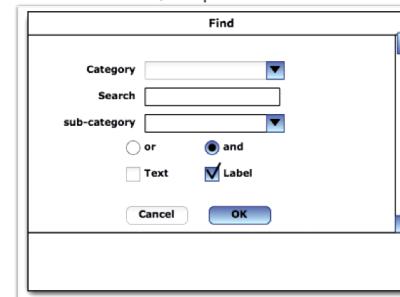
- La distribuzione di responsabilità fra vari oggetti può risultare in molte connessioni fra oggetti, nel caso peggiore un oggetto conosce tutti gli altri
- Molte connessioni rendono un oggetto dipendente da altri e l'intero sistema si comporta come se fosse monolitico. Inoltre potrebbe essere difficile cambiare il comportamento del sistema poiché il comportamento è distribuito fra oggetti
- Si possono evitare questi problemi incapsulando il comportamento collettivo in un oggetto *mediatore* separato. Il mediatore serve da intermediario ed evita che gli oggetti dipendano fra loro

Prof. Tramontana - Giugno 2020

# Mediator

- Per la finestra di ricerca mostrata

- Ogni elemento visualizzato (testo, lista, bottone, etc.) è controllato da una corrispondente classe
- Ciascuna classe deve comunicare il suo stato alle altre per far aggiornare la visualizzazione
- Ciascuna classe (senza un Mediator) chiamerà i metodi di tutte le altre classi, e quindi ciascuna classe è dipendente dalle altre

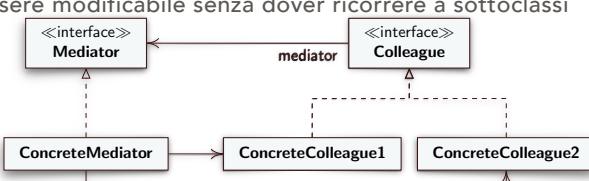


Prof. Tramontana - Giugno 2020

# Mediator

- Applicabilità

- Usare il Mediator quando
  - Un insieme di oggetti comunicano in modo ben definito ma complesso. Le interdipendenze che ne risultano sono non strutturate e difficili da comprendere
  - Riusare un oggetto è difficile poiché esso comunica con tanti altri oggetti
  - Un comportamento che è distribuito fra tante classi dovrebbe essere modificabile senza dover ricorrere a sottoclassi



Design Pattern Mediator

3

Prof. Tramontana - Giugno 2020

# Mediator

- Soluzione

- Isolare le comunicazioni (complesse) tra oggetti dipendenti creando una classe separata per esse
- **Mediator** definisce una interfaccia (punto di incontro) per gli oggetti connessi **Colleague**
- **ConcreteMediator** implementa il comportamento cooperativo e coordina oggetti **Colleague**
- Ogni **Colleague** conosce il **Mediator**, e comunica con il Mediator quando avrebbe comunicato con un altro **Colleague**
- I **ConcreteColleague** mandano richieste, e ricevono richieste, a un oggetto **Mediator**. Il **Mediator** implementa il comportamento cooperativo inoltrando le richieste a opportuni **ConcreteColleague**

4

Prof. Tramontana - Giugno 2020

# Mediator

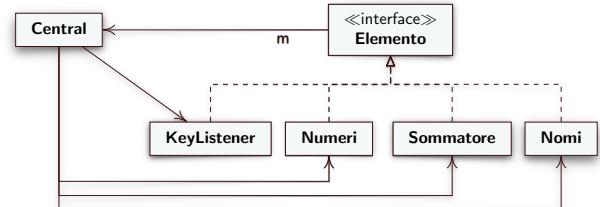
- Conseguenze
- La maggior parte della complessità che risulta nella gestione delle dipendenze è spostata dagli oggetti cooperanti al Mediator. Questo rende gli oggetti più facili da implementare e mantenere
- Le classi Colleague sono più riusabili poiché la loro funzionalità fondamentale non è mischiata con il codice che gestisce le dipendenze
- Il codice del Mediator non è in genere riusabile poiché la gestione delle dipendenze implementata è solo per una specifica applicazione

5

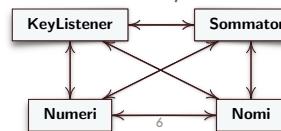
Prof. Tramontana - Giugno 2020

# Esempio

- Si legge un dato dalla tastiera e si compiono delle operazioni, su numeri o su stringa in base al dato letto
- La classe KeyListener legge da tastiera un dato e ritorna il valore letto a Central (un Mediator), quest'ultima chiama i metodi dei ConcreteColleague Numeri, Sommatore, Nomi, in base al tipo di dato letto



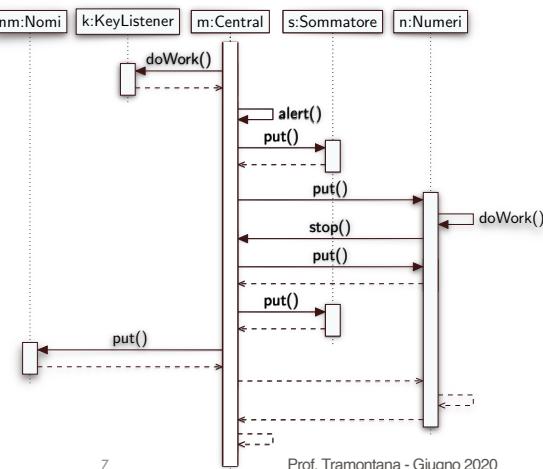
- Nel caso non si adottasse il Mediator, le varie classi si chiamerebbero fra loro



Prof. Tramontana - Giugno 2020

# Esempio

- Il Mediator Central avvia la lettura da tastiera tramite il metodo doWork() di KeyListener e ottiene da esso il valore letto, quindi Central chiama put() sugli oggetti interessati al valore letto
- Quando un oggetto ConcreteColleague riconosce una condizione di arresto, chiama stop() su Central, che avvisa gli altri ConcreteColleague
- In figura si mostra il caso in cui Numeri chiama stop() su Central



7

Prof. Tramontana - Giugno 2020

# Definizioni

- Manutenzione
  - Il processo di introduzione di modifiche ad un prodotto software dopo la sua consegna al cliente
- Evoluzione
  - Spesso usato con lo stesso significato di manutenzione, oppure
  - Singolo passo di un processo di manutenzione che prevede: evoluzione, rilascio di patch, rimozione sistema

(Ing. E. Tramontana - Evoluzione, metriche - 9 -Giu-06) 1

# Dati statistici

- I costi di manutenzione rappresentano il 67-80% dei costi del software
- I cambiamenti si possono raggruppare in 4 categorie
  - *Correttivi* - rimozione errori (17%)
  - *Adattativi* - aggiustamenti per un nuovo ambiente (18%)
  - *Perfettivi* - miglioramento e aggiunta di funzionalità (60%)
  - *Preventivi* - modifiche interne per prevenire problemi (5%)
- Incorporare nuove funzionalità è la porzione maggiore di modifiche
  - E' buona pratica anticipare i cambiamenti a design time (tramite parametrizzazione, incapsulamento, etc.)

(Ing. E. Tramontana - Evoluzione, metriche - 9 -Giu-06) 2

# Dinamiche di evoluzione

- Sono i processi di cambiamento di un sistema
- Si basano su studi empirici
  - Effettuati da Lehman e Belady (dal 1968 e confermati da studi recenti)
  - Su sistemi software di grandi dimensioni sviluppati da grandi aziende

(Ing. E. Tramontana - Evoluzione, metriche - 9 -Giu-06) 3

# Leggi di Lehman

- Cambiamento continuo [1974]
  - I sistemi hanno bisogno di essere continuamente adattati altrimenti diventano progressivamente meno soddisfacenti
- Aumento della complessità [1974]
  - Quando un sistema evolve, la sua struttura aumenta di complessità, a meno che del lavoro viene fatto per preservare o semplificare la sua struttura
- Auto-regolazione [1974]
  - Attributi come dimensione, intervallo tra release e numero di errori trovati in ciascuna release sono approssimativamente invarianti
- Stabilità organizzativa [1978]
  - Durante la vita di un sistema il suo tasso di sviluppo è circa costante e indipendente dalle risorse impiegate per lo sviluppo

(Ing. E. Tramontana - Evoluzione, metriche - 9 -Giu-06) 4

## Leggi di Lehman

- Conservazione di familiarità [1978]
  - In media, l'incremento di crescita di un sistema tende a rimanere costante o a diminuire
- Continua crescita [1978]
  - Il contenuto di funzioni di un sistema deve continuamente essere incrementato per mantenere la soddisfazione dell'utente
- Diminuzione della qualità [1994]
  - La qualità di un sistema diminuisce se non viene rigorosamente gestita ed adattata durante i cambiamenti

(Ing. E. Tramontana - Evoluzione, metriche - 9 -Giu-06) 5

## Applicabilità delle leggi

- Sono in generale applicabili a grandi sistemi sviluppati da grandi organizzazioni
- Non è chiaro come si adattano a
  - Piccoli prodotti
  - Prodotti che incorporano un certo numero di COTS
  - Piccole organizzazioni

(Ing. E. Tramontana - Evoluzione, metriche - 9 -Giu-06) 6

## Costo di manutenzione

- Fattori che contribuiscono al costo
  - Il costo è ridotto se il team di sviluppo è coinvolto nella manutenzione
  - Gli sviluppatori potrebbero non avere responsabilità contrattuali per la manutenzione, quindi non hanno incentivi a fare un design che può essere cambiato in futuro
  - La struttura del programma si degrada mano a mano che si introducono cambiamenti

(Ing. E. Tramontana - Evoluzione, metriche - 9 -Giu-06) 7

## Modelli di manutenzione

- Modelli di manutenzione
  - Quick-fix
    - Cambiamenti a livello del codice
  - Miglioramento iterativo
    - Cambiamenti fatti in base ad un'analisi del sistema esistente
    - Controllo della complessità e mantenimento del design
  - Riuso
    - Stabilire i requisiti per il nuovo sistema, riusando il più possibile

(Ing. E. Tramontana - Evoluzione, metriche - 9 -Giu-06) 8

# Tipi di modifiche

- Re-factoring o re-structuring
  - Processo di cambiamento del software che non altera il comportamento del codice ma migliora la struttura interna
  - Ovvero: prendere un sistema fatto male e modificarlo per ottenere una struttura ben fatta
- Reverse engineering
  - Analizzare un sistema per estrarre informazioni sul suo comportamento o sulla sua struttura
- Re-engineering
  - Alterare un sistema per ricostituirlo in un'altra forma

(Ing. E. Tramontana - Evoluzione, metriche - 9 -Giu-06) 9

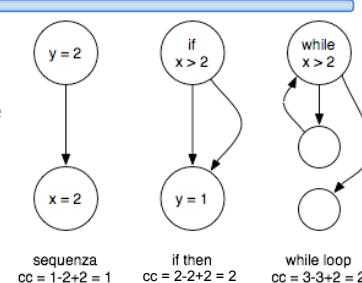
# Metriche

- Una metrica definisce un set di misure per un sistema software
- Obiettivi dell'adozione di metriche
  - Monitorare il prodotto mentre si costruisce
  - Identificare i livelli per ciascuna metrica
  - Rimediare in caso i livelli non sono soddisfacenti
- Solo gli attributi interni possono essere misurati direttamente (es. dimensione), quelli esterni sono ricavati indirettamente

(Ing. E. Tramontana - Evoluzione, metriche - 9 -Giu-06) 10

# Metriche tradizionali

- Complessità ciclomatica (cc)
  - Valuta la complessità di un algoritmo
  - cc è il numero di test necessari per valutare esaurientemente l'algoritmo
    - Per una sequenza -> un solo test
    - Per una condizione -> due test
  - cc = numero di test = archi - nodi + 2
- Dimensione
  - LOC (lines of code), oppure NCNB (non comment non blank)
- Comment Percentage
  - Percentuale di commenti rispetto a LOC (consigliato 30%)



(Ing. E. Tramontana - Evoluzione, metriche - 9 -Giu-06) 11

# Metriche OO

- Suite di Chidamber & Kemerer o metriche CK
  - WMC (Weighted Methods per Class)
    - La somma delle complessità dei metodi per una classe
      - Se i metodi hanno pari complessità, WMC è il numero di metodi della classe
  - DIT (Depth of Inheritance Tree)
    - Massimo numero di livelli dalla classe alla radice della gerarchia (radice=0)
  - NOC (Number of Children of a Class)
    - Numero di sottoclassi di una classe della gerarchia
  - CBO (Coupling Between Object Classes)
    - Numero di classi con cui una classe interagisce
  - RFC (Response for a Class)
    - Numero di metodi eseguiti al ricevimento di un messaggio, cioè numero di metodi di una classe + numero di metodi invocati da ciascuno di essi
  - LCOM (Lack of Cohesion of Methods)
    - Per ogni campo della classe, si calcola la percentuale di metodi che usano tale campo; si mediano le percentuali e si sottrae dal 100%

(Ing. E. Tramontana - Evoluzione, metriche - 9 -Giu-06) 12

# Interpretazione metriche CK

- WMC
  - Alto valore di WMC indica grande lavoro di manutenzione, grande specificità della classe e quindi poca possibilità di riuso
- DIT
  - Più alto è DIT per una classe, più difficile è capirne il comportamento, data la grande quantità di metodi ereditati
  - Alto DIT indica maggiore complessità dell'intero sistema, ma anche maggiore riuso
- NOC
  - Indica l'influenza che una classe ha sul sistema
  - Maggiore è NOC, maggiore è il riuso

# Interpretazione metriche CK

- CBO
  - Maggiore è CBO maggiore è la dipendenza di una classe da altre, quindi minore possibilità di riuso, maggiore difficoltà a comprendere, modificare e correggere la classe
- RFC
  - Alto RFC indica grande complessità, quindi difficoltà di comprensione e di testing
- LCOM
  - Alta coesione (basso LCOM) indica semplicità, elevata possibilità di riuso

# Java

- I nuovi paradigmi e linguaggi tendono a semplificare il lavoro del programmatore, nascondendo dentro le librerie (o nei costrutti del linguaggio) parte della programmazione precedentemente necessaria
- Da Java 5 (settembre 2004), in un ciclo, si può scorrere una lista, o un array, senza dover dare il valore di inizio e fine dell'indice (*ciclo for avanzato*)

```
List<String> nomi;
...
for (String nome : nomi) ...;
```
- Da Java 7 (luglio 2011), si può evitare di indicare il tipo di elementi nella collection al momento dell'istanziazione, lasciando al compilatore l'inferenza, ovvero

```
List<String> nomi = new ArrayList<>();
```
- Java è un linguaggio imperativo, tuttavia Java 8 (marzo 2014) include caratteristiche tipiche della programmazione funzionale. La programmazione funzionale è in genere più concisa, espressiva, e facile da parallelizzare rispetto alla programmazione ad oggetti

Prof. Tramontana - Maggio 2020

# Classi Anonime In Java

- Da Java 1.1 si possono usare classi anonime per implementare interfacce

```
public interface Hello {
    public void greetings(String s);
}

public class Sera {
    private Hello myh;
    public Sera(Hello h) {
        myh = h;
    }
    public void saluti() {
        myh.greetings("buonasera");
    }
}

public class Saluti implements Hello {
    public void greetings(String s) {
        System.out.println("Ciao, "+s);
    }
}

public class MainSal {
    public static void main(String[] args) {
        Sera sr = new Sera(new Hello() {
            public void greetings(String s) {
                System.out.println("Ciao, "+s);
            }
        });
        sr.saluti();
    }
}
```

2 Prof. Tramontana - Maggio 2020

# Espressioni Lambda

- Il codice sotto è un'espressione lambda  
`s -> System.out.println("Ciao, " + s)`
- Un'espressione lambda è una funzione anonima, che prende in ingresso parametri, con nome dato a sinistra del segno freccia, e un blocco di codice, a destra del segno freccia. È simile a un metodo: ha una lista di parametri, un corpo e un tipo di ritorno. Può essere passata come argomento di un metodo. È concisa: non serve scrivere preamboli
- Nei linguaggi funzionali puri, un'espressione lambda è una funzione pura, ovvero il risultato dipende solo dagli input (non ha uno stato). In Java, un'espressione lambda può accedere a uno stato esterno (può non essere pura)
- La seguente è un'espressione lambda che prende in ingresso due parametri, `x` e `y`, e implementa la somma, il risultato è il valore di ritorno  
`(x, y) -> x + y`
- I parametri in ingresso sono `x` e `y`, a sinistra della freccia, il cui tipo è determinato automaticamente. Quando non ha parametri, o ne ha più di uno, occorre racchiudere i parametri fra parentesi tonde  
`() -> System.out.println("Buongiorno")`
- Il corpo della funzione anonima è il codice a destra della freccia, nel caso di più di un'istruzione, occorre racchiuderle fra parentesi graffe  
`(x, y) -> { System.out.println("x: " + x); return x+y; }`

3

Prof. Tramontana - Maggio 2020

# Implementazione Interfaccia

```
public interface Hello {
    public void greetings(String s);
}
```

- La classe anonima implementa l'interfaccia Hello con codice fornito nel punto dove si istanzia  

```
Sera sr = new Sera(new Hello() {
    public void greetings(String s) {
        System.out.println("Ciao, "+s);
    }
}); sr.saluti();
```
- Con Java 8 le espressioni lambda si evitano i preamboli (o codice standard, boilerplate), e si implementa solo il comportamento, quando l'interfaccia ha un singolo metodo  

```
// Java 8
Sera sr = new Sera(s2 -> System.out.println("Ciao, " + s2));
sr.saluti();
```
- L'espressione lambda implementa il metodo dichiarato in Hello, e si passa al costruttore di Sera. Per il compilatore, `s2` è di tipo `String` poiché così è il parametro del metodo in Hello

4

Prof. Tramontana - Maggio 2020

# Cerca Valori Su Lista

```
public class Trova {  
    private List<String> listaNomi = Arrays.asList("Nobita", "Nobi",  
    "Suneo", "Honekawa", "Shizuka", "Minamoto", "Takeshi", "Gouda");  
  
    // in stile imperativo  
    public void trovaImper() {  
        boolean trovato = false;  
        for (String nome : listaNomi)  
            if (nome.equals("Nobi")) {  
                trovato = true;  
                break;  
            }  
        if (trovato) System.out.println("Nobi trovato");  
        else System.out.println("Nobi non trovato");  
    }  
  
    // in stile dichiarativo  
    public void trovaDichiar() {  
        if (listaNomi.contains("Nobi")) System.out.println("Nobi trovato");  
        else System.out.println("Nobi non trovato");  
    }  
}
```

5

Prof. Tramontana - Maggio 2020

# Considerazioni

- L'implementazione imperativa si serve di una variabile boolean come flag per indicare se l'elemento è stato trovato. Inoltre si usa un ciclo per scorrere la lista e controllare se uscire dal ciclo
- Lo stile imperativo dà al programmatore il controllo di quel che il programma deve fare, tuttavia
  - Bisogna implementare varie linee di codice, e molto spesso si hanno cicli che scorrono liste per trovare valori, calcolare somme, etc.
  - Per capire cosa si vuol fare, dobbiamo prima leggere tanti dettagli all'interno del corpo del ciclo
  - Il ciclo è esterno al codice che implementa la lista (l'iterazione è esterna)
- Nella versione dichiarativa vari dettagli dell'implementazione sono nella libreria sottostante (sul metodo contains() della classe ArrayList), l'iterazione è interna

6

Prof. Tramontana - Maggio 2020

# Stile Funzionale

- Lo stile di programmazione funzionale è dichiarativo. La programmazione funzionale aggiunge allo stile dichiarativo funzioni di ordine più alto, ovvero funzioni che hanno come parametri altre funzioni
- In Java si possono passare oggetti ai metodi, creare oggetti dentro i metodi, e ritornare oggetti dai metodi
- In Java 8 si possono passare funzioni ai metodi, creare funzioni dentro i metodi e ritornare funzioni dai metodi
- Un metodo è una parte di una classe, mentre una funzione non è associata ad una classe
- Un metodo o una funzione che ricevono, creano o ritornano una funzione, si considerano essere funzioni di ordine più alto

7

Prof. Tramontana - Maggio 2020

# Java Collection

- Le librerie di Java hanno tante interfacce e classi collection. Collection è un'interfaccia che definisce i metodi add(), remove(), size(), contains(), containsAll(), etc.
  - List definisce i metodi get(), indexOf(), set(), etc.
- |            | get      | add      | contains | remove   |
|------------|----------|----------|----------|----------|
| ArrayList  | costante | costante | O(n)     | O(n)     |
| LinkedList | O(n)     | costante | O(n)     | costante |
| TreeSet    |          | O(lg n)  | O(lg n)  |          |
- 
- ```
classDiagram
    class Collection {
        <<interface>>
        add()
        remove()
        clear()
    }
    class List {
        <<interface>>
        get()
        add()
        remove()
        size()
    }
    class Set {
        <<interface>>
        add()
        remove()
        clear()
    }
    class AbstractList {
        <<abstract>>
        add()
        remove()
        clear()
    }
    class LinkedList {
        <<concrete>>
        add()
        remove()
        clear()
    }
    class ArrayList {
        <<concrete>>
        add()
        remove()
        clear()
    }
    class AbstractSet {
        <<abstract>>
        add()
        remove()
        clear()
    }
    class TreeSet {
        <<concrete>>
        add()
        remove()
        clear()
    }
```
- ArrayList è un array espandibile: cresce del 50% quando non vi è spazio. Gli elementi sono contigui, quindi l'accesso al generico elemento è veloce
  - LinkedList è una lista, ogni elemento ha i riferimenti al successivo e al precedente.
  - Vector è simile ad ArrayList, ma è synchronized

8

Prof. Tramontana - Maggio 2020

# Metodi Di Default

- Java 8 introduce i metodi di default per le interfacce
- Ciò ha permesso di includere nuovi metodi su interfacce già esistenti, senza compromettere la compatibilità delle applicazioni conformi a precedenti versioni di Java
- stream() è un metodo di default dell'interfaccia Collection
- I metodi di default non agiscono sullo stato, invece le classi astratte che hanno metodi implementati possono avere costruttori e metodi che agiscono sullo stato
- stream() restituisce uno Stream<T> che è una sequenza di elementi di tipo T (T è definito dalla collection su cui è invocato stream()). stream() permette di fare operazioni sugli elementi presenti sull'istanza di collection su cui è invocato
- Il tipo Stream definisce vari metodi che prendono in ingresso funzioni

9

Prof. Tramontana - Maggio 2020

# Metodo Filter

- Si abbia una lista di String, contare quante volte è presente un certo valore
- ```
List<String> nomi = List.of("Nobita", "Nobi", "Suneo");
long c = nomi.stream()
    .filter(s -> s.equals("Nobi"))
    .count();
```
- of() è un metodo statico di List (da Java 9) che restituisce una lista non-modificabile contenente gli elementi passati
  - filter(Predicate<T> p) è un metodo di Stream, prende come argomento una funzione che ritorna un boolean (quindi solo true o false). Una funzione che ritorna un boolean è detta predicato. filter() ritorna uno Stream costituito da tutti gli elementi della lista che soddisfano il predicato passato in input
  - L'espressione lambda s -> s.equals("Nobi") è un predicato che ha in ingresso s, che è un elemento dello stream, e restituisce un boolean. Inoltre, equals() è un metodo di String che restituisce un boolean
  - filter() è lazy, ovvero operazione intermedia, può essere eseguita in un momento successivo rispetto a dove è inserita, è eseguita quando è necessario
  - count() è un metodo di Stream, è eager, ovvero operazione terminale, genera un valore e forza l'esecuzione delle precedenti operazioni
  - Tutte le operazioni che restituiscono uno Stream sono operazioni intermedie

10

Prof. Tramontana - Maggio 2020

# Esempi Con Filter

- Contare quanti elementi della lista nomi hanno lunghezza 5 caratteri
  - Si noti che non si può usare il metodo contains() di List (che è invece utile per verificare se una lista contiene un certo elemento)

```
long c = nomi.stream()
    .filter(s -> s.length() == 5)
    .count();
```

- L'espressione lambda s -> s.length() == 5 ha in ingresso s, ovvero un elemento dello stream; su s si invoca length() (metodo di String che restituisce la lunghezza di s), quindi si valuta se è pari a 5

- Contare quanti elementi della lista nomi sono stringhe vuote

```
long c = nomi.stream()
    .filter(s -> s.isEmpty())
    .count();
```

- isEmpty() è un metodo di String (non ha parametri di ingresso e restituisce un boolean). Tale metodo è passato a filter(), e sarà chiamato su ciascun elemento dello stream, quindi ciascun elemento dello stream è valutato da isEmpty()

11

Prof. Tramontana - Maggio 2020

# Tipo Predicate

- Si può definire una funzione che restituisce un boolean
- ```
Predicate<Integer> positive = x -> x >= 0;
```
- Predicate rappresenta un'interfaccia funzionale, ovvero un'interfaccia che definisce un solo metodo
  - L'annotazione @FunctionalInterface indica al compilatore che l'interfaccia ha solo un metodo astratto, così il compilatore controlla che abbia un solo metodo (evita che versioni successive inseriscano altri metodi)
  - Predicate definisce il metodo test() che prende in ingresso un parametro di tipo Object
  - In questo esempio, il metodo test() ha parametro in ingresso x di tipo Integer, e la sua implementazione è x >= 0;
  - Il predicato positive si può passare ad un metodo
- ```
Stream<Integer> result = Stream.of(2, 5, 10, -1)
    .filter(positive);
```
- Il metodo of() è un metodo statico di Stream che costruisce uno Stream

12

Prof. Tramontana - Maggio 2020

# Metodo Reduce

- `reduce(T identity, BinaryOperator<T> accumulator)` è un metodo di `Stream`, prende un valore dello stesso tipo degli elementi dello stream, e un'espressione lambda che ha due valori in ingresso e ritorna un valore
  - `reduce()` si usa quando si vuol passare da un insieme di valori ad un singolo valore, per esempio per ottenere la somma
- ```
reduce(0, (accum, v) -> accum + v);
```
- `reduce()` applica la funzione di accumulazione che gli viene passata (secondo argomento), in questo caso la somma, a tutti gli elementi dello stream e ritorna un risultato dello stesso tipo dei valori nello stream
  - Ciascun valore dello stream è rappresentato dal parametro `v`
  - La somma calcolata per ciascun `v` è tenuta dalla variabile `accum` e inizialmente vale `0` (valore fornito dal primo parametro di `reduce()`)
  - `reduce()` è un'operazione terminale

13

Prof. Tramontana - Maggio 2020

# Riferimenti A Metodi

- La sintassi `::` permette di avere il riferimento a un metodo, che si può quindi passare a un altro metodo
  - Es. `Integer::sum` è il riferimento al metodo statico `sum()` di `Integer` che prende due parametri `Integer` e ne ritorna la somma
  - Passando il riferimento `Integer::sum` alla `reduce()`, verrà calcolata la somma degli elementi dello stream, come quando si passa l'espressione lambda `(s, v) -> s + v`
- ```
reduce(0, Integer::sum)
```
- `reduce()` può prendere in ingresso solo un'espressione lambda, e in tal caso restituisce un tipo `Optional<T>` che può contenere il risultato
- ```
reduce(Integer::sum)
```
- Il risultato non esiste se lo stream di partenza è vuoto
  - `Optional` rappresenta un valore che può esistere o meno, se esiste il metodo `isPresent()` darà un valore `true`; per estrarre il valore si usa `get()`

14

Prof. Tramontana - Maggio 2020

# Reduce

```
public class Pagamenti {  
    private List<Float> importi = new ArrayList<>();  
  
    // in stile imperativo  
    public float calcolaSommaImper() {  
        float risultato = 0f;  
        for (float v : importi)  
            risultato += v;  
        return risultato;  
    }  
  
    // in stile funzionale  
    public float calcolaSomma() {  
        return importi.stream()  
            .reduce(0f, Float::sum);  
    }  
}
```

15

Prof. Tramontana - Maggio 2020

# Metodo Map

- `map(Function<T, R> mapper)` di `Stream` prende in ingresso una funzione `mapper`, e restituisce uno stream contenente i risultati dell'esecuzione della funzione su ciascun elemento dello stream iniziale
  - Ovvero, `map()` chiama su ciascun elemento dello stream iniziale la funzione passata e dà in uscita il risultato della funzione. Ciascun risultato è inserito in un nuovo stream
  - `map()` è un'operazione intermedia
- ```
map(x -> x * 2)
```
- Eseguito su uno stream contenente valori interi, restituisce uno stream contenente i valori iniziali raddoppiati
- ```
map(Persona::getEta)
```
- Eseguito su uno stream di istanze di `Persona`, restituisce uno stream contenente i valori in uscita da `getEta()`, quest'ultimo è un metodo di `Persona` ed è invocato su ciascun elemento dello stream iniziale

16

Prof. Tramontana - Maggio 2020

```

public class Persona {
    private String nome;
    private int eta;
    public Persona(String n, int e) {
        nome = n;
        eta = e;
    }
    public String getNome() {
        return nome;
    }
    public int getEta() {
        return eta;
    }
}

List<Persona> p = Arrays.asList(new Persona("Saro", 24),
    new Persona("Taro", 21), new Persona("Ian", 19), new Persona("Al", 16));

```

- Calcoliamo la somma delle età in stile funzionale

```

int somma = p.stream()
    .map(Persona::getEta)
    .reduce(0, Integer::sum);

```

- La funzione passata a map() è il metodo getEta() di Persona

17

```

// in stile imperativo
int somma = 0;
for (Persona x : p)
    somma += x.getEta();

```

Prof. Tramontana - Maggio 2020

## Dichiarativo–Funzionale

- Data una lista contenente valori String

```
List<String> nomi = Arrays.asList("Saro", "Taro", "Ian", "Al");
```

- Per determinare se la lista contiene un certo valore, in stile dichiarativo

```
if (nomi.contains("Saro")) System.out.println("Saro trovato");
```

- Data una lista contenente istanze di Persona

```
List<Persona> p = Arrays.asList(new Persona("Saro", 24),
    new Persona("Taro", 21), new Persona("Ian", 19), new Persona("Al", 16));
```

- Lo stile funzionale consente di estrarre il campo nome, e inoltre permette di valutare una funzione ad-hoc, quindi è molto più flessibile e potente

```

long c = p.stream()
    .filter(s -> s.getNome().equals("Saro"))
    .count();

```

- count() conta quanti elementi ha lo stream prodotto da filter

18

Prof. Tramontana - Maggio 2020

## Ricerca Su Lista (Imperativa)

- Data la lista di istanze di Persona, trovare il nome della persona che è più grande (di età) fra quelli che hanno meno di 20 anni

```

List<Persona> p = Arrays.asList(new Persona("Saro", 24),
    new Persona("Taro", 21), new Persona("Ian", 19), new Persona("Al", 16));

```

- In versione imperativa, se volessimo scorrere la lista solo una volta

```

Persona pmax = null;
for (Persona x : p)
    if (x.getEta() < 20) {
        if (pmax == null) pmax = x;
        if (pmax.getEta() < x.getEta()) pmax = x;
    }
if (pmax != null) System.out.println("persona: " + pmax.getNome());

```

- Il corpo del ciclo ha varie condizioni, queste rendono il codice più difficile da comprendere

19

Prof. Tramontana - Maggio 2020

## Ricerca Funzionale, Versione 1

- Aggiungendo su Persona il metodo getMax()

```

/** restituisce l'istanza con il valore massimo di eta' */
public static Persona getMax(Persona p1, Persona p2) {
    if (p1.getEta() > p2.getEta())
        return p1;
    return p2;
}

```

- Possiamo implementare la ricerca in modo funzionale

```

Optional<Persona> pmax = p.stream()
    .filter(x -> x.getEta() < 20)
    .reduce(Persona::getMax);

if (pmax.isPresent())
    System.out.println("persona: " + pmax.get().getNome());

```

- filter() è usata per separare gli elementi che soddisfano la condizione sull'età, prende in ingresso la funzione (predicato) da eseguire su ciascun elemento

- reduce() è usata per selezionare un elemento: invoca getMax() che confronta a due a due

20

Prof. Tramontana - Maggio 2020

# Ricerca Funzionale, Versione 2

```
Optional<Persona> pmax = p.stream()
    .filter(x -> x.getEta() < 20)
    .max(Comparator.comparing(Persona::getEta));

if (pmax.isPresent())
    System.out.println("persona: " + pmax.get().getNome());
```

- `filter()` è usata per separare gli elementi che soddisfano la condizione sull'età, prende in ingresso la funzione (predicato) da eseguire su ciascun elemento
- `max()` trova il valore massimo, è un'operazione terminale, prende un `Comparator`, restituisce un `Optional`, `max()` opera in modo simile a `reduce()`
- `comparing()` di `Comparator` prende una funzione che estrae una chiave e restituisce un `Comparator`
- `Comparator` implementa una funzione di confronto (`compare()`) che controlla l'ordinamento di una collezione di oggetti
- `max()` al suo interno chiama il metodo `compare()` del `Comparator` in input

21

Prof. Tramontana - Maggio 2020

# Metodo Collect

```
List<Persona> p = List.of(new Persona("Saro", 24),
    new Persona("Taro", 21), new Persona("Ian", 19), new Persona("Al", 16));
```

- Ricaviamo la lista delle età
- Come prima, `map()` restituisce uno stream con i valori delle età, e la funzione passata dice come trasformare ciascun elemento dello stream
- `collect()` permette di raggruppare i risultati e prende in ingresso un `Collector`
- La classe `Collectors` implementa metodi utili per raggruppamenti, il metodo `toList()` restituisce un `Collector` che accumula elementi in una `List`

```
// in stile imperativo
List<Persona> p; // come sopra
List<Integer> e = new ArrayList<>();
for (Persona x : p)
    e.add(x.getEta());
```

22

Prof. Tramontana - Maggio 2020

# Programmazione Parallelia

- I processori attuali hanno vari core, tipicamente due per i portatili, quattro per i desktop, dodici per i server. La programmazione parallela è più difficile di quella sequenziale e usare bene l'hardware risulta più complicato
- In Java, la classe `Thread` permette di lanciare un nuovo thread di esecuzione, ma spesso bisogna risolvere i problemi di corsa critica, usando opportunamente `synchronized`, `wait`, `notify`
  - Java 5 mette a disposizione `Lock`, `Esecutori`, etc.
- Le operazioni `map()` e `filter()` di `Stream` sono stateless, ovvero non tengono uno stato durante l'esecuzione, quindi il risultato non dipende dall'ordine in cui i singoli elementi su cui eseguono vengono prelevati
- Lo stream di origine non viene modificato e le operazioni `map()`, `filter()`, etc., generano uno stream distinto. Questo facilita grandemente la parallelizzazione
- Attenzione: se l'applicazione durante l'esecuzione di un'operazione, per es. di `map()`, aggiorna uno stato globale, l'operazione non è più stateless

23

Prof. Tramontana - Maggio 2020

# Stream Paralleli

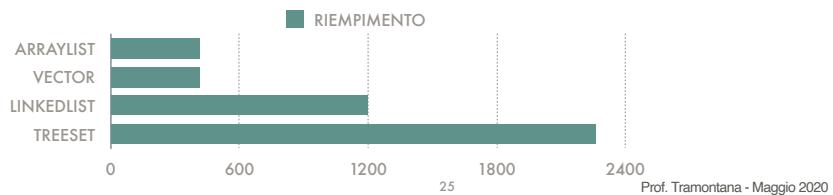
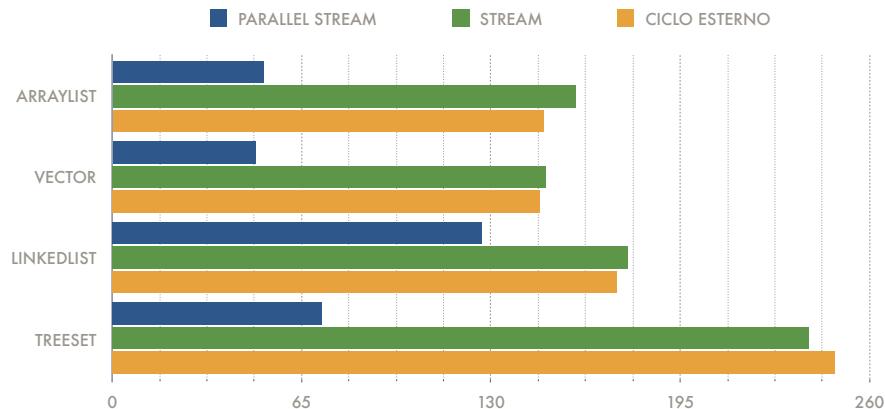
- Collection ha i metodi di default `stream()` e `parallelStream()`
- Il metodo `parallelStream()` possibilmente dà uno stream parallelo. Quindi si può avere l'esecuzione parallela basata su `parallelStream()` (niente più bisogno di thread e sincronizzazione, per molti casi)
- Le prestazioni dipendono da: (i) numero di elementi dello stream, (ii) operazioni da svolgere, (iii) hardware, e (iv) tipo di Collection su cui si invoca l'operazione
- La Collection potrebbe essere `ArrayList`, `LinkedList`, etc., con tempi di accesso diversi (vedere tabella precedente)
- Eseguendo su milioni di elementi il seguente codice, le prestazioni migliorano (su hardware con più core) per `ArrayList`, `Vector`, `TreeSet`; migliorano poco (o non migliorano) quando si usa `LinkedList` a causa dell'accesso sequenziale

```
long c = nomi.parallelStream()
    .map(String::toUpperCase)
    .filter(s -> s.equals("NOBI"))
    .count();
```

24

Prof. Tramontana - Maggio 2020

Hardware 4 core, Ricerca su 5 Milioni di elementi, Java 11.0.2



## Esempio

- Data una lista di istanze di Persona trovare i nomi delle persone che sono giovani ed hanno ruolo Programmer, e ordinare i risultati

```
List<Persona> team = List.of(new Persona("Kent", 29, "CTO"), new Persona("Luigi", 25, "Programmer"), new Persona("Andrea", 26, "GrLeader"), new Persona("Sofia", 26, "Programmer"));

team.stream()
    .filter(p -> p.giovane())
    .filter(p -> p.isRuolo("Programmer"))
    .sorted(Comparator.comparing(Persona::getNome))
    .forEach(p -> System.out.print(p.getNome() + " "));

// Output: Luigi Sofia
```

- filter() operazione intermedia che restituisce gli elementi che soddisfano il predicato passato
- sorted() operazione intermedia stateful che restituisce uno stream che ha gli elementi ordinati in base al Comparator passato
- comparing() permette di estrarre la chiave per il confronto
- forEach() operazione terminale che esegue un'azione su ciascun elemento dello stream (su uno stream parallelo l'ordine non è garantito) Prof. Tramontana - Maggio 2019

## Esempio

- Data una lista di istanze di Persona trovare i diversi ruoli

```
team.stream()
    .map(p -> p.getRuolo())
    .distinct()
    .forEach(s -> System.out.print(s + " "));

// Output: CTO Programmer GrLeader
```

- distinct() operazione intermedia stateful che restituisce uno stream di elementi distinti
- Data una lista di istanze di Persona trovare il nome di una che ha il ruolo Programmer

```
Optional<Persona> r = team.stream()
    .filter(p -> p.isRuolo("Programmer"))
    .findAny();

if (r.isPresent()) System.out.println(r.get().getNome());
```

// Output: Luigi

- findAny() (simile a findFirst()) operazione terminale che restituisce un Optional, per valutarla non è necessario esaminare tutto lo stream, si dice short-circuiting (può far sì che alcune parti non eseguano)

Prof. Tramontana - Maggio 2019

## Stateless–Stateful

- Le operazioni map() e filter() sono stateless, ovvero non hanno uno stato interno, prendono un elemento dello stream e danno zero o un risultato
- Le operazioni come reduce(), max() accumulano un risultato. Quest'ultimo ha una dimensione limitata, indipendente da quanti elementi vi sono nello stream. Il risultato in una passata viene dato in ingresso alla passata successiva
- Le operazioni come sorted() e distinct() devono conoscere gli altri elementi dello stream per poter eseguire, si dicono stateful

## Generare Stream: iterate()

- Gli stream possono essere generati a partire da una funzione tramite le operazioni iterate() e generate(). Queste operazioni creano stream infiniti
- iterate() restituisce un stream infinito e ordinato dato dall'esecuzione iterativa di un funzione f applicata inizialmente ad un elemento seme, quindi produce uno stream di seme, f(seme), f(f(seme)), etc.
- limit() tronca lo stream ad una lunghezza pari al numero indicato

```
Stream.iterate(2, n -> n * 2)
    .limit(10)
    .forEach(System.out::println);
```

- Il codice sopra dà lo stream di 10 elementi che consiste in 2, 4, 8, ... 1024
- Ad ogni passo, dopo il primo, il valore in input alla funzione è il valore calcolato dalla funzione al passo precedente
- L'operazione iterate() è sequenziale

## Generare Stream: generate()

- Il metodo generate() permette di produrre uno stream infinito di valori, tramite una funzione di tipo Supplier, ovvero che fornisce un valore
- generate() non applica una funzione ad ogni nuovo valore prodotto, come invece fa iterate()

```
Stream.generate(() -> Math.round(Math.random()*10))
    .limit(5)
    .forEach(System.out::println);
```

- Il codice sopra genera uno stream di 5 numeri casuali, ciascuno fra 0 e 10

5

Prof. Tramontana - Maggio 2019

## Tipo IntStream

- IntStream rappresenta uno stream di valori int

```
IntStream.rangeClosed(1, 6)
    .map(x -> x*x)
    .forEach(System.out::println);
```

- Il codice sopra genera e stampa i quadrati dei numeri da 1 a 6
- rangeClosed() restituisce una sequenza di int nell'intervallo specificato (estremi inclusi) con incremento pari a 1
- sum() somma i valori presenti nello stream IntStream

```
int v = IntStream.rangeClosed(1, 5).sum();
// Output: v = 15
```

6

Prof. Tramontana - Maggio 2019

## Tipo IntStream

```
int result =
Stream.of("truth", "flows", "to", "them", "sweetly", "by", "nature")
    .mapToInt(x -> x.length()).sum();
// Output: result = 31
```

- mapToInt() esegue la funzione passata e restituisce un IntStream

```
Stream<Integer> s = IntStream.rangeClosed(1, 10).boxed();
• boxed() restituisce uno Stream di Integer a partire da un IntStream
• Si abbia la lista che contiene istanze di Persona, si generi uno stream contenente i primi 4 elementi
```

```
List<Persona> lista;
```

```
Stream<Persona> p =
    IntStream.rangeClosed(0, 3)
        .mapToObj(i -> lista.get(i));
```

- mapToObj() restituisce uno stream di oggetti a partire da un IntStream

7

Prof. Tramontana - Maggio 2019

## Debug

- Per esigenze di debug potrei voler conoscere come è fatto lo stream mano a mano che si eseguono le operazioni

```
List<Integer> numbers = Arrays.asList(2, 3, 4, 5);
numbers.stream()
    .map(x -> x + 17)
    .filter(x -> x % 2 == 0)
    .limit(3)
    .forEach(System.out::println);
```

```
// Output: 20 22
```

- Sarebbe utile capire cosa produce ciascuna operazione. Il metodo forEach() consuma l'elemento dello stream, quindi non si può usare

8

Prof. Tramontana - Maggio 2019

## Debug Con Peek

```
List<Integer> numbers = Arrays.asList(2, 3, 4, 5);
numbers.stream()
    .peek(x -> System.out.println("from stream: " + x))
    .map(x -> x + 17)
    .peek(x -> System.out.println("after map: " + x))
    .filter(x -> x % 2 == 0)
    .peek(x -> System.out.println("after filter: " + x))
    .limit(3)
    .peek(x -> System.out.println("after limit: " + x))
    .collect(Collectors.toList());
```

// Output:  
// from stream: 2  
// after map: 19  
// from stream: 3  
// after map: 20  
// after filter: 20  
// after limit: 20  
// from stream: 4  
// after map: 21  
// from stream: 5  
// after map: 22  
// after filter: 22  
// after limit: 22

9

Prof. Tramontana - Maggio 2019

## Esercizio 1 Con Java 8

- Data una lista di stringhe
  - Esempio: {"author", "auto", "autocorrect", "begin", "big", "bigger", "biggish"}
  - Produrre una lista che contiene solo le stringhe che cominciano con un certo prefisso noto
    - Esempio: se il prefisso è "au", la lista prodotta è {"author", "auto", "autocorrect"}
- Suggerimento
  - Usare il metodo `substring(int beginIndex, int endIndex)` della classe `String` che restituisce la sottostringa che inizia a `beginIndex` e termina a `endIndex`
  - Esempio: "ciao".`substring(0, 2)` restituisce "ci"

10

Prof. Tramontana - Maggio 2019

## Esercizio 2 Con Java 8

- Data una lista di stringhe
  - Esempio: {"to", "speak", "the", "truth", "and", "pay", "your", "debts"}
  - Produrre una stringa contenente le iniziali di ciascuna stringa della lista
  - Esempio: per la lista sopra si produrrà la stringa "tsttapyd"

11

Prof. Tramontana - Maggio 2019

## Esercizio 3 Con Java 8

- Data una lista di terne di numeri interi
    - Per ciascuna terna verificare se essa costituisce un triangolo
    - Restituire la lista dei perimetri per le terne che rappresentano triangoli
  - Suggerimenti
    - In un triangolo, ciascun lato è minore della somma degli altri due
    - Si può rappresentare la terna come un array di tre elementi interi
- ```
int[] t = new int[] { 2, 3, 4 };
```
- Si può rappresentare la lista di terne come lista di array di interi
- ```
List<int[]> lista;
```

12

Prof. Tramontana - Maggio 2019

## Esercizio 3

- Soluzione

```
private List<int[]> terne = Arrays.asList(new int[] { 2, 2, 3 },
    new int[] { 3, 2, 3 }, new int[] { 3, 3, 3 }, new int[] { 3, 4, 5 },
    new int[] { 5, 2, 3 });

private List<Integer> verifica() {
    return terne.stream()
        .filter(t -> t[0]<t[1]+t[2])
        .filter(t -> t[1]<t[0]+t[2])
        .filter(t -> t[2]<t[0]+t[1])
        .map(t -> t[0] + t[1] + t[2])
        .collect(Collectors.toList());
}
```

13

Prof. Tramontana - Maggio 2019

## Esercizio 4

- Data una lista di numeri interi
  - Verificare se ciascuna terna formata prendendo dalla lista tre numeri contigui costituisce un triangolo
  - Esempio: lista {2, 3, 5, 7, 8}, terne {2, 3, 5}, {3, 5, 7}, {5, 7, 8}
  - Restituire la lista delle terne che rappresentano triangoli
  - Esempio: terne {3, 5, 7}, {5, 7, 8}
- Suggerimenti
  - In un triangolo, ciascun lato è minore della somma degli altri due
  - Si generano gli indici da 0 a n-2
  - Si conserva in un array di tre elementi, per ciascun indice i, l'elemento i, ed i due successivi elementi, così da formare una terna

14

Prof. Tramontana - Maggio 2019

## Esercizio 4

- Soluzione

```
private List<Integer> lista = List.of(2, 2, 4, 6, 3, 6, 3, 3, 4, 5);

private List<int[]> verifica() {
    return IntStream.rangeClosed(0, lista.size() - 3)
        .mapToObj(i -> new int[] { lista.get(i), lista.get(i+1), lista.get(i+2) })
        .filter(t -> t[0] < t[1] + t[2])
        .filter(t -> t[1] < t[0] + t[2])
        .filter(t -> t[2] < t[0] + t[1])
        .collect(Collectors.toList());
}
```

15

Prof. Tramontana - Maggio 2019

## Esercizio 5

- Data una lista di numeri interi positivi
  - Verificare se la lista è ordinata
- Suggerimenti
  - Si generano gli indici da 0 a n-1
  - Per ciascun valore dell'indice i, si confrontano l'elemento con indice i ed il successivo, se il secondo è minore del primo la lista non è ordinata e si può fermare la verifica

16

Prof. Tramontana - Maggio 2019

## Esercizio 5

- Soluzione 1
  - Ogni iterazione accede a due elementi della lista
  - L'operazione filter emette l'indice i dell'elemento che è più grande del successivo
  - Appena filter trova un elemento, con l'operazione findAny ferma la ricerca
  - Nessuna operazione conserva uno stato globale

```
private List<Integer> lista = Arrays.asList(2, 2, 4, 6, 12, 3);
private boolean isOrdinata() {
    return IntStream.rangeClosed(0, lista.size() - 2)
        .filter(i -> lista.get(i) > lista.get(i+1))
        .peek(v -> System.out.print(lista.get(v) + " > " + lista.get(v+1)))
        .findAny()
        .isEmpty();
}
// 12 > 3
```

17

Prof. Tramontana - Maggio 2019

## Esercizio 5

- Soluzione 2
  - Si conserva uno stato che è condiviso fra varie iterazioni

```
private List<Integer> lista = Arrays.asList(2, 2, 4, 6, 3, 6, 3, 3, 4, 5);
private int prec; // conserva l'elemento precedente, e' lo stato condiviso
private boolean isOrdinata() {
    prec = lista.get(0);
    return lista.stream()
        .filter(v -> seMinoreDiPrec(v)) // modifica prec, scarta valori false
        .findAny() // si ferma quando vi e' un false
        .isEmpty();
}
private boolean seMinoreDiPrec(int x) { // non puo' eseguire in parallelo
    int p = prec;
    prec = x; // modifica lo stato
    return x < p; // ritorna true se elemento corrente > elemento prec
}
```

18

Prof. Tramontana - Maggio 2019

## Considerazioni

- Per la soluzione 1, si ha

```
.filter(i -> lista.get(i) > lista.get(i+1))
```
- l'espressione lambda passata a filter legge due numeri dalla lista e dà in output un boolean, senza altri effetti collaterali (side-effect-free), ovvero **non modifica uno stato condiviso**. Tale espressione lambda è una **funzione pura**
- Per la soluzione 2, si ha

```
.filter(v -> seMinoreDiPrec(v))
ovvero
.filter(v -> { int p = prec; prec = v; return v < p; })
```
- l'espressione lambda passata a map modifica un attributo. Tale modifica è un effetto collaterale (voluto), quindi non è una funzione pura
- Si noti che: (i) prec è un attributo, definito al di fuori dell'espressione lambda che può essere acceduto da essa (accessi al contesto sono consentiti); (ii) la modifica di uno stato condiviso non permette l'esecuzione parallela

19

Prof. Tramontana - Maggio 2019

# Software e difetti

- Il software con difetti è un grande problema
- I difetti nel software sono comuni
- Come sappiamo che il software ha qualche difetto?
  - Conosciamo tramite ‘qualcosa’, che non è il codice, cosa un programma dovrebbe fare
  - Tale ‘qualcosa’ è una specifica
  - Tramite il comportamento anomalo, il software sta comunicando qualcosa -> i suoi difetti -> questi non devono passare inosservati

E. Tramontana - Testing - 10 Gen-08

1

# Processo di V & V

- Si dovrebbe applicare il processo di V&V ad ogni fase durante lo sviluppo
- Il processo di V&V ha due obiettivi principali: scoprire i difetti del sistema e valutare se il sistema è usabile in una situazione operativa
- I difetti possono essere raggruppati, in base alle fasi di sviluppo
  - Difetti di specifiche: la descrizione di ciò che il prodotto fa è ambigua, contraddittoria o imprecisa
  - Difetti di progettazione: le componenti o le loro interazioni sono progettati in modo non corretto, le cause: algoritmi (es. divisione per zero), strutture dati (es. campo mancante, tipo sbagliato), interfaccia moduli (parametri di tipo inconsistente), etc.
  - Difetti di codice: errori derivanti dall'implementazione dovuti a poca comprensione del progetto o dei costrutti del linguaggio di (es. overflow, conversione tipo, priorità delle operazioni aritmetiche, variabili non inizializzate, non usate tra due assegnazioni, etc.)
  - A volte è difficile classificare se un difetto è di progettazione o di codice
  - Difetti di test: i casi di test, i piani per i test, etc. possono avere difetti

E. Tramontana - Testing - 10 Gen-08

3

# Verifica e Validazione (V & V)

- Obiettivo di V & V: assicurare che il sistema software soddisfi i bisogni dei suoi utenti
- Verifica
  - Stiamo costruendo il prodotto nel modo giusto?
  - Il sistema software dovrebbe essere conforme alle sue specifiche
- Validazione (convalida)
  - Stiamo costruendo il giusto prodotto?
  - Il sistema software dovrebbe fare ciò che l'utente ha realmente richiesto

E. Tramontana - Testing - 10 Gen-08

2

# Test

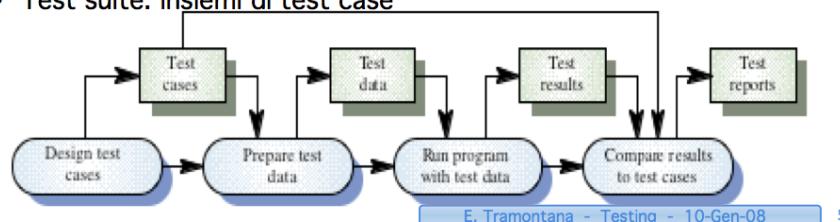
- Il test del software
  - Può rivelare la presenza di errori, non la loro assenza
  - Un test ha successo se scopre uno o più errori
  - I test dovrebbero essere condotti insieme alle verifiche sul codice statico
  - La fase di test ha come obiettivo rivelare l'esistenza di difetti in un programma
- Il debugging si riferisce alla localizzazione ed alla correzione degli errori
- Debugging
  - Formulare ipotesi sul comportamento del programma
  - Verificare tali ipotesi e trovare gli errori

E. Tramontana - Testing - 10 Gen-08

4

# Test: definizioni

- Dati di test (test data)
  - Dati di input che sono stati scelti per testare il sistema
- Casi di test (test case)
  - Dati di input per il sistema e output stimati per tali input nel caso in cui il sistema operi secondo le sue specifiche
    - Gli input sono non solo parametri da inviare ad una funzione, ma anche eventuali file, eccezioni, e stato del sistema, ovvero le condizioni di
- Test suite: insiemi di test case



## Testing

- Solo un test esaustivo può mostrare se un programma è privo di difetti
  - I test esaustivi sono impraticabili
    - Es. Una funzione che prende in ingresso 2 int, per essere testata esaustivamente dovrebbe essere eseguita  $2^{32} \times 2^{32}$  volte, ovvero circa  $1.8 \times 10^{19}$  volte
    - Se la funzione esegue in 1ns =  $10^{-9}$ s occorrono  $1.8 \times 10^{10}$ s ovvero, essendo  $1Y = 3 \times 10^7$ , 600 anni!
  - Priorità
    - I test dovrebbero mostrare le capacità del software più che eseguire i singoli componenti
    - Il test delle vecchie funzionalità è più importante del test delle nuove
    - Testare situazioni tipiche è più importante rispetto a testare situazioni limite

E. Tramontana - Testing - 10-Gen-08 7

## Difficoltà per chi fa i test (tester)

- Deve avere una conoscenza vasta delle discipline di ingegneria del software
- Deve avere conoscenza ed esperienza su come un software è descritto (specifiche), progettato e sviluppato
- Deve essere in grado di gestire molti dettagli
- Deve conoscere quali tipi di fault possono generare i costrutti del codice
- Deve ragionare come uno scienziato per proporre ipotesi che spiegano la presenza di tipi di difetti
- Deve avere una buona comprensione del dominio del software
- Deve creare e documentare casi di test, quindi selezionare gli input che con maggiore probabilità possono rivelare difetti
- Necessita di lavorare e cooperare con chi si occupa di requisiti, design, sviluppo codice e spesso con clienti ed utenti

E. Tramontana - Testing - 10-Gen-08 6

## Test sotto stress

- Eseguire il sistema oltre il massimo carico previsto consente di rendere evidenti i difetti presenti
- Il sistema eseguito oltre i limiti consentiti non dovrebbe fallire in modo catastrofico
- Test di stress indagano su perdite, di servizio o dati, ritenute inaccettabili
- Particolarmente rilevanti per i sistemi distribuiti che possono subire degradazioni in dipendenza delle condizioni della rete
- PS: completare le specifiche in accordo ai risultati dei test

E. Tramontana - Testing - 10-Gen-08 8

## Test sotto stress

- Stress
  - Prestazioni: inserire i dati con frequenza molto alta, o molto bassa
  - Strutture dati: funziona per qualsiasi dimensione dell'array?
  - Risorse: test con poca memoria RAM, numero basso di file che possono essere aperti, connessioni di rete, etc.

## Testing manuale

- I casi di test sono liste di istruzioni per una persona
  - Click su “login”
  - Inserisci username e password
  - Click su “ok”
  - Inserisci il dato ...
- Molto comune, poiché
  - Non sostituibile: test di usabilità
  - Non pratico da automatizzare: troppo costoso
  - Le persone che fanno i test non sanno gestire automatismi complessi

## Testing automatico

- Registrare un test manuale e rieseguirlo automaticamente
  - Con macro, script, programmi appositi (es. AutoHotkey)
  - Spesso poco robusto
    - Smette di funzionare se cambia qualcosa dell'ambiente (es. posizione campi, nome campi, etc.)
- Sviluppare programmi che eseguono il test sul codice
  - Chiamano funzioni, confrontano risultati, etc.

## Copertura del codice

- Fino a quando dovremmo continuare a fare test?
- Metrica: Copertura del codice (Code coverage)
  - Dividere il programma in unità (es. costrutti, condizioni, comandi)
  - Definire la copertura che dovrebbe avere la suite di test (es. 60%)
  - Copertura codice = numero di unità già eseguite / numero di unità del programma
- Si smette di eseguire test quando si è raggiunta la copertura desiderata
- Avere una copertura del 100% non significa non avere difetti
  - Pensare ad esempio ai dati di input scelti
- Parti critiche del sistema possono avere copertura maggiore di altre parti
- La misura di copertura permette di capire se alla suite di test manca qualcosa

# Decorator

- **Intento:** Aggiungere ulteriori responsabilità ad un oggetto dinamicamente. I decorator forniscono un'alternativa flessibile all'implementazione di sottoclassi per estendere funzionalità
- **Motivazione**
  - Alcune volte si vogliono aggiungere responsabilità a singoli oggetti, e non all'intera classe. Per aggiungere responsabilità si potrebbe far uso dell'ereditarietà, tuttavia non si avrebbe flessibilità: un client non può controllare come e quando *decorare* un componente
  - Un approccio più flessibile è racchiudere (wrap) il componente in un altro oggetto che aggiunge una responsabilità. L'oggetto *wrapper* è un decorator. Il decorator è conforme all'interfaccia che decora, quindi la sua presenza è trasparente agli oggetti che usano il componente
  - Le responsabilità possono essere sottratte dinamicamente. I decorator possono essere annidati ricorsivamente, per aggiungere più responsabilità
  - A volte la creazione di sottoclassi non è praticabile. Un numero grande di estensioni produrrebbe un numero enorme di sottoclassi per gestire tutte le combinazioni

1

Prof. Tramontana - Giugno 2020

# Decorator

- **Esempio**
  - Si vuol aggiungere la proprietà *Bordo* ad un componente *Testo*
  - Se si eredita *Testo* dalla classe *Bordo*, gli oggetti della sottoclasse avranno questa proprietà, ma non si avrà flessibilità: non si possono avere oggetti senza tale proprietà
  - Alternativa flessibile, inserire il componente *Testo* dentro un oggetto che aggiunge *Bordo*
  - L'oggetto che racchiude, chiamato *DecoratorBordo*, manda la richiesta al componente *Testo* e aggiunge altre attività prima o dopo l'invio della richiesta



2

Prof. Tramontana - Giugno 2020

# Decorator

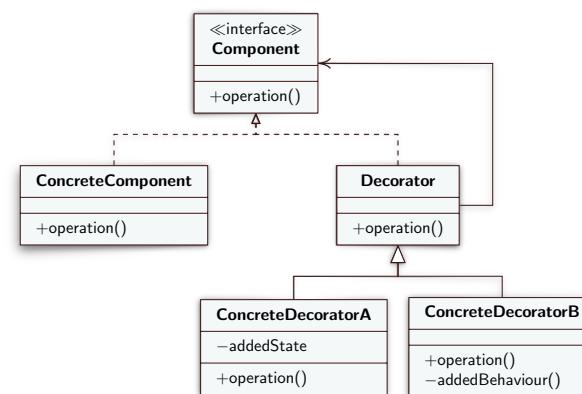
- **Soluzione**
  - **Component** definisce l'interfaccia per gli oggetti che possono avere aggiunte le responsabilità dinamicamente
  - **ConcreteComponent** definisce un oggetto su cui poter aggiungere responsabilità
  - **Decorator** mantiene un riferimento a un oggetto **Component** e definisce un'interfaccia conforme a quella di **Component**. **Decorator** inoltre le richieste al suo oggetto **Component** e può fare altre operazioni prima e dopo l'inoltro della richiesta
  - **ConcreteDecorator** implementa la responsabilità aggiunta al **Component**

3

Prof. Tramontana - Giugno 2020

# Decorator

- **Struttura**

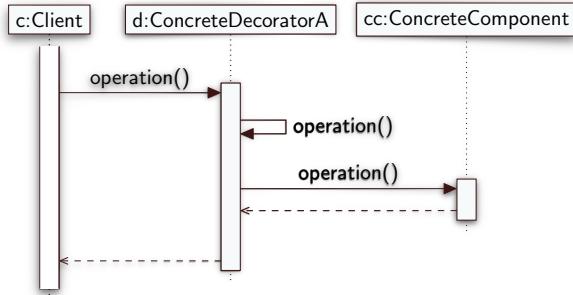


4

Prof. Tramontana - Giugno 2020

# Decorator

- Interazioni



5

Prof. Tramontana - Giugno 2020

# Implementazione

```

interface Component {
    public void operation();
}

class ConcreteComponent implements Component {
    @Override public void operation() {
        // ...
    }
}

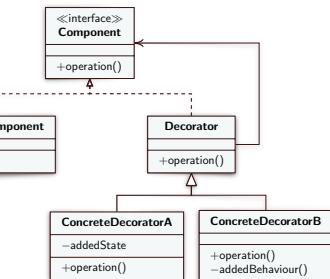
class Decorator implements Component {
    private final Component innerC;
    public Decorator(Component c) {
        innerC = c;
    }
    @Override public void operation() {
        innerC.operation();
    }
}

class ConcreteDecorator extends Decorator {
    public ConcreteDecorator(Component c) {
        super(c);
    }
    @Override public void operation() {
        super.operation();
        // ...
    }
}
  
```

Component c = new ConcreteDecorator(new ConcreteComponent());

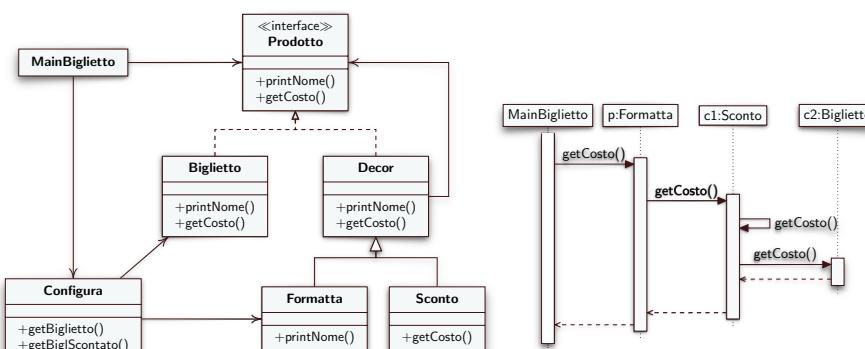
6

Prof. Tramontana - Giugno 2020



# Esempio

- Vi sono alcuni prodotti, ad es. Biglietto, ognuno con un nome ed un costo; e si hanno diversi modi di calcolare il costo dei prodotti, in base agli sconti e diversi modi di stampare i dettagli
- Si vuol poter combinare a runtime sconti (Sconto) e messaggi (Formatta) sui prodotti, per singole istanze di Biglietto



7

Prof. Tramontana - Giugno 2020

# Conseguenze

- Più flessibilità rispetto all'ereditarietà poiché si possono aggiungere responsabilità dinamicamente
- La stessa responsabilità può essere aggiunta più volte, semplicemente aggiungendo due istanze dello stesso ConcreteDecorator
- Prevedere per le classi in alto nella gerarchia tutte le responsabilità che servono significherebbe avere per esse troppe responsabilità. I ConcreteDecorator sono invece indipendenti e permettono di aggiungere responsabilità successivamente
- L'identità (tipo e riferimento) del ConcreteDecorator non è quella del ConcreteComponent, quindi non si dovrebbero confrontare i riferimenti
- Si avranno tanti piccoli oggetti, che differiscono nel modo in cui sono interconnessi

8

Prof. Tramontana - Giugno 2020

## Function

- L'interfaccia funzionale Function<T, R> definisce un metodo chiamato apply() che prende in ingresso un oggetto di tipo generico T e ritorna un tipo generico R

```
Function<String, Integer> stringLength = x -> x.length();
```

- In questo esempio, il metodo apply() ha in ingresso un parametro String, l'implementazione è x.length(), e restituisce il valore dato da length()
- Nel frammento seguente, map() usa la funzione definita sopra stringLength per calcolare la lunghezza di ogni parola dello stream, e quindi reduce() per sommare le lunghezze

```
int result =
Stream.of("truth", "flows", "to", "them", "sweetly", "by", "nature")
.map(stringLength)
.reduce(0, Integer::sum);

System.out.println(result);
// Output: 31
```

1

Prof. Tramontana - Maggio 2020

## Supplier

- Un Supplier<T> è un'interfaccia funzionale che ha un singolo metodo chiamato get() e rappresenta una funzione che non prende in ingresso alcun parametro e restituisce un valore di tipo T

```
Supplier<String> sup = () -> "ciao ciao";
```

```
String s = sup.get();
```

```
// s: ciao ciao
```

- Il codice sopra implementa un Supplier che restituisce una stringa, quando è chiamato il metodo get(). Il supplier non tiene una stringa ma il codice per generarla

2

Prof. Tramontana - Maggio 2020

## Factory Method (Breve Ripasso)

- Il design pattern Factory Method permette di creare oggetti e di nascondere la loro classe al client che li usa

```
public class Creator {
    public static Prodotto getProdotto(String name) {
        switch (name) {
            case "primo": return new ProdottoA();
            case "secondo": return new ProdottoB();
            case "terzo": return new ProdottoC();
            case "quarto": return new ProdottoD();
            default: return new ProdottoA();
        }
    }
}
```

- Nel metodo della classe Creator si sceglie quale classe istanziare
- ProdottoA, ProdottoB e ProdottoC sono sottotipi di Prodotto
- Tipicamente si avrà un chiamante

```
Prodotto p = Creator.getProdotto("primo");
```

3

Prof. Tramontana - Maggio 2020

## Factory Method Con Supplier

- Usando un Supplier  
Supplier<Prodotto> prodSupplier = ProdottoA::new;
- La linea di codice sopra è equivalente a  
Supplier<Prodotto> suppl = () -> new ProdottoA();
- Per avere istanze di sottotipi di Prodotto, si chiama get() sul Supplier  
Prodotto p1 = prodSupplier.get();
- Quindi si crea una mappa che fa corrispondere al nome di un Prodotto la sua creazione  
Map<String, Supplier<Prodotto>> map = Map.of("primo", ProdottoA::new, "secondo",
ProdottoB::new, "terzo", ProdottoC::new);
- Si usa la mappa per istanziare sottotipi di Prodotto  
public static Prodotto getProdotto(String name) {
 Supplier<Prodotto> s = map.get(name);
 if (s != null)
 return s.get();
 return new ProdottoA();
}
- Il frammento di codice sopra è equivalente a  
public static Prodotto getProdotto(String name) {
 return map.getOrDefault(name, ProdottoA::new).get();
}
- Se un costruttore ha parametri in ingresso si dovrà usare un'interfaccia funzionale appropriata (diversa da Supplier)

4

Prof. Tramontana - Maggio 2020

## Classe Persona

```
public class Persona {  
    private String nome, ruolo;  
    private int eta, costo;  
  
    public Persona(String n, int e, String r, int c) {  
        nome = n;  
        eta = e;  
        ruolo = r;  
        costo = c;  
    }  
  
    public int getCosto() {  
        return costo;  
    }  
  
    public int getEta() {  
        return eta;  
    }  
  
    public String getNome() {  
        return nome;  
    }  
  
    public String getRuolo() {  
        return ruolo;  
    }  
}
```

5

Prof. Tramontana - Maggio 2020

## Operazioni Su Istanze Di Persona

- Data una lista di istanze di Persona, stampare e contare i nomi dei programmati

```
private List<Persona> team = List.of(  
    new Persona("Al", 28, "Architect", 44),  
    new Persona("Claire", 29, "Programmer", 38),  
    new Persona("Ed", 26, "Programmer", 36),  
    new Persona("Pam", 25, "Programmer", 35),  
    new Persona("Ted", 32, "Tester", 40));  
  
public void conta(String ruolo) {  
    System.out.print("Hanno ruolo " + ruolo + ": ");  
    long c = team.stream()  
        .filter(p -> p.getRuolo().equals(ruolo))  
        .peek(p -> System.out.print(p.getNome() + ", "))  
        .count();  
    System.out.println("\nCi sono " + c + " " + ruolo);  
}  
// Output  
// Hanno ruolo Programmer: Claire, Ed, Pam,  
// Ci sono 3 Programmer  
  
// chiamante  
conta("Programmer");
```

6

Prof. Tramontana - Maggio 2020

## Operazione Su Ruoli

- Data una lista di istanze di Persona, stampare i ruoli presenti e per ciascun ruolo la lista delle persone aventi quel ruolo

```
public void scriviRuoli() {  
    team.stream()  
        .map(p -> p.getRuolo())  
        .distinct()  
        .peek(r -> System.out.print("\nRuolo " + r + ": "))  
        .forEach(r -> team.stream()  
            .filter(p -> p.getRuolo().equals(r))  
            .forEach(p -> System.out.print(p.getNome() + " ")));  
}  
  
// Output  
// Ruolo Architect: Al  
// Ruolo Programmer: Claire Ed Pam  
// Ruolo Tester: Ted
```

7

Prof. Tramontana - Maggio 2020

## Classe Pagamento

```
public class Pagamento {  
    private Persona pers;  
    private int importo;  
  
    public Pagamento(Persona p, int v) {  
        pers = p;  
        importo = v;  
    }  
  
    public Persona getPers() {  
        return pers;  
    }  
  
    public int getImporto() {  
        return importo;  
    }  
}
```

8

Prof. Tramontana - Maggio 2020

## Liste Di Pagamenti

- Data una lista di nomi di persona, creare la lista di istanze di Pagamento con il costo calcolato in base a ciascuna persona, e stampare i pagamenti

```
private List<String> daPagare = List.of("Pam", "Ed", "Ted");

private List<Pagamento> pagati = new ArrayList<>();

public void pagamenti() {
    pagati = team.stream()
        .filter(p -> daPagare.contains(p.getNome()))
        .map(p -> new Pagamento(p, p.getCosto() * 30))
        .peek(v -> System.out.print(v.getNome() + " " + v.getImporto()))
        .collect(Collectors.toList());
}

// Output
// Ed 1080  Pam 1050  Ted 1200
```

9

Prof. Tramontana - Maggio 2020

## Liste Di Buste Paga

- Data una lista di istanze di Persona, creare una lista con istanze di BustaPaga con l'importo calcolato in base al costo di ciascuna persona, e ordinare la lista per nome persona

```
private List<BustaPaga> buste;

public void generaBustaPaga() {
    buste = team.stream()
        .map(p -> new BustaPaga(p))
        .peek(b -> b.calcolaCostoBase())
        .peek(b -> b.aggiungiBonus())
        .sorted(Comparator.comparing(b -> b.getPersona().getNome()))
        .collect(Collectors.toList());
}
```

10

Prof. Tramontana - Maggio 2020

```
public class BustaPaga {
    private Persona pers;
    private int totale;

    public BustaPaga(Persona p) {
        pers = p;
    }

    public void calcolaCostoBase() {
        totale = pers.getCosto() * 30;
    }

    public void aggiungiBonus() {
        totale = (int) Math.round(totale * 1.1);
    }

    public Persona getPersona() {
        return pers;
    }

    public void stampa() {
        System.out.println(pers.getNome() + "\t " + totale + " euro");
    }

    public int getImporto() {
        return totale;
    }
}
```

11

Prof. Tramontana - Maggio 2020

## Liste Di Busta Paga

- Data la lista di istanze di BustaPaga, stampare il nome di ciascuna persona e l'importo e calcolare la somma degli importi

```
public int calcolaSomma() {
    return buste.stream()
        .peek(b -> b.stampa())
        .mapToInt(b -> b.getImporto())
        .sum();
}

// Output
// Al      1452 euro
// Claire  1254 euro
// Ed      1188 euro
// Pam     1155 euro
// Ted     1320 euro
// Totale: 6369 euro

// chiamante
System.out.println("Totale:\t " + calcolaSomma() + " euro");
```

12

Prof. Tramontana - Maggio 2020