

# HOMEWORK C - 2025

---

git & Github Intro

Java Collection Framework

git: *Crash Course*

Classi Astratte e Nidificate

Riflessione

Eccezioni e I/O

Tipi Enumerativi

# Git & GitHub: Introduzione

- Sinora abbiamo consegnato i nostri elaborati come release su Github
- GitHub è un servizio di «hosting»
  - Ovvero una applicazione Web che ospita alcuni servizi offerti da applicativi a se stanti e distinti da Github stesso
- Tra tutti, il principale è quello di versionamento dei file; offerto da un programma: `git`
  - Github rende disponibile un comodo accesso via Web
  - Altrimenti dovremmo imparare da subito molti comandi manuali di `git`
    - ✓ ci aspettano comunque dietro l'angolo... (>>)
- Permette agli sviluppatori di condividere, collaborare, e «versionare» i propri sorgenti
  - versionare: conservare la storia di tutte le modifiche di un file

# Git & GitHub: Motivazioni

- Perché versionare il codice sorgente dei propri progetti?
  - ✓ Almeno due importanti motivazioni:
    - I. Per collaborare in team di sviluppo composti da diversi programmatori
    - II. Per gestire la complessità dei progetti
- I progetti software moderni sono sempre più complicati e sempre più grandi
  - ✓ e tutto lascia presagire che in futuro lo saranno ancora più;
    - non ci si può permettere, spesso, di lavorarci da soli
- Per la gestione di progetto di una certa dimensione sono possibili solo modifiche *incrementali* e *parcellizzate*, meglio controllabili per dimensioni, costi, tempi ed in generale «risorse» impegnate

# Git & GitHub: Repository

- L'unità di raggruppamento dei file per git (e per Github) versionati prende il nome di «repository» ("deposito")
- Github (ed il sottostante git) permettono di creare repository: è stato già creato il repository "diadia" per distribuire la versione base del codice, a suo tempo
  - <https://github.com/Programmazione-Orientata-agli-Oggetti/diadia>
- Un repository:
  - contiene tutti i file di un progetto
    - volendo anche di molteplici progetti correlati che fa comodo tenere nello stesso repository
  - di ogni file viene conservata l'intera «storia», ovvero tutte le versioni con ogni modifica
    - dall'aggiunta al repository ad oggi
  - è accessibile da molteplici utenti
    - alcuni possono anche fare modifiche
      - i «collaboratori» del progetto
    - altri accedono in sola lettura

# Git & GitHub: Dove Siamo

- Grazie a Github, siamo già riusciti a
  - Creare una repository
  - Caricare tutti i file
  - Creare una release
- Siamo a buon punto per prendere ancora più consapevolezza di come funziona lo strumento
- Ci manca un ingrediente irrinunciabile:
  - Vivere i problemi che scaturiscono con la condivisione del codice
  - Ci vuole tempo ed esperienza...

*Difficile capire le soluzioni se non si è avuto il tempo di vivere i problemi che pretendono di risolvere (!)*

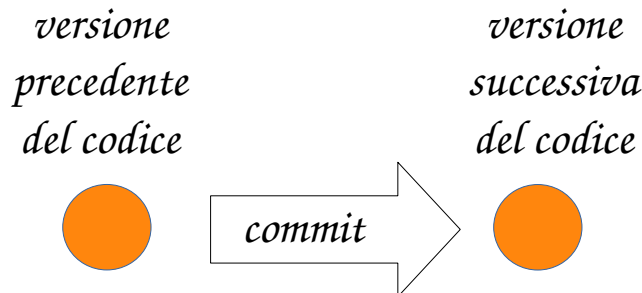
- Per questo motivo è il momento di collaborare attivamente in due persone su uno stesso progetto git & Github: HWC

# Collaboratori su Github

- Git & GitHub supportano la collaborazione tra molteplici sviluppatori
  - più persone possono così modificare un medesimo repository condiviso
- Aggiungere il collega di gruppo al proprio repository
  - Dalla pagina del proprio Repository
    - *impostazioni/setting*
    - *collaborators and teams*
  - ✓ quindi cercare ed aggiungere l'account del collega

# Git Commit: Introduzione

- `git` richiede di raggruppare ed “impacchettare” le modifiche ai file in apposite unità denominate «commit»
- Pertanto ogni file evolve da una versione alla successiva sempre e soltanto mediante un commit che lo coinvolge
  - E che può benissimo coinvolgere anche altri file, oggetto di modifiche nello stesso commit
- ✓ Pensandoci bene, anche i nostri «refactoring» hanno spesso coinvolto molteplici file
  - ✓ Es. rename di un metodo: bisogna cambiare la definizione del metodo e di tutte le sue invocazioni
  - ◆ Pertanto fa molto comodo raggruppare modifiche correlate di diversi file in un solo commit



# Git Commit

- Anche il primo caricamento dei file nel repository in realtà ha prodotto un commit sopra un repository vuoto iniziale
  - Per questo dopo l'upload dei file avevamo usato un bottone verde *Commit Changes* (<<)

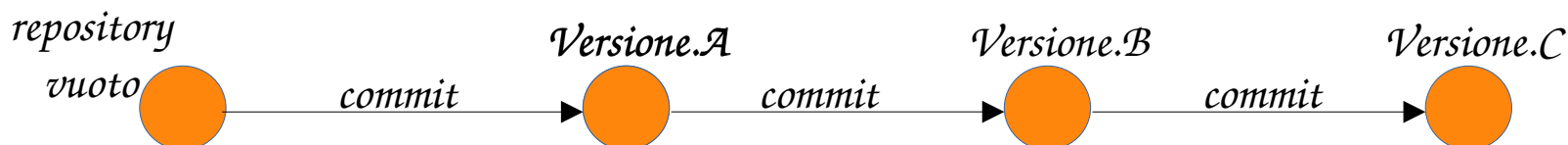


- git tiene traccia di tutti commit con:

- un ID unico
- un timestamp
- l'autore



- Github ci consente di consultare queste informazioni da Web, comodamente inseguendo qualche link
- I commit possono pertanto organizzarsi in grafi
  - Per il momento piuttosto banali (lineari)



- In generale possono essere piuttosto complicati (>>)



# Repository: Locale vs Remoto

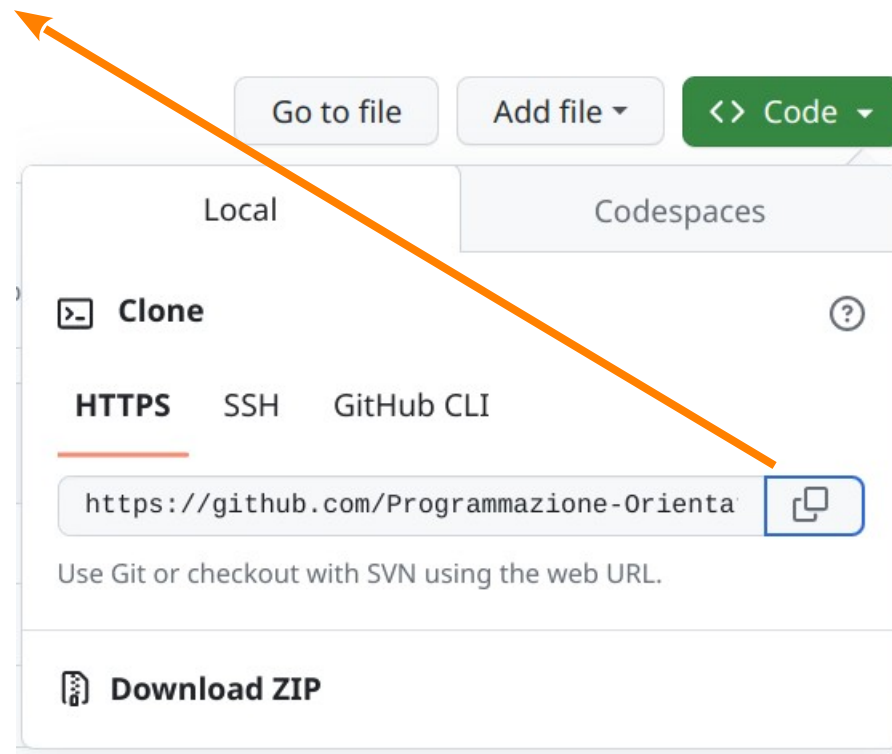
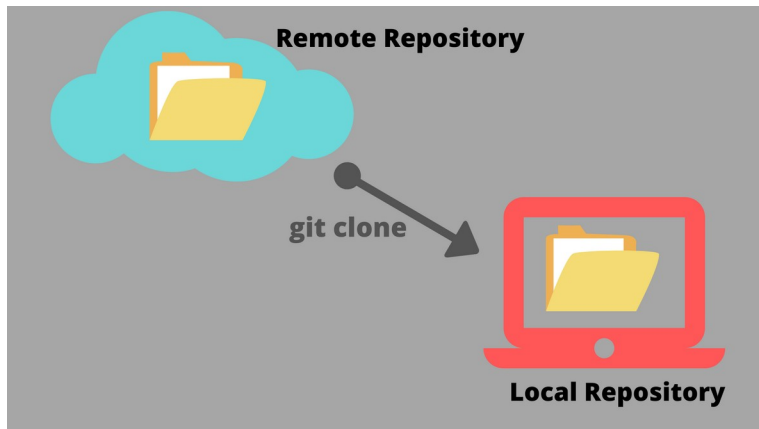
- In realtà `git` (l'applicazione che Github ospita) è installabile anche sulla propria macchina e permette di creare e di lavorare anche su repository locali
  - vi si può accedere da riga di comando od anche attraverso interfacce grafiche come quelle offerte da alcuni plugin di Eclipse
- Le potenzialità di `git` + Github si “sprigionano” completamente solo quando al repository remoto che abbiamo già creato ogni collaboratore affianca un repository locale su cui fare le proprie modifiche
- ✓ Idea principale:
  - le modifiche si fanno su un proprio repository localmente
  - si condividono poi sincronizzando il repository locale con il repository remoto
    - Ovvero caricando all'interno del repository remoto i commit fatti nel proprio repository locale
    - Le proprie «evoluzioni»/«refactoring» del codice saranno così visibili anche agli altri collaboratori

# Github & git clone

- Una volta installato sulla propria macchina git, il modo più immediato di creare un repository locale è quello di «clonare» il repository remoto
- Eseguire:

```
git clone https://github.com/Programmazione-Orientata-agli-Oggetti/diadia.git
```

- Sarà creata (in locale all'interno della cartella in cui si è eseguito il comando) la cartella diadia con tutto il codice presente nel repository remoto
- E' il vostro repository «locale»



# Github & git pull

- Da questo punto in poi ogni collaboratore può trasferire le modifiche caricate sul repository remoto direttamente nel proprio repository ("repo") locale
  - **cd diadia** (per posizionarsi all'interno della cartella diadia)
  - **git pull** (per scaricare dal repo remoto i commit ed applicare le stesse modifiche al repo locale)
- Quando un collaboratore intende modificare il codice, anzitutto deve scaricare in locale le modifiche che sono nel frammento state caricate dentro il repo remoto
  - Questo per ottenere l'ultima versione del codice
- Quindi effettua le modifiche desiderate nel repo locale
- Poi, per condividerle, deve fare un commit sul repo remoto
  - per il momento anche caricando il nuovo codice tramite Github e fare il commit sul repository remoto da Web

(Attenzione che questo richiede di ripetere **git pull** per recepire nel proprio repo locale anche il commit appena fatto sul repo remoto)
- Vedremo come farlo direttamente da riga di comando (>>)

# Esercizio 0

- Chi non avesse concluso la scrittura dei test, lo faccia in questo homework, prima di fare le modifiche al codice
- E' importante partire da una versione del codice funzionante, con un sufficiente numero di test per garantirlo

# Esercizio 1

- Sostituire tutti gli array utilizzati nelle classi **Stanza** e **Borsa** con opportune collezioni (**List**, **Set**, **Map**)
  - ✓ Assumere che non possano esistere due oggetti **Attrezzo** con lo stesso nome in stanze dello stesso **Labirinto**
    - Eliminare il vincolo che al max 10 attrezzi possano essere collocati nella borsa (ma mantenere quello sul peso max)
    - Provare ad usare (in alternativa) **List** e **Map** per implementare la collezione di attrezzi nella borsa. Quale risulta più semplice, e perché?
- ✓ Queste modifiche alterano l'implementazione ma non la logica: dopo averle effettuate confermare con i test sviluppati nei precedenti homework la correttezza del codice *anche* dopo le nuove modifiche

# Esercizio 2

- Rivisitare il codice delle nuove tipologie di stanze introdotte nel precedente homework:
  - La stanza *buia*: se nella stanza non è presente un attrezzo con un nome particolare (ad esempio "lanterna") il metodo **getDescrizione()** di una stanza buia ritorna la stringa "*qui c'è un buio pesto*"
  - La stanza *bloccata*: una delle direzioni della stanza non può essere seguita a meno che nella stanza corrente non sia presente un oggetto di un certo nome (ad es. «piedediporco»)
- Queste sottoclassi subiscono modifiche consequenziali ai cambiamenti effettuati nella implementazione della loro superclasse **Stanza**?
- ✓ Le modifiche che subiscono le due versioni di **Stanza** (*con e senza* campi **protected**) ipotizzate nel precedente homework, sono le stesse?

# Esercizio 2 (cont.)

- Ampliare i test di tutte le classi della gerarchia che ha radice in **Stanza**:
  - **Stanza, StanzaMagica, StanzaBuia e StanzaBloccata**
- Eliminare dal codice delle classi **Borsa, Stanza, StanzaMagica, StanzaBuia e StanzaBloccata** (estensioni di **Stanza**) *OGNI* ciclo di ricerca lineare da una collezione (ad es. di un attrezzo per nome, o di una stanza per direzione)
- ✓ affidarsi invece sempre e solo alle funzionalità offerte dai metodi già offerti nelle classi del JCF
  - ad es. quelli di ricerca in una collezione

# TDD (Facoltativo)

- ✓ N.B. È perfettamente lecito e consigliabile fare l'esercizio 5 anche prima degli esercizi 3&4



# Esercizio 3

- Aggiungere alla classe **Borsa** dei metodi di interrogazione del suo contenuto:
  - `List<Attrezzo> getContenutoOrdinatoPerPeso();`  
restituisce la lista degli attrezzi nella borsa ordinati per peso e quindi, a parità di peso, per nome
  - `SortedSet<Attrezzo> getContenutoOrdinatoPerNome();`  
restituisce l'insieme degli attrezzi nella borsa ordinati per nome
  - `Map<Integer, Set<Attrezzo>> getContenutoRaggruppatoPerPeso();`  
restituisce una mappa che associa un intero (rappresentante un peso) con l'insieme (comunque *non* vuoto) degli attrezzi di tale peso: tutti gli attrezzi dell'insieme che figura come valore hanno lo stesso peso pari all'intero che figura come chiave
- Utilizzare questi metodi per migliorare la stampa del contenuto della **Borsa** (ad es. aggiungere e/o modificare un comando *guarda* per la stampa del suo contenuto>>)

# Esercizio 3 (Notazione Es.)

- Si utilizzi *piombo:10* per indicare un riferimento ad un oggetto **Attrezzo** di nome “piombo” e peso 10
- Per brevità scriviamo *piombo* al posto di *piombo:10* quando non è utile ripetere il dettaglio sul peso
- Si utilizzi quindi:
  - $\{ \textit{piombo}, \textit{piuma}, \textit{libro}, \textit{ps} \}$  per indicare un **Set** di attrezzi
  - $[ \textit{piuma}, \textit{libro}, \textit{ps}, \textit{piombo} ]$  per indicare una **List** di attrezzi
  - $( 5, \{ \textit{libro}, \textit{ps} \} )$  per indicare una coppia chiave/valore di una **Map**`<Integer, Set<Attrezzi>>`

# Esercizio 3 (Esempio)

- Si consideri una **Borsa** contenente questo insieme di riferimenti ad oggetti **Attrezzo**:  
*{ piombo:10, ps:5, piuma:1, libro:5 }*
- Allora i metodi di cui prima, invocati sullo questa **Borsa**:
  - **List<Attrezzo> getContenutoOrdinatoPerPeso()**  
deve restituire: *[ piuma, libro, ps, piombo ]*
  - **SortedSet<Attrezzo> getContenutoOrdinatoPerNome()**  
deve restituire: *{ libro, piombo, piuma, ps }*
  - **Map<Integer, Set<Attrezzo>> getContenutoRaggruppatoPerPeso()**  
deve restituire una **Map** contenente tutte e sole le coppie:  
*( 1, { piuma } ) ; ( 5, { libro, ps } ) ; ( 10, { piombo } )*

# Esercizio 4

- Aggiungere alla classe **Borsa** un nuovo metodo
  - `SortedSet<Attrezzo> getSortedSetOrdinatoPerPeso();`  
restituisce l'insieme gli attrezzi nella borsa ordinati per peso e quindi, a parità di peso, per nome
- ✓ Scrivere un test per verificare che due attrezzi di stesso peso ma nome diverso rimangano distinti nel risultato

# Esercizio 5

- Utilizzando JUnit, scrivere una batteria di test-case *minimali* per verificare la correttezza delle soluzioni prodotte negli esercizi 3&4 precedente
  - *minimali*: ovvero facenti utilizzo delle collezioni più semplici possibile utili alla verifica (piccole e con **Attrezzi** di nomi/pesi in configurazioni *a loro volta minimali* )
  - ✓ N.B. È perfettamente lecito e consigliabile fare questo esercizio anche prima degli esercizi 3&4
- Solo dopo aver completato il precedente punto, valutare se ampliare i test-case di sopra con altri test-case non minimali, per migliorarne la copertura

# Partite Testabili

- Cambiare il codice di **DiaDia** affinché supporti la creazione di una **Partita** da svolgersi in un certo **Labirinto**
  - Ovvero, aggiungere il costruttore: **DiaDia(Labirinto, io)**
- Ad esempio:

```
public class DiaDia {  
    ...  
    public static void main(String[] argc) {  
        /* N.B. unica istanza di IOConsole  
           di cui sia ammessa la creazione */  
        IO io = new IOConsole();  
        Labirinto labirinto = new LabirintoBuilder()  
            .addStanzaIniziale("LabCampusOne")  
            .addStanzaVincente("Biblioteca")  
            .addAdiacenza("LabCampusOne", "Biblioteca", "ovest")  
            .getLabirinto();  
  
        DiaDia gioco = new DiaDia(labirinto, io);  
        gioco.gioca();  
    }  
    ...}
```

# Esercizio 6

- Supportare partite svolte in labirinti diversi
  - deve essere possibile aggiungere e conservare il riferimento all'oggetto **Labirinto** in cui si svolge una partita direttamente dentro un oggetto **Partita** con il costruttore **Partita(Labirinto)** *ed anche* con il metodo **Partita.setLabirinto(Labirinto)**
- Per facilitare la costruzione di questi oggetti **Labirinto**, si aggiunga la classe **LabirintoBuilder**
- **LabirintoBuilder**: Classe dedicata esclusivamente alla creazione di oggetti **Labirinto** utilizzando una tecnica (*method-chaining*) che faciliti la costruzione incrementale di un oggetto il cui stato è composito
  - formato da molte informazioni, ovvero (per **Labirinto**): stanze, adiacenze, stanza iniziale, stanza vincente, attrezzi, ecc. ecc.

# Esercizio 6 (continua)

- Per chiarire cosa deve fare questa classe **LabirintoBuilder**, e come deve essere utilizzato l'insieme dei metodi che offre, e la loro semantica, si consideri il codice esemplificativo del suo utilizzo mostrato di seguito:

- ```
Labirinto monolocale = new LabirintoBuilder()
    .addStanzaIniziale("salotto") // aggiunge una stanza, che sarà anche iniziale
    .addStanzaVincente("salotto") // specifica quale stanza sarà vincente
    .getLabirinto(); // restituisce il Labirinto così specificato
```

```
Labirinto bilocale = new LabirintoBuilder()
    .addStanzaIniziale("salotto")
    .addStanzaVincente("camera")
    .addAttrezzo("letto",10) // dove? fa riferimento all'ultima stanza aggiunta: la "camera"
    .addAdiacenza("salotto", "camera", "nord") // camera si trova a nord di salotto
    .getLabirinto(); // restituisce il Labirinto così specificato
```

```
Labirinto trilocale = new LabirintoBuilder()
    .addStanzaIniziale("salotto")
    .addStanza("cucina")
    .addAttrezzo("pentola",1) // dove? fa riferimento all'ultima stanza aggiunta: la "cucina"
    .addStanzaVincente("camera")
    .addAdiacenza("salotto", "cucina", "nord")
    .addAdiacenza("cucina", "camera", "est")
    .getLabirinto(); // restituisce il Labirinto così specificato
```



# Esercizio 6 (continua)

- Implementare **LabirintoBuilder** con la tecnica del method-chaining
- ✓ *Method-chaining*: i metodi **addXYZ(...)** di **LabirintoBuilder** sono tutti implementati secondo questo schema:

```
public LabirintoBuilder addXYZ(...) {  
    // tiene traccia  
    return this;  
}
```
- Scrivere dei test sulla classe per individuare e correggere gli errori di **LabirintoBuilder**
- Completare le funzionalità rispetto alle sole esemplificate per gestire la specifica
  - della stanza di ingresso e di uscita del labirinto
  - le adiacenze (uscite) che collegano le stanze
  - la collocazione degli attrezzi
  - i diversi tipi di stanza
  - ecc. ecc.
- ✓ Per quanto segue, meglio verificare se la propria implementazione di **LabirintoBuilder** soddisfa anche alcuni test di unità di riferimento che si possono trovare a [questo indirizzo su github](#)

# Esercizio 7

## (IOSimulator con JCF)

- Rimuovere ogni riferimento agli array anche nella classe **IOSimulator**
  - Valutare in alternativa l'utilizzo di **List** o **Map**
  - ✓ E se volessimo ricordare, per ogni riga letta, i corrispondenti messaggi prodotti?
- Scrivere/ampliare i test per simulare *interi* partite e non solo *singoli* metodi
  - Iniziare con partite semplici
    - Obiettivo: chiarire le regole di *dominio*
    - Ovvero, come funziona il gioco diadia
  - Aumentare la complessità creando test che comprendono più comandi in fila
  - Se necessario creare ogni volta labirinti ad hoc tramite l'utilizzo di **LabirintoBuilder.getLabirinto()**
- ✓ *Attenzione*: questi NON sono affatto test di unità

# Esercizio 8

- Una volta appurato il corretto funzionamento di **LabirintoBuilder** (esercizio 6) rivedere ed ampliare i test già scritti su altre classi
  - ad esempio **ComandoVai**
- Sfruttare la nuova classe per creare test-case con fixture di complessità crescenti: labirinto «monolocale», «bilocale», ecc.

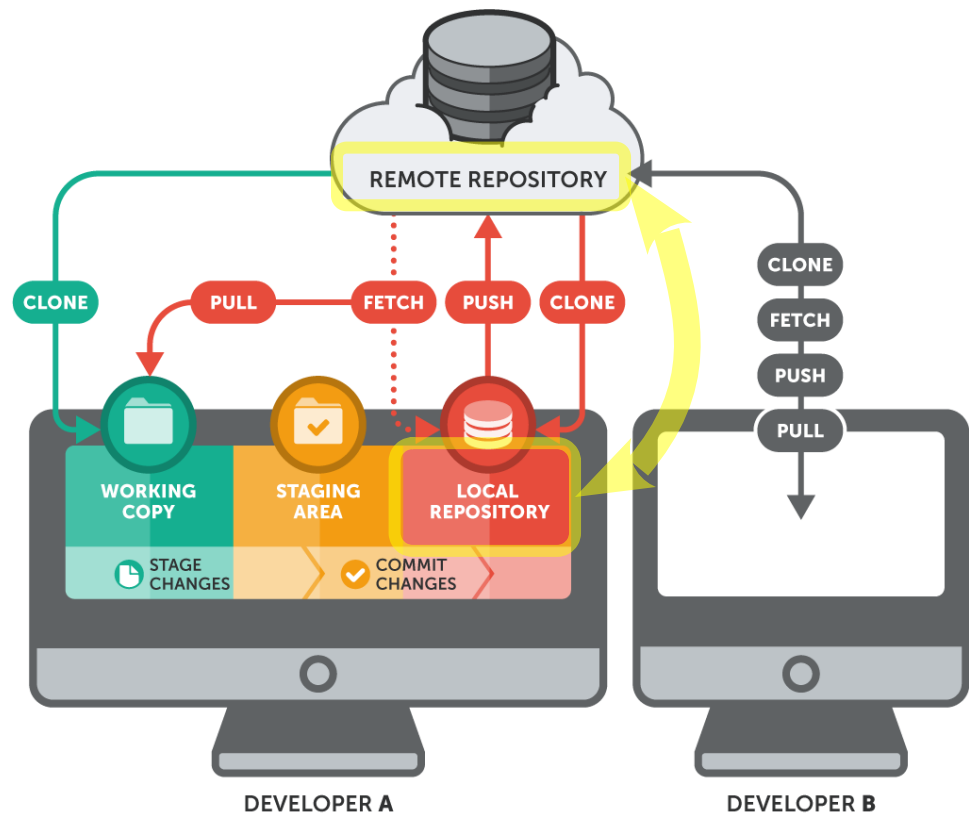
# Esercizio 9 (Facoltativo) : Cominciare ad Usare “Seriamente” git & Github

- Da questo esercizio in poi, cercate di simulare un ciclo di sviluppo semi-professionale con il vostro compagno
- Ogni refactoring/esercizio che segue deve essere associato ad una pull-request dedicata approvata dal vostro compagno
- Alternatevi nel ruolo di chi crea la PR e chi invece la revisiona e l'approva

# Versionamento del Codice

## Git Repository: Locale vs Remoto

- `git` (l'applicazione che lo stesso Github ospita per versionare il codice) è installabile anche localmente
- permette di creare e lavorare anche su repository «locali»
  - vi si può accedere da riga di comando utilizzando il comando `git` od anche attraverso plugin dedicati di Eclipse, tra tutti: *EGit*
  - Il plugin di Eclipse EGit viene già fornito con la versione della piattaforma *Eclipse for Java Developers* consigliata all'inizio di questo corso
  - Dovreste già averlo!
- ad un repository remoto creato su Github, ogni collaboratore affianca un repository locale su cui fare le proprie modifiche



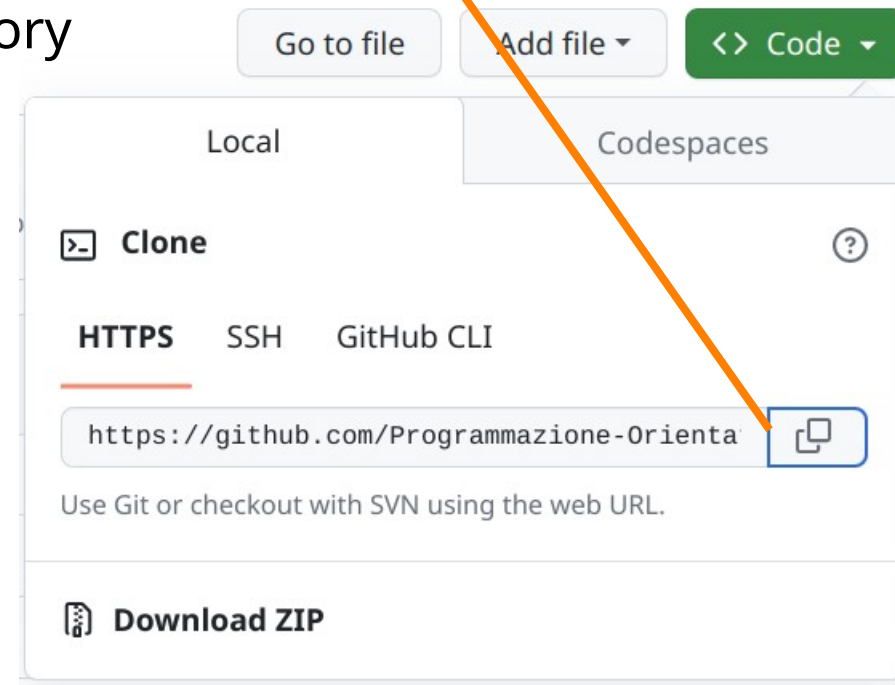
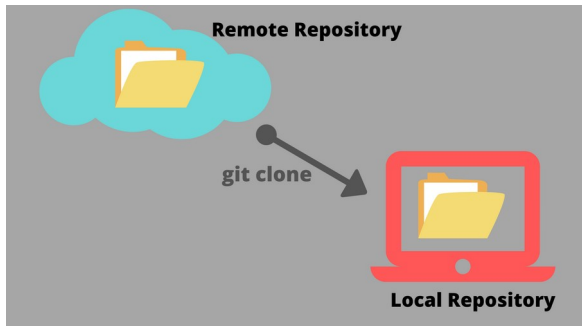
# Github & git clone

- Installato **git** sulla propria macchina, il modo più immediato di creare in locale un repository è quello consistente nel «clonare» il corrispondente repository remoto
- Eseguire: **git clone <url-del-vostro-repository>**
- Ad es. per creare una copia locale del repository su cui abbiamo caricato la versione base del codice dello studio di caso diadia

```
git clone https://github.com/Programmazione-Orientata-agli-Oggetti/diadia.git
```

- ✓ Usare invece l'URL del *vostro* repository remoto:

- Sarà creata (in locale all'interno della cartella in cui si è stato eseguito il comando) la cartella diadia con una copia di tutto il codice

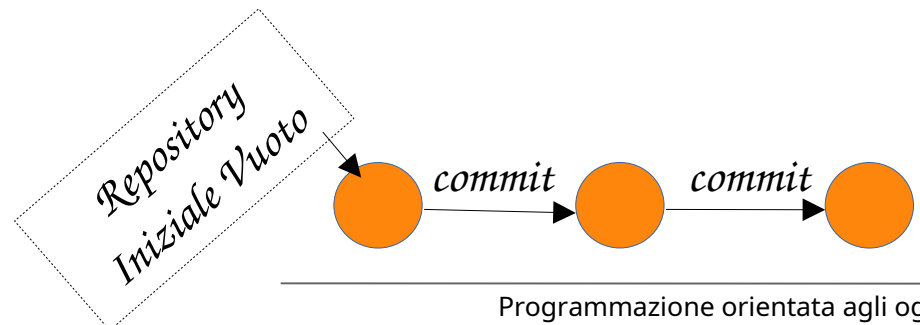


# Condivisione del Codice Tramite Git Repository: Locale vs Remoto

- ✓ Idea principale: Le modifiche si fanno su un proprio repository localmente
  - Vedremo tra poco come organizzare e “impacchettare” una serie di modifiche in uno o più «commit» (>>)
- Si condivide il proprio lavoro copiando tali modifiche dal repository locale all'interno del repository remoto condiviso
  - Caricando all'interno del repository remoto i commit fatti nel proprio repository locale
  - Il proprio lavoro (i propri commit sul codice) sarà così visibile anche agli altri collaboratori che possono scaricarlo sulle proprie macchine
  - Sicuramente le modifiche caricate da altri sono accessibili direttamente anche tramite il portale Github che ospita il repository remoto: quest'ultimo finisce per essere considerato il punto di riferimento condiviso tra tutti i collaboratori
- Tuttavia conviene praticamente sempre scaricare le ultime modifiche dal repository remoto nel proprio repository locale sulla propria macchina prima di lavorarci
  - comodamente con Eclipse come sempre fatto per tutti i progetti (anche non condivisi)

# Commit (1)

- Per comunicare ad altri e condividere le modifiche operate su una versione del progetto, è necessario impacchettare queste modifiche dentro un «commit»
- Perché raggruppare le modifiche di molti file in un solo commit?
- Si pensi ad un refactoring→rename di un metodo Java:
  - ✓ bisogna modificare sia la definizione del metodo sia tutte le sue invocazioni: soprattutto per metodi pubblici, i contenuti di diverse classi/file finiscono per essere alterati da un singolo refactoring
- Per portare il progetto da una versione compiuta ad un'altra altrettanto «compiuta» è necessario operare tutte le modifiche ai file coinvolti e non solo una parte
  - Molta confusione scaturirebbe dalla coesistenza in un repository del progetto di diversi nomi (prima e dopo la ridenominazione) dello stesso metodo





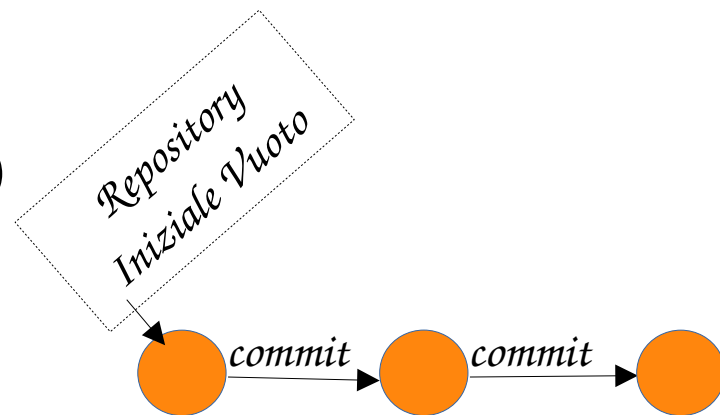
# Commit (2)

- Ciascun commit include *almeno* queste informazioni; anzitutto:

- Quali file sono stati modificati
- Le modifiche operate per ciascuno
- Un identificatore unico (*hash* del commit)
  - Ad es. **9bf4df831c8089dc805fcac177ba13b2d8c93d50**

ed inoltre:

- L'autore
  - Il timestamp
  - Un messaggio descrittivo
- Vien da sé che una «versione» (ad esempio quella corrente) del progetto si può ottenere applicando la sequenza di modifiche operate da tutti i commit a partire dal repository vuoto dello stesso
  - Per questo motivo talvolta il termine commit viene anche utilizzato per indicare una certa versione del progetto:
    - dato un commit, risultano univocamente determinati tutti i file ed i loro contenuti del progetto risultanti dopo quel commit



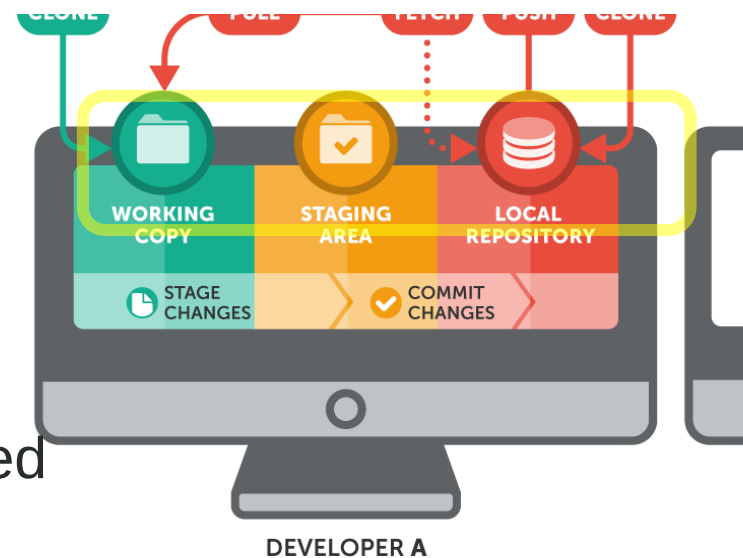
# Commit con git (1)



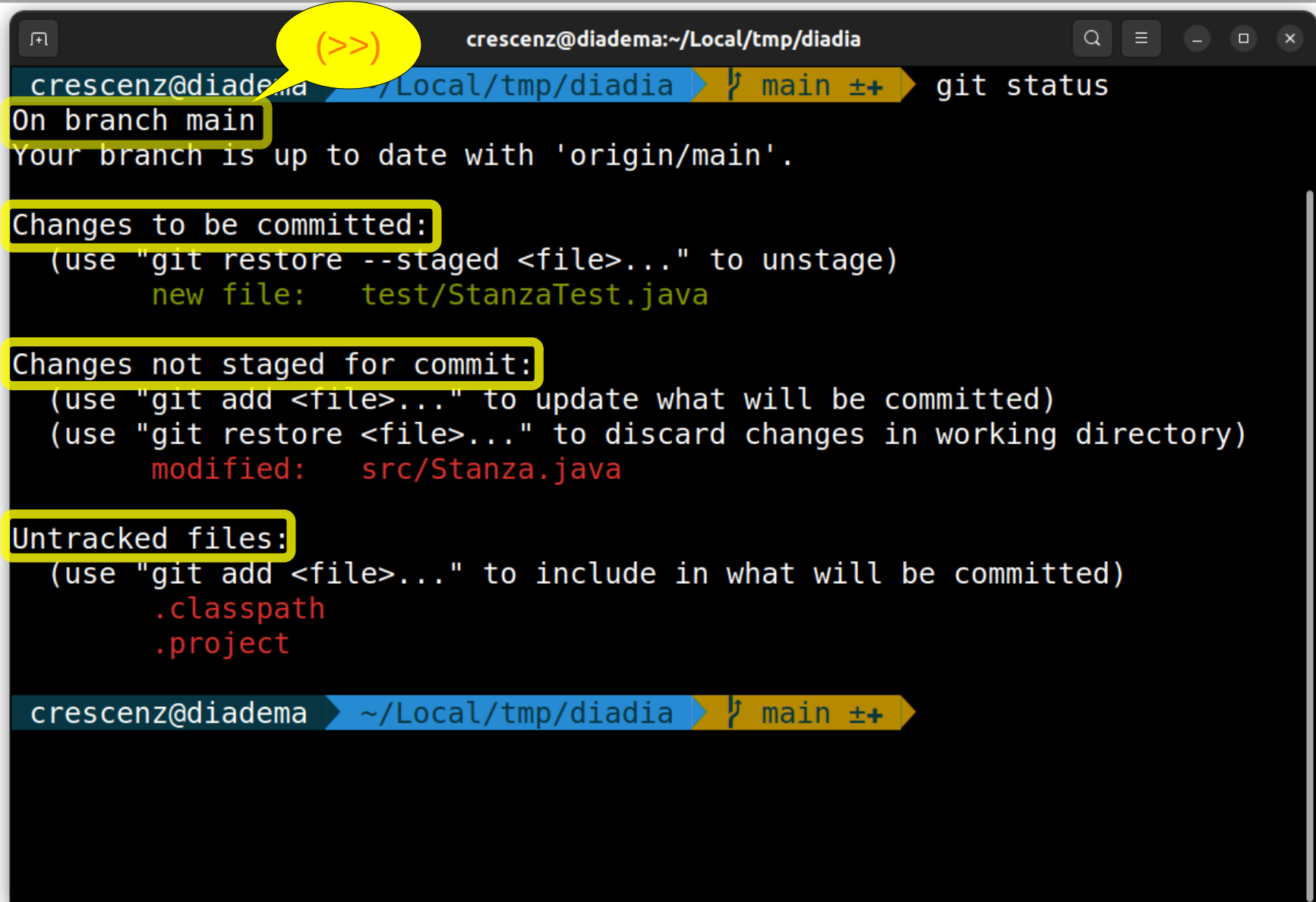
- Specificare i contenuti di un commit significa stabilire
  - quali e quanti file ne fanno parte
  - quali modifiche operare per ciascuno dei file coinvolti
- **git** offre pertanto una serie di comandi (con molte opzioni, su cui non insisteremo) per gestire la «specificità» di un commit
- Principali caratteristiche
  - *gestione incrementale*: si possono aggiungere le modifiche anche di un file alla volta a quello che sarà il prossimo commit
  - i file “migrano”, in un certo senso, dalla propria copia dei file del repository nel file system locale («working copy») verso il «repository locale» passando per un’area temporanea in cui vengono “parcheeggiati” in attesa del commit finale («staging area»):  
«working copy» → «staging area» → local repository
  - *concetto di «stato» di un file*
    - *Modified*: già presente nel repository locale ma modificato nel file system rispetto all’ultima versione ivi presente
    - *Committed*: file presente nel repository locale e nel file system; il file risulta anche allineato all’ultima versione presente nel repository
    - *Staged*: si sono “calcolate” le modifiche necessarie per allineare il file presente nel file-system alla versione più recente nel repository
      - Le modifiche sono pronte ad essere parte di un commit ancora da creare
    - *Untracked*: presente nel file system ma mai considerato parte del repository locale

# Commit con git (2)

- Nella sostanza git permette di collezionare tutte le modifiche che si vogliono aggregare in uno stesso commit portando ciascuno dei file coinvolti nello stato di *staged*
- Anche incrementalmente con una serie di comandi di git successivi
  - `git add src/it/uniroma3/diadia/DiaDia.java`
  - `git add test/it/uniroma3/diadia/DiaDiaTest.java`
  - ...
- Per vedere lo stato dei file
  - `git status`
- Solo quando tutti i file che si desidera inserire nel medesimo commit sono staged si può completare il commit
  - `git commit -m 'descrizione del commit'`



# Commit con git (3)



A terminal window titled "crescenz@diadema:~/Local/tmp/diadia" showing the output of the "git status" command. The terminal has a dark background with light blue and yellow highlights. A yellow speech bubble with "(>>)" points to the first line of the output. Several lines of the output are highlighted with yellow boxes. The output text is as follows:

```
crescenz@diadema ~/Local/tmp/diadia main ±+ git status
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   test/StanzaTest.java

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   src/Stanza.java

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    .classpath
    .project

crescenz@diadema ~/Local/tmp/diadia main ±+
```

# Commit con EGit (1)

- Il plugin di Eclipse ( EGit ), rende questi passaggi ancora più semplici, perché la scelta dei file da inserire nel commit avviene con l'ausilio di una GUI
  - ✓ Attenzione: questo non scalfisce minimamente la necessità di capire i concetti su cui git si basa come lo stato dei file
    - Consente di non doversi ricordare alcuni comandi git e relative opzioni
- L'analisi delle differenze rispetto di un file .java rispetto ad una precedente versione (spesso la più recente, ovvero oggetto dell'ultimo commit che ha riguardato tale file) risulta molto più agevole
- ✓ Didatticamente può essere forse utile abituarsi ad usare EGit solo dopo saper padroneggiare almeno i comandi di base di git da linea di comando

# Commit con EGit (2)

didattica - Compare /diadia-git/src/Stanza.java Current and Index - Eclipse IDE

File Edit Navigate Search Project Run Window Help

DiaDia.java Attrezzo.java Stanza.java x StanzaTest.java Compare Stanza.java Current and Index x

Java Structure Compare

Compilation Unit

Stanza

- toString()

Java Source Compare

Local: Stanza.java

```
130 risultato.append("\nAttrezzi nella stanza: ");
131 boolean nessunAttrezzo = false;
132 for (int i=0; i<this.numeroAttrezzi; i++) {
133     final Attrezzo attrezzo = this.attrezzi[i];
134     if(attrezzo!=null) {
135         nessunAttrezzo = true;
136         risultato.append(attrezzo.toString()+" ");
137     }
138 }
139 if(!nessunAttrezzo)
140     risultato.append("nessun attrezzo");
141
142 risultato.append("\n");
143 return risultato.toString();
```

Index: Stanza.java (editable)

```
130 risultato.append("\nAttrezzi nella stanza: ");
131 boolean nessunAttrezzo = false;
132 for (Attrezzo attrezzo : this.attrezzi) {
133     if(attrezzo!=null) {
134         nessunAttrezzo = true;
135         risultato.append(attrezzo.toString()+" ");
136     }
137 }
138 if(!nessunAttrezzo)
139     risultato.append("nessun attrezzo");
140
141 risultato.append("\n");
142 return risultato.toString();
143 }
```

History Synchronize Git Staging x Git Reflog Properties

Filter files

> diadia [main]

Unstaged Changes (1/3)

- > Stanza.java - src

Staged Changes (1)

- StanzaTest.java - test

Commit Message

Bug fix (NPE): stampa solo this.numeroAttrezzi attrezzi

Con la rappresentazione compatta di una lista tramite array solo i primi this.numeroAttrezzi dell'array this.attrezzi sono effettivamente utilizzati (e non quindi null)

Author: <crescenz@dia.uniroma3.it>

Committer: <crescenz@dia.uniroma3.it>

Commit and Push Commit

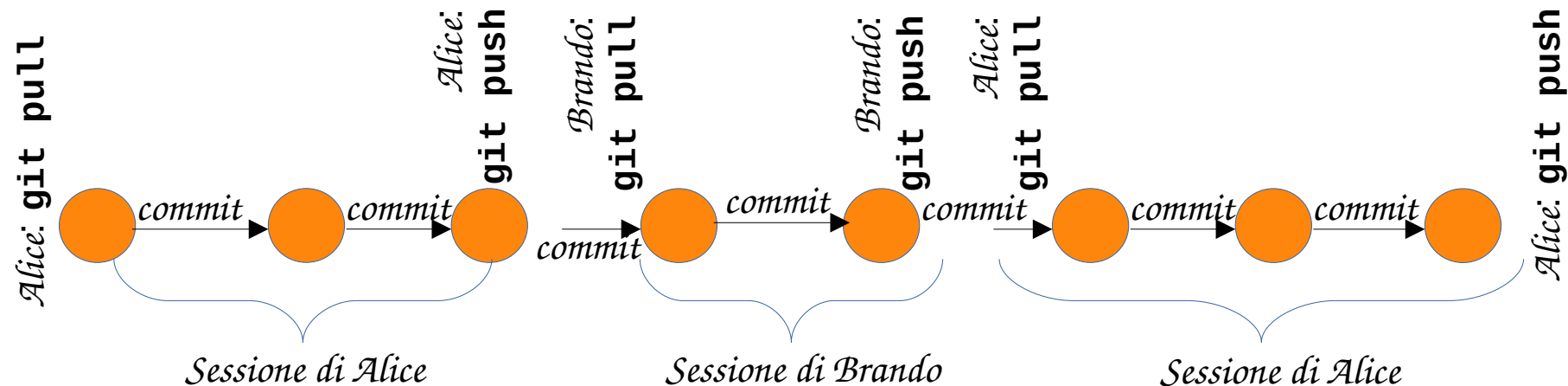
# Versionamento del Codice

## Git Repository: Caso Ideale (1)

- Abbiamo chiarito come *definire & creare* commit
  - ✓ ora usiamoli per condividere le modifiche tra sviluppatori e spostarle tra diversi repository (remoti e locali)
- Supponiamo (per il momento) di vivere in un semplificato mondo ideale con interazioni molto “stilizzate”
  - Sviluppatori perfettamente disciplinati collaborano assieme
  - Anche a costo di rendere impraticabilmente lento lo sviluppo, i collaboratori organizzano il flusso di lavoro in «sessioni seriali»
    - Al più un solo sviluppatore alla volta lavora sul progetto
    - Nessuno può lavorarci sino a quando un altro non termina e pubblica i commit risultanti dalla sua ultima sessione di lavoro
  - In questo scenario basta conoscere i pochi comandi git necessari per per trasferire bidirezionalmente i commit:
    - Inizio della sessione di lavoro:  
**git pull** (per «scaricare» dal repo remoto in quello locale i commit ed applicare le stesse modifiche al repo locale)
    - Fine della sessione di lavoro:  
**git push** (per «caricare» dal repo locale in quello remoto i commit locali ed applicare le stesse modifiche al repo remoto)

# Versionamento del Codice

## Git Repository: Caso Ideale (2)

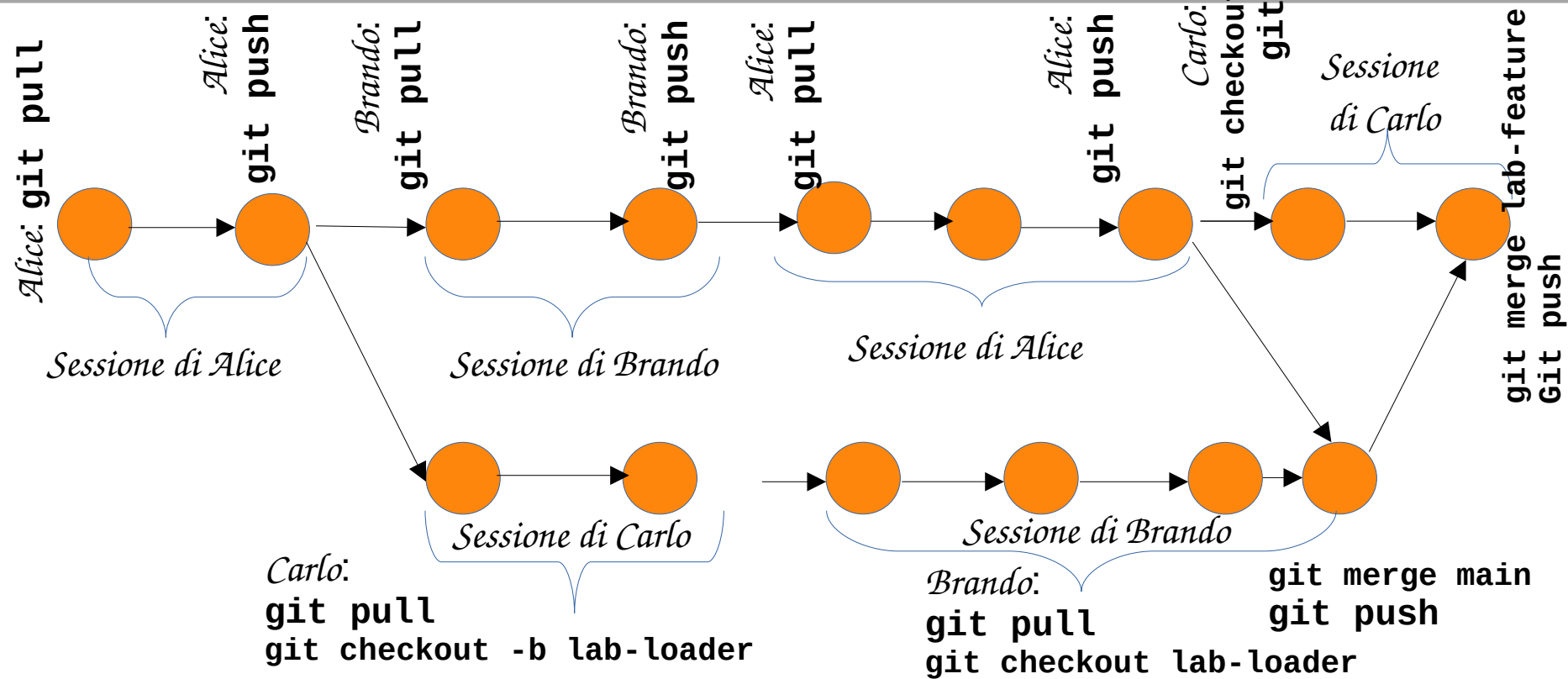


- In aggiunta gli sviluppatori (*Alice* Matricola 612345 e *Brando* Matricola 654321) sono invitati a parcellizzare le modifiche effettuate in ogni propria sessione di modo da renderle «compiute» prima di condividerle
- Dovendo altri cominciare da dove si è deciso di terminare la propria sessione, sarebbe meglio che
  - l'insieme dei commit della propria sessione sia tale da apportare miglioramenti auto-contenuti e quindi meglio comprensibili ad altri
    - ✓ meglio ancora se con tanto di test di unità funzionanti
  - secondo questo modello, potrebbe convenire rinunciare alla condivisione pur di non bloccare gli altri sviluppatori in attesa di partecipare



# Versionamento del Codice

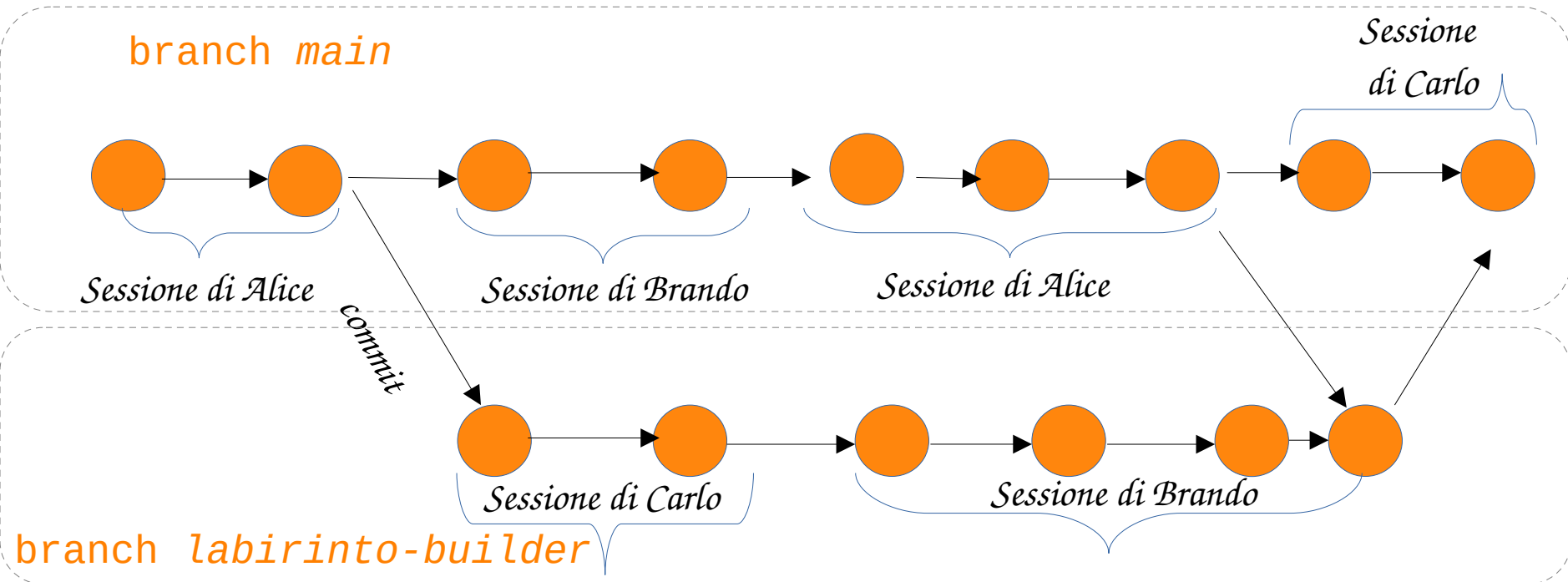
## Git Repository: Caso Reale (1)



- Nella realtà nessuna di queste ipotesi è vera, e può succedere, e succede, di tutto...
- Partendo dall'ovvio, se pensiamo al contenimento dei tempi di sviluppo:
  - ✓ Diversi sviluppatori possono organizzare il proprio lavoro in parallelo possibilmente su sviluppi quanto più possibile indipendenti
  - ✓ Alcuni preferiscono condividere anche solo bozze del codice

# Versionamento del Codice

## Git Repository: Caso Reale (2)



- Per questo risulta necessario
  - saper «sdoppiare» le linee di sviluppo («branch»)
  - saper «fondere» e riconciliare le linee di sviluppo
    - ✓ gestendo eventuali «conflitti» (modifiche in branch diversi agli stessi file)
- git permette di creare e manipolare qualsiasi grafo aciclico di commit
  - ✓ e si possono trovare praticamente sempre ottimi casi reali per conoscere ciascuno di tutti i suoi comandi con relative varianti

# Versionamento del Codice

## Git Repository: Caso Reale (3)

- Fermiamoci qui: Obiettivi formativi --- introduzione a Github & git
- Conclusioni: Collaborare in un team di sviluppo su codice condiviso non è affatto banale; il problema, nella sua vera complessità non è di facile risoluzione, nemmeno con gli strumenti giusti (git)
  - esperienza di programmazione in team: necessaria ed insostituibile: collaborate nello sviluppo degli HW(!)
  - ✓ Se non ci provate, non potete nemmeno immaginare quali problemi esistano
- Alcuni comandi per muovere i primi passi
  - ✓ git status
  - ✓ git log
  - ✓ git add [-A]
  - ✓ git diff
  - ✓ git commit [-m]
  - ✓ git push
  - ✓ git pull
- Alcuni comandi che bisogna conoscere per un uso di base:
  - x git checkout [-b]
  - x git merge
  - x git branch [-m]

➤ Uso avanzato: tutto il resto

# Esercizio 10

## (FabbricaComandiRiflessiva)

- Introdurre ed utilizzare la fabbrica di comandi basato sull'uso delle API per l'introspezione per la creazione degli oggetti **Comando**
- Ricontrollare tutto per assicurarsi che finalmente l'elenco dei comandi disponibili nel gioco sia effettivamente specificato una sola volta in tutto il codice ed il disaccoppiamento completo
- ✓ A riprova: provare ad aggiungere un nuovo comando e controllare quali e quante modifiche risultano necessarie
  - Ad es. **ComandoRegala** dell'Esercizio 5 che segue (>>)
- Rendere il nuovo comando *perfettamente* funzionante ed integrato con il resto del gioco ...

# Esercizio 11 (Facoltativo):

## ComandoAiuto Introspeettivo/usa Javadoc

- In particolare il nuovo comando deve figurare automaticamente nell'elenco stampato dal comando **aiuto**
- Ricercando i javadoc delle API relative all'introspezione (package **java.lang.reflect.\***) utilizzare l'introspezione per rifattorizzare l'implementazione **ComandoAiuto**
  - ✓ Questa deve recuperare da sola l'elenco dei nomi dei comandi disponibili nel gioco direttamente dal nome delle classi che li implementano assumendo che:
    - si trovino tutte nello stesso noto package
    - abbiano tutte un nome con il prefisso **Comando**
- Ricontrollare tutto per assicurarsi che finalmente l'elenco dei comandi disponibili nel gioco sia effettivamente specificato una sola volta in tutto il codice (nel nome delle classi stesse) ed il disaccoppiamento *completo*

# Esercizio 12 (AbstractComando)

- Scrivere una classe astratta **AbstractComando** per eliminare le implementazioni “vuote” dei metodi **setParametro()** dalle classi concrete che implementano l'interfaccia **Comando** (in particolare dalle classi che modellano comandi privi di parametri come ad es. **ComandoGuarda**, **ComandoAiuto**)
- Scrivere nuovi test per la nuova classe astratta e rifattorizzare i test della gerarchia **Comando** già sviluppati durante lo svolgimento dei precedenti homework

# Esercizio 13 (AbstractPersonaggio)

- Scrivere la classe astratta **AbstractPersonaggio** e le classi concrete che la estendono
  - **Strega**
  - **Mago**
  - **Cane**
- Scrivere le classi **ComandoSaluta** e **ComandoInteragisci** che modellano i comandi attraverso i quali il giocatore può rispettivamente salutare e interagire con un personaggio
- Queste modifiche sono descritte anche nelle trasparenze reperibili nella pagina del materiale didattico del corso:

**P00-classi-astratte-enum**

# Esercizio 14 (ComandoRegala)

- Modificare la classe astratta **AbstractPersonaggio** introducendo il metodo astratto:

```
public String riceviRegalo(Attrezzo attrezzo, Partita partita)
```

- Scrivere la classe **ComandoRegala**, attraverso la quale il giocatore può regalare un attrezzo al personaggio presente nella stanza
  - ✓ in una stanza può trovarsi un solo personaggio: affinché un attrezzo possa essere regalato il parametro del comando *regala* deve essere il nome di uno degli attrezzi presenti nella borsa



# Esercizio 14 (cont.)

- Nelle classi **Cane**, **Strega**, **Mago** implementare il metodo astratto:

**public String riceviRegalo(Attrezzo attrezzo)**

- un cane riceve un regalo: accetta il suo cibo preferito, e butta a terra un attrezzo; ma morde e toglie un CFU per tutto il resto
  - una strega riceve un regalo, che trattiene scoppiando a ridere
  - un mago riceve un regalo, gli dimezza il peso e lo lascia cadere nella stanza
- La stringa restituita rappresenta il messaggio che deve essere prodotto dal comando quando eseguito (analogamente al comando *interagisci*)

# Esercizio 15 (CaricatoreLabirinto)

- Modificare la classe **Labirinto** affinché la specifica del labirinto venga letta da file testuale utilizzando la classe **CaricatoreLabirinto**
  - ✓ Una *bozza* è fornita nella pagina del materiale didattico
- Va modificata e rifattorizzata per due ottimi motivi:
  - dovrà, a sua volta, avvalersi di **LabirintoBuilder**
  - contiene degli errori

# Esercizio 15 (cont.)

- Scrivere dei test di unità per correggere gli errori di **CaricatoreLabirinto**
  - ✓ per favorire leggibilità, ed autocontenimento dei test, e per non vincolarli all'effettiva presenza di file, usare fixture specificate tramite stringhe direttamente nei test stessi:
    - *suggerimento*: ricorrere a **StringReader**
  - ✓ Scrivere diversi test-case su fixture di complessità crescenti: labirinto «monolocale», «bilocale», ecc.
- Un plausibile es. della sintassi di una specifica di labirinto:
  - Stanze:
  - N10
  - Biblioteca
  - Estremi:
  - N10
  - Biblioteca
  - Attrezzi:
  - Osso 5 N10
  - Uscite:
  - N10 nord Biblioteca
  - Biblioteca sud N10

# Esercizio 16

- Modificare la classe **CaricatoreLabirinto** affinché sia possibile caricare anche personaggi, stanze chiuse, stanze buie ecc. ecc.

# Esercizio 17 (**diadia.properties**)

- Modificare l'applicazione affinché la specifica delle costanti non sia cablata nel codice ma sia *esternalizzata* in un opportuno file di properties  
**diadia.properties** da distribuire assieme al codice
- Ad es.
  - il numero di CFU iniziali
  - il peso max della borsa
- Esportare l'applicativo in formato **.jar** e verificarne il funzionamento in un ambiente diverso da quello di sviluppo nonostante la dipendenza verso risorse aggiuntive rispetto al codice (ad es. il file **diadia.properties**)
  - ✓ *Suggerimento:* non cablare nel codice il percorso fisico del file di properties, ma *solo* il suo nome logico

# Esercizio 18 (Tipizzazione Lasca)

- Modificare l'applicazione affinché siano utilizzati i tipi enumerativi laddove ancora resistono dei concetti di primo ordine per il gioco che risultino tuttavia ancora troppo lascamente tipati
  - ✗ Ad esempio ancora e di nuovo *stringhe*...
  - ✓ Sono sempre le stesse 4; e spesso le chiamiamo per nome... (!)

# Esercizio 19 (Classi Nidificate)

- Trasformare **LabirintoBuilder** in una classe statica nidificata di **Labirinto**
  - Rendere il costruttore di **Labirinto** privato
  - Aggiungere quindi un factory method statico e pubblico:

```
public static LabirintoBuilder Labirinto.newBuilder()
```
- E' preferibile dichiarare la classe nidificata pubblica o privata? perché?
- Scrivere dei test di unità a supporto della verifica di correttezza del codice prima e dopo questi cambiamenti

# Esercizio 20 (*try-with-resource*)

- Lo **scannerDiLinee** nel metodo **leggiRiga()** di **IOConsole** non viene mai chiuso
  - Dobbiamo inserire l'invocazione di **Scanner.close()**
- Utilizzare la forma sintattica *try-with-resource*
- Cambiare tutto il codice affinché non si presenti l'errore dovuto alla prematura chiusura di **System.in**
  - *Suggerimento*: creare lo scanner direttamente nel metodo che deve gestirne l'intero ciclo di utilizzo, dalla creazione, al rilascio, per tutta la durata di ogni partita:
- ✓ Il **main()**



# MODALITA' DI CONSEGNA

- Per la consegna utilizzare [poo.roma3@gmail.com](mailto:poo.roma3@gmail.com)
  - Per consegnare usare questo indirizzo di email! Gestita automaticamente.
- Non inviare mail di richiesta di conferma di avvenuta ricezione, non saranno gestite
  - Eventuali problematiche saranno gestite successivamente, quando e se *veramente* serve
- Consegne multiple da evitare per quanto possibile
  - ✓ Inutile: fa fede la data della release non dell'email
  - ✓ Ma solo le email verranno gestite automaticamente per raccogliere le matricole di chi ha consegnato
- ATTENZIONE:
  - NON allegare direttamente archivi .jar, .zip, .tar.gz
  - ✓ Per motivi di sicurezza l'email non verrebbe consegnata
  - Aggiungere nel corpo del messaggio:
    - Il link al repository github (>>)
    - Descrizione degli eventuali malfunzionamenti noti, ma non risolti

# TERMINI E MODALITA' DI CONSEGNA

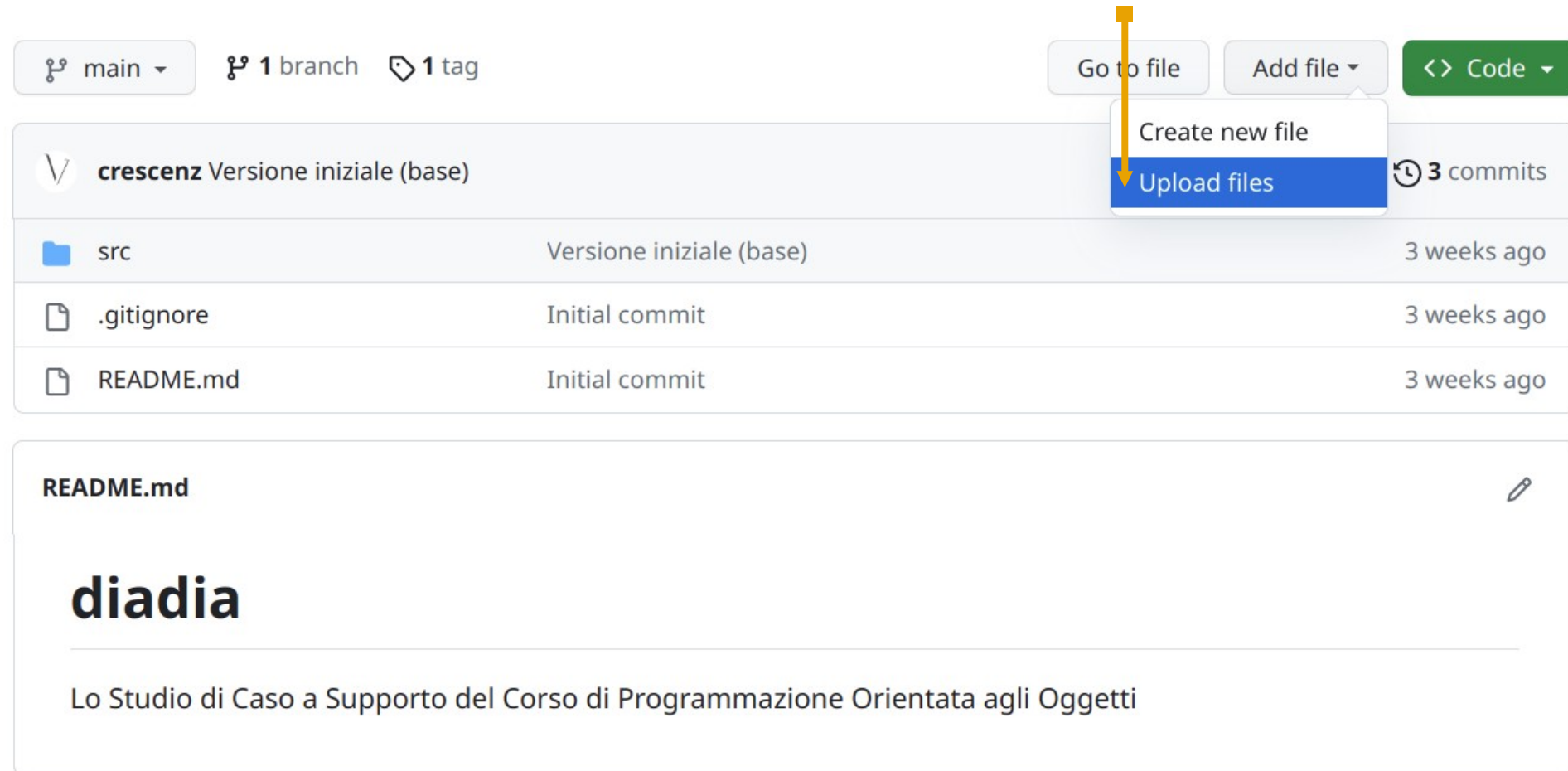
La soluzione deve essere inviata al docente entro le 21:00 di **domenica 8 giugno 2025** come segue:

- Svolgere in gruppi di esattamente 2 persone
- L'oggetto (subject) *DEVE* iniziare con la stringa **[2025-HOMEWORKC]** seguita dalle matricole

Ad es.: **[2025-HOMEWORKC] 612345 654321**

# Consegna con GitHub (1)

- Nella stessa pagina della repository github già creata per la consegna del primo homework utilizzare: upload files



The screenshot shows a GitHub repository named 'crescenz' with the 'main' branch selected. The repository has 1 branch and 1 tag. The 'Add file' dropdown menu is open, showing options to 'Create new file' and 'Upload files'. The 'Upload files' option is highlighted in blue. Below the repository overview, a table lists the files in the repository:

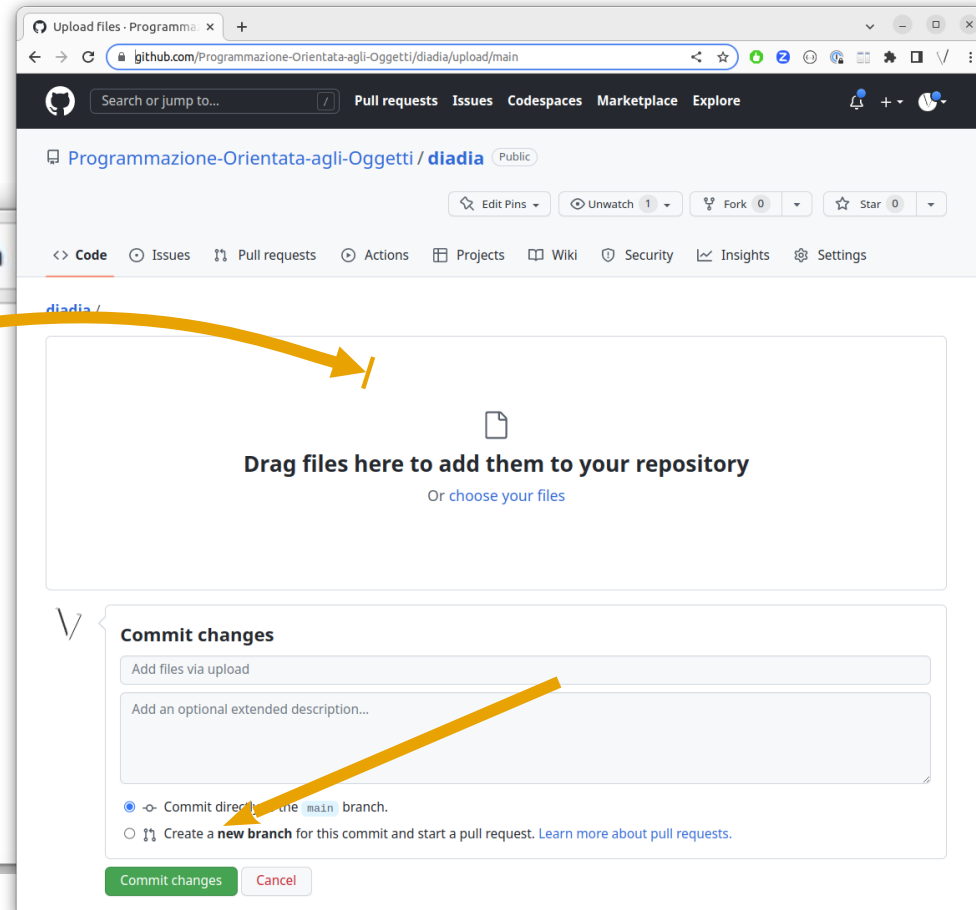
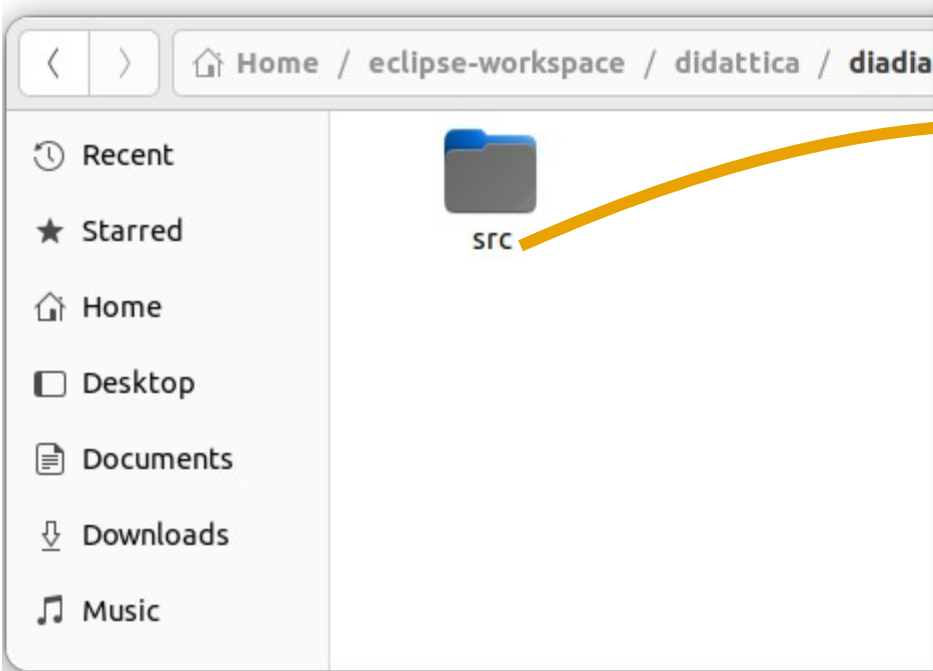
| File       | Commit                   | Time        |
|------------|--------------------------|-------------|
| src        | Versione iniziale (base) | 3 weeks ago |
| .gitignore | Initial commit           | 3 weeks ago |
| README.md  | Initial commit           | 3 weeks ago |

Below the table, the 'README.md' file is displayed. The title is 'diadia' and the content is 'Lo Studio di Caso a Supporto del Corso di Programmazione Orientata agli Oggetti'.

**2025 - HOMEWORK - 612345 - 654321**

# Consegna con GitHub (2)

- Navigare fino alla directory del vostro progetto locale
  - ad es. `/eclipse-workspace/didattica/diadia/`
  - Trascinare *tutta* la cartella `src` dentro il git repository già creato
    - Successivamente ripetere l'operazione con la cartella `test`
- Quindi salviamo le modifiche
  - tasto verde *Commit changes*



# Consegna con GitHub (3)

- Caricare *tutte* le modifiche rispetto al precedente homework
- Creare una nuova *release*
  - Selezionare <<create new release>> a dx nella pagina

The screenshot shows a GitHub repository named 'crescenz'. At the top, there are buttons for 'main', '1 branch', and '1 tag'. Below this is a table of files: 'src' (Versione iniziale (base)), '.gitignore' (Initial commit), and 'README.md' (Initial commit). The 'README.md' file is expanded, showing the text 'diadia' and 'Lo Studio di Caso a Supporto del Corso di Programmazione Orientata agli Oggetti'. On the right side, there is a sidebar with 'About' (Lo Studio di Caso a Supporto del Corso di Programmazione Orientata agli Oggetti), 'Releases' (1 tags, Create a new release), and 'About' (Lo Studio di Caso a Supporto del Corso di Programmazione Orientata agli Oggetti). A yellow arrow points from the text 'Selezionare <<create new release>> a dx nella pagina' to the 'Create a new release' button in the sidebar.

- Scegliere il cosiddetto *tag* della release
  - Questa volta usare **versione.C**
  - Come nome della release inserire
  - 2025-HOMEWORK**C** <<matricola1>> <<matricola2>>
    - Ad es. 2025-HOMEWORKC 612345 654321
- Nella descrizione scrivere eventuali malfunzionamenti noti ma non risolti