Esame Metodologie di Programmazione 2023-2024

20/11/2023

Il documento mostra le regole di base per l'esame di *Metodologie di Programmazione*. Il documento invalida automaticamente tutte le precedenti versioni degli anni passati.

ATTENZIONE: per comprendere i contenuti del documento, soprattutto dalla terza sezione in poi, è necessario che abbiate già seguito buona parte del corso o che abbiate già studiato buona parte del materiale del corso.

1 Regole di base

L'esame consiste di

- Progetto da svolgere a casa (individuale, NON in gruppi), da consegnare prima dell'orale (maggiori dettagli saranno forniti nelle prossime sezioni);
- Orale (solo se il progetto è stato considerato sufficiente) sugli argomenti del corso, che include anche la discussione del progetto.

La data ufficiale dell'appello coincide con la data dell'esame orale.

Dovreste lavorare al progetto solo dopo aver studiato tutto il materiale del corso (incluse le esercitazioni) cioè solo dopo che siete pronti per sostenere anche l'esame orale. Maggiori dettagli a tal proposito e riguardo ai tempi di consegna e alla validità dei progetti saranno mostrati nelle prossime sezioni. Allo stesso modo, maggiori dettagli sul progetto saranno mostrati nelle sezioni successive.

Se il progetto consegnato è sufficiente sarà discusso durante l'orale. Durante l'orale, si deve essere in grado di discutere ogni singolo aspetto e dettaglio di quello che si è implementato nel progetto e di saper commentare in modo appropriato tutte le scelte effettuate nell'implementazione del progetto. Inoltre, durante l'orale, gli studenti devono essere in grado di rispondere alle domande sugli argomenti del corso in modo formale e con la dovuta proprietà di linguaggio tecnico.

Il voto finale, che sarà assegnato solo se entrambe le parti sono sufficienti, consisterà della media dei risultati delle due parti.

Il progetto viene annullato se:

- Lo studente non supera l'esame orale (cioè non prende la sufficienza all'orale) oppure
- Lo studente si prenota per l'esame orale ma non si presenta senza darne preavviso oppure
- Lo studente rifiuta il voto finale oppure
- Il progetto perde di validità (si faccia riferimento alla sezione successiva)

Se il progetto viene annullato, lo studente dovrà farne uno completamente nuovo.

Se il progetto consegnato è sufficiente sarà discusso SOLO durante l'orale: non potete chiedere durante il ricevimento o per email le motivazioni del voto di un progetto sufficiente. Se non ritenete il voto del progetto soddisfacente, potete annullare il progetto (inviando un'email al docente e assicurandovi di ricevere la conferma da parte del docente) e a quel punto potete conoscere le motivazioni del voto durante un ricevimento (vale quanto detto sopra riguardo l'annullamento del progetto).

Gli studenti il cui progetto è stato valutato insufficiente sono vivamente invitati a partecipare a un ricevimento per conoscere le motivazioni che hanno portato all'insufficienza. Lo stesso vale per gli studenti che decidono di annullare il progetto perché non ritengono il voto soddisfacente.

2 Consegna e Orale

Indipendentemente dalle date di consegna e degli appelli, potete implementare il progetto in qualsiasi momento dell'anno (come vedrete di seguito, il progetto non vi viene assegnato: siete voi a decidere cosa implementerete nel progetto).

Tuttavia, come accennato sopra, dovreste lavorare al progetto solo dopo aver studiato tutto il materiale del corso (incluse le esercitazioni), cioè solo dopo che siete pronti per sostenere anche l'esame orale. Per questo motivo, potete sostenere l'orale solo nello stesso appello in cui consegnate il progetto (ovviamente, solo se il progetto è stato valutato sufficiente), **oppure**, nell'appello immediatamente successivo, a meno che l'appello successivo non sia a distanza di mesi. Il progetto non sarà più valido al di fuori del suddetto lasso di tempo, e dovrà essere rifatto completamente ex-novo. Valgono ovviamente anche le regole già esposte riguardo all'annullamento dei progetti.

Per esempio, supponiamo, senza specificare il giorno, che gli appelli siano così distribuiti: gennaio, febbraio, giugno I, giugno II, luglio, settembre. Se consegnate il progetto a gennaio (ottenete la sufficienza e non volete sostenere subito l'orale) potete sostenere l'orale a febbraio. Lo stesso vale per giugno I e giugno II, giugno II e luglio, luglio e settembre. Mentre se consegnate il progetto a febbraio potete sostenere l'orale solo in quell'appello. Lo stesso vale per settembre. Anche le seguenti combinazioni non sono valide: giugno I – luglio e giugno II – settembre.

Ricordate comunque che (vedere sopra) durante l'orale, si deve essere in grado di discutere ogni singolo aspetto e dettaglio di quello che si è implementato nel progetto e di saper commentare in modo appropriato tutte le scelte effettuate nell'implementazione del progetto. Se quindi non date l'orale nello stesso appello in cui consegnate il progetto, prima dell'orale assicuratevi di riguardare in modo approfondito il progetto.

La consegna dei progetti avverrà unicamente tramite Moodle. Le consegne saranno abilitate al momento che saranno aperte le iscrizione per un dato appello (sul sito studenti.unifi.it). Le consegne su Moodle rimarranno aperte solo per qualche giorno durante la finestra di iscrizione all'appello. Le valutazioni dei progetti saranno comunicate tramite Moodle (riceverete un avviso personale).

Il docente eseguirà la valutazione del progetto dopo che lo studente avrà consegnato il progetto su Moodle **e** si sarà iscritto (sul sito studenti.unifi.it) per l'appello. Da quel momento non sarà più possibile modificare il progetto. Progetti consegnati su Moodle ai quali non corrisponde un'iscrizione non verranno presi in considerazione (e come tali potranno essere modificati e

consegnati negli appelli successivi). Si invita caldamente gli studenti a consegnare su Moodle e iscriversi subito all'appello sul sito studenti.unifi.it, senza lasciar passare ore o giorni.

Al momento dell'iscrizione all'appello sul sito studenti.unifi.it, nel caso si intenda sostenere l'orale nell'appello successivo (vedere le regole sopra), si dovrà specificare nel campo note la stringa "SOLO PROGETTO". Nel caso ci si iscriva solo per l'orale (vedere le regole sopra), si dovrà specificare nel campo note la stringa "ORALE". Quindi, in mancanza di note, si intenderà che lo studente intende sia consegnare il progetto sia sostenere l'orale (in caso di valutazione sufficiente del progetto) in quell'appello. Ricordate che vi dovete iscrivere all'appello anche se dovete sostenere l'orale.

Il risultato della valutazione del progetto sarà comunicata in tempo utile prima della data dell'orale. A tal proposito, sarà data la precedenza alla valutazione dei progetti degli studenti che intendono sostenere l'orale nello stesso appello.

Ricordate che potete lavorare al progetto quando volete, indipendentemente dalle finestre di consegna (ma dovreste farlo solo dopo aver studiato tutto il materiale del corso e le esercitazioni).

In generale, si consiglia vivamente di sostenere l'esame orale nello stesso appello di consegna del progetto.

3 Il progetto

Il progetto consiste in un sistema software a vostra scelta che dovete implementare in Java, rispettando i vari principi mostrati nel corso (inclusi leggibilità e pulizia di codice) e applicando qualche pattern.

Per la scelta del software da implementare NON potete riusare codice o esempi visti a lezione o a esercitazione. Ad es., NON implementate un *DiscountStrategy* o uno *StringFinder*, ecc.

Il progetto NON deve essere incentrato tanto sulla parte computazionale quanto sulla corretta (secondo quanto visto nel corso) strutturazione in interfacce e classi, sulla collaborazione fra tali interfacce e classi e sulla collaborazione tra oggetti.

Il progetto deve essere corredato da dei test JUnit (eventualmente usando librerie aggiuntive come AssertJ – librerie aggiuntive devono essere parte del progetto). Tali test devono dare garanzie sulla correttezza dell'implementazione nel modo più esaustivo possibile e devono essere anche loro scritti bene, leggibili e facilmente comprensibili.

Il progetto non deve essere necessariamente un programma utilizzabile in pratica: deve essere un prototipo che mostra le vostre capacità di design e implementazione usando l'OOP e i principi e pattern visti a lezione. Quindi deve essere più una modellazione di un sistema software che un software realmente utilizzabile in pratica. Ovviamente, anche tale modellazione deve essere corretta e concreta, e testabile tramite test JUnit. Una classe col metodo *main* NON è necessaria. Si consiglia inoltre di NON fornire nessun tipo di interfaccia utente, nemmeno da riga di comando.

Siccome il progetto deve solo modellare un prototipo di sistema software, potete avere parti non concretamente implementate, se non sono rilevanti ai fini degli argomenti del corso e, soprattutto se non implementabili coi mezzi e le conoscenze a vostra disposizione. Tali parti saranno rappresentate da un'astrazione, senza nessuna implementazione concreta. Per esempio, ricordate l'interfaccia *FileSystemPrinter* che abbiamo usato alle esercitazioni che modellavano un file system: la usavamo

per “delegare” le funzionalità di *print*, astraendo dall’implementazione effettiva della stampa (su schermo, su stampante, su file, ecc.). Non abbiamo mai implementato tale interfaccia con una classe concreta. Abbiamo creato un’implementazione fittizia (“mock”) nella directory dei test che fosse sufficiente a permetterci di testare le funzionalità delle altre classi concrete del sistema che abbiamo implementato. In quel particolare contesto, non ci interessava avere un’implementazione effettiva di *FileSystemPrinter*. Tuttavia, tutto quello che fa parte del codice del progetto (cioè quello che sta nella directory sorgente principale, non quello che è nella directory dei test) deve avere un’implementazione effettiva. Per esempio, nel vostro codice viene delegato il pagamento di una somma di denaro a un oggetto dichiarato di tipo astratto (es., un’interfaccia) *PaymentService*. È lecito non avere nessuna implementazione del pagamento vero e proprio (magari solo nei test potete “mockare” quella parte). Anzi, NON dovete implementare il pagamento restituendo semplicemente una stringa, stampando su schermo o restituendo un booleano (questo sarebbe considerato molto negativamente).

Dovreste quindi trovare un contesto per applicare i principi visti nel corso e applicare qualche pattern. ATTENZIONE: non ha assolutamente senso applicare TUTTI i pattern visti a lezione. I pattern devono essere applicati solo nel contesto giusto e quando e se servono. Ricordate che applicare un pattern male è considerato un errore molto grave. Indicativamente, si dovrebbe cercare di usare qualche pattern (es., 3 o 4) che sia abbastanza complesso e interessante in quel contesto. Usare solo 2 pattern molto semplici (es., solo *Iterator* e *Builder*, oppure solo *Iterator* e *Adapter*) porterà quasi sicuramente a una valutazione insufficiente del progetto.

Quando applicate un pattern, cercate di avere almeno due classi concrete per quel pattern. Per esempio se usate il *Decorator* cercate di implementare almeno due decorator concreti. Allo stesso modo per *Iterator*, *Visitor*, ecc. In modo simile, se usate un pattern basato sulla ridefinizione di uno o più metodi (ad es, rispettivamente, *Factory Method* e *Template Method*), dovreste avere almeno due classi derivate che ridefiniscono il metodo o i metodi coinvolti nel pattern. Tuttavia, non ha senso avere tantissime implementazioni concrete relative allo stesso pattern, quindi cercate di non esagerare. Avere solo la parte “astratta” di un pattern, senza nessuna implementazione concreta, non sarà considerato come l’applicazione di un pattern. Tornando all’esempio del *PaymentService* di poco fa, senza nessuna implementazione concreta (che, come detto sopra, è ammissibile non avere) non potete dire che state applicando il pattern *Strategy*. Ricordate comunque che non sareste in grado di implementare in modo effettivo un servizio di pagamento per quanto detto sopra.

Le varie parti del vostro progetto devono in qualche modo essere collegate fra di loro. Non ci devono essere parti “isolate”, cioè che non usano nessun’altra classe e che non sono usate da nessun’altra parte del vostro progetto. In pratica, nel diagramma delle classi (vedere dopo per quanto riguarda i diagrammi UML), non ci devono essere parti “isolate”. Cioè, se vedete il vostro diagramma delle classi come un grafo, questo deve essere “connesso”. Per esempio, se implementate delle classi per *ShopClient* e delle classi per *Shop* (tutte con relativi pattern) che non comunicano mai fra di loro, state sbagliando il progetto. Oppure se implementate uno *Strategy* che non viene mai usato da altre classi del progetto, state sbagliando il progetto. Non ha senso usare lo stesso pattern più volte per parti diverse. Allo stesso modo non ha senso implementare piccole varianti concrete come *MinPriceVisitor* e *MaxPriceVisitor* (soprattutto se poi avete duplicazione di codice!).

Nel caso dobbiate modellare gli oggetti del dominio applicativo tramite una gerarchia di classi (cosa molto probabile) cercate di non creare una gerarchia inutilmente troppo grossa. Fate riferimento agli

esempi e alle esercitazioni (nell'esempio del file system ci sono solo una classe astratta e due classi concrete per rappresentare gli oggetti del dominio applicativo). Avere tante classi nella gerarchia del dominio applicativo è controproducente ai fini del progetto. A tal proposito, ha senso avere classi concrete differenti per gli oggetti del dominio applicativo solo se poi si fanno operazioni differenti a seconda del tipo (ricordate che, nell'esercitazione delle espressioni e del visitor, *Sum* e *Multiplication*, che estendono *BinaryExpression*, non hanno uno stato che le contraddistingue, ma le operazioni che agiscono su tali oggetti sono diverse a seconda del tipo effettivo di tali oggetti).

Eventualmente, potete anche utilizzare altri pattern non visti al corso (ma presenti in letteratura); in tal caso, dovreste descrivere brevemente tali pattern nella relazione (vedere più avanti per quanto riguarda la relazione).

Si ricorda comunque che ai fini dell'esame si deve fare riferimento al materiale mostrato a lezione e disponibile sul sito Moodle, ed eventualmente, per approfondimenti su base volontaria, a libri o articoli consigliati nel corso. Fare riferimento ad altro materiale, soprattutto trovato online, può essere rischioso, in quanto non necessariamente affidabile. Il materiale online non sarà automaticamente ritenuto autorevole e accreditato. La decisione finale a tal proposito spetterà solo al docente.

Sbagliare ad applicare un pattern porterà quasi sicuramente a rendere il progetto insufficiente.

Il progetto deve essere un progetto Eclipse (cioè, inclusi i file e directory tipici di un progetto Eclipse, *.project*, *.classpath*, directory *.settings*, ecc.), consegnato zippato, senza i file *.class* (cioè, senza la directory *bin*) ma con eventuali librerie aggiuntive (ad es., AssertJ). Tale zip contenente il progetto Eclipse deve essere importabile dal docente in Eclipse tramite "Import" → "Existing Projects into Workspace" → "Select archive file" senza ulteriori operazioni e aggiustamenti da parte del docente.

Il progetto, una volta importato, deve compilare senza errori e senza warning.

In particolare,

- Se ci sono errori di compilazione il progetto non sarà valutato (inclusi i problemi dovuti all'errata configurazione del progetto per la versione di Java, come spiegato sotto)
- Se non ci sono test JUnit il progetto non sarà valutato
- Se ci sono test JUnit che falliscono il progetto non sarà valutato
- Se ci sono dei warning il voto del progetto subirà delle penalizzazioni

Se un progetto non viene valutato, non siete tenuti a farne uno completamente nuovo nell'appello successivo, quindi lo potete risottomettere all'appello successivo dopo avere sistemato i problemi che lo hanno reso non valutabile. Tenete però in considerazione che, non essendo stato valutato, il progetto potrebbe contenere altri errori o problemi che in fase di valutazione successiva potrebbero portare all'insufficienza. Infatti, "non valutato" vuol dire proprio che non è stato per niente controllato dal docente, il quale ha rilevato solo i problemi che lo hanno reso non valutabile. NON potete risottomettere il progetto allo stesso appello, anche se la sistemazione dei problemi che hanno reso il progetto non valutabile è sistemabile in poco tempo.

Il progetto deve essere debitamente configurato per Java 8, oppure, se volete per Java 11. Non usate altre versioni di Java che non siano la 8 o la 11. Fate riferimento alle slide “Configurare Java in Eclipse”.

Dovete assicurarvi che eseguendo tutti i test insieme (cioè con “Run As” → “JUnit test” dal menù contestuale del progetto) non si abbia nessun fallimento ne’ errore.

Il nome del progetto Eclipse deve essere della forma “progetto.mp.*cognome.nome*” (ATTENZIONE: prima il cognome e poi il nome, differentemente dagli anni precedenti) dove dovete sostituire *nome* e *cognome* col vostro nome e cognome (in caso di doppi nomi, usatene solo uno), tutto minuscolo. La cartella in cui è contenuto progetto deve avere lo stesso nome del progetto. Il nome del file zip deve essere “progetto.mp.*cognome.nome*.zip”, tutto minuscolo. I nomi dei package Java sono a vostra libera scelta. Progetti consegnati che non rispettano tali specifiche non saranno valutati (vedere sopra per quanto riguarda i progetti non valutati).

Si consiglia vivamente di utilizzare sempre l’ultima versione rilasciata di Eclipse (ricordate che Eclipse viene rilasciato ogni 3 mesi, e i numeri di versione sono del tipo “2023-06” rilasciata a giugno, “2023-09” rilasciata a settembre, “2023-12” rilasciata a dicembre, ecc.). Il docente utilizzerà infatti sempre l’ultima versione rilasciata di Eclipse per valutare i progetti.

All’interno del progetto Eclipse, nella cartella principale del progetto (quindi visibile anche da Eclipse), deve essere inclusa anche una relazione (indicativamente di circa 10 pagine), in formato PDF. La relazione deve includere

- all’inizio una descrizione delle funzionalità del sistema implementato e poi, non necessariamente in quest’ordine,
- diagrammi UML delle classi e interfacce e, se necessario o opportuno, eventuali diagrammi UML di interazione fra gli oggetti (vedere la prossima sezione per quanto riguarda l’UML)
- una descrizione sufficientemente dettagliata delle scelte di design e implementative e in particolare le motivazioni per i pattern applicati e le strategie per testare
- una lista che riassume i pattern applicati (se si tratta di pattern visti a lezione non dovete descrivere i pattern, se invece si tratta di pattern non visti a lezione dovete anche descrivere brevemente i nuovi pattern)

I progetti e le implementazioni non devono essere simili a quelli di altri studenti, quindi si consiglia di NON lavorare al progetto insieme ad altri colleghi e di NON confrontarvi tra di voi riguardo al progetto in nessun modo. Progetti ritenuti simili (anche se confrontati con progetti consegnati in passato) saranno annullati.

NOTA. Lo studente deve lavorare in modo autonomo al progetto, senza confrontarsi continuamente con il docente sulle scelte di progettazione e implementazione, che sono oggetto della valutazione.

4 Software UML

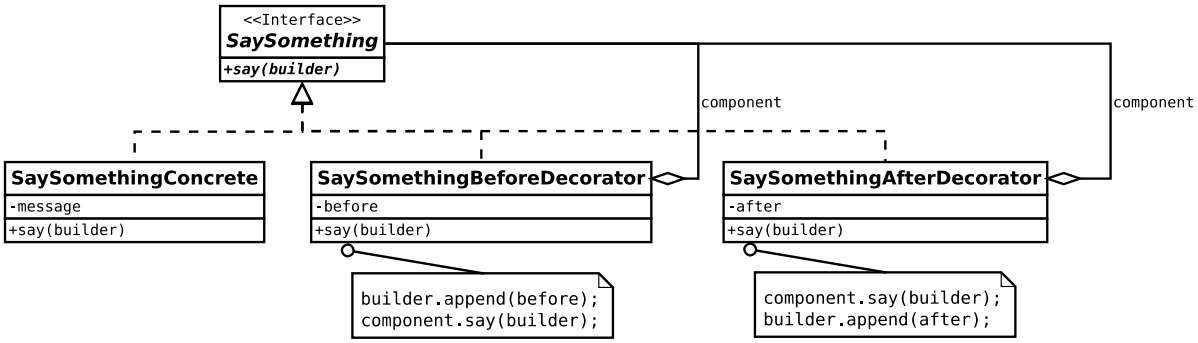
Per creare i diagrammi UML nella vostra relazione si consiglia l’uso di software appropriato. Siete liberi di usare il software che più preferite e siete invitati a valutare diversi software fino a che non trovate quello che fa per voi. Di seguito sono riportati alcuni software che ho personalmente usato (tutti gratuiti e open-source), che permettono di esportare in tantissimi formati:

- *Dia*, software per diagrammi vari, che include anche una sezione apposita per UML. Leggermente spartano ma vi lascia molta libertà ed è relativamente semplice da utilizzare. https://wiki.gnome.org/Apps/Dia
- *yEd*, utilizzabile eventualmente anche direttamente dal sito web. https://www.yworks.com/products/yed
- *Eclipse Papyrus*, plugin Eclipse per diverse tipologie di diagrammi, tra cui UML; il più potente e completo dei 3, ma anche il più complesso da usare. https://www.eclipse.org/papyrus/

Il diagramma o i diagrammi contenuti nella relazione devono essere facilmente leggibili. Inoltre, i diagrammi devono essere effettivamente relativi al vostro progetto, quindi i nomi delle classi, interfacce, campi e metodi devono trovare corrispondenza nel vostro codice. Se cambiate qualcosa nel codice (anche semplice renaming) ricordatevi di aggiornare in modo opportuno i diagrammi (e viceversa).

Tenete conto che per i diagrammi che dovrete disegnare voi, non avrete bisogno di tutte le funzionalità complete di UML, e senz'altro non avrete bisogno delle caratteristiche delle versioni più recenti di UML.

Per esempio, il diagramma mostrato di seguito (visto a lezione) è stato fatto con Dia e poi esportato in formato SVG (nel documento è stato poi incluso il file in formato SVG):



5 Check list

Di seguito sono mostrati, in modo non necessariamente esaustivo ne' in ordine di importanza, una serie di cose da tenere a mente per lo sviluppo del progetto. Tutto quello che segue è relativo ai contenuti del corso mostrati a lezione e presenti nelle slide, tutorial e nelle esercitazioni. Tali cose saranno usate per la valutazione del progetto. Non rispettare anche un solo punto di quello che segue porterà ad abbassare il voto del progetto.

In generale, ricordate che oltre all'uso corretto dei pattern, conta molto l'adesione ai principi di design visti nel corso (S.O.L.I.D.).

Rispettate le convenzioni (maiuscole/minuscole) di Java per quanto riguarda i nomi (di classi e interfacce, variabili di istanza, costanti, nomi dei metodi, nomi dei pacchetti, ecc.)

Cercate di strutturare le classi in pacchetti Java in modo opportuno. Evitate di avere tutte le classi in un unico pacchetto.

I test devono essere in una cartella sorgente separata dalla cartella sorgente del codice vero e proprio.

Fate attenzione a specificare correttamente la visibilità dei vari membri. Ricordate che le variabili di istanza devono essere tutte private. Inoltre, i metodi devono essere pubblici solo se sono pensati per essere usati da "client" (cioè anche da vostro codice). Ricordate che se un metodo serve solo nei test NON deve essere reso pubblico (ma si deve usare il "package-private"). L'uso scorretto della visibilità sarà considerato un errore molto grave.

Assicuratevi che il vostro progetto Eclipse sia configurato correttamente per quanto riguarda la versione di Java che intendete utilizzare (vedere sopra per quanto riguarda le versioni di Java ammesse).

Non usare accenti o caratteri non standard nei nomi delle directory, dei pacchetti, delle classi e dei membri (evitando così problemi di encoding che porterebbero il codice a non essere compilabile in alcuni sistemi operativi). Non usate caratteri accentati o non standard nemmeno nelle stringhe (potrebbero portare al fallimento dei test a seconda del sistema operativo usato). Ricordate che se il codice non compila o un test fallisce il progetto non verrà valutato.

Attenti ai carattere di fine linea che sono diversi fra Windows ("\r\n") e Unix ("\n"). Nel dubbio, usate sempre e solo "\n", e non leggete tale informazioni dalle proprietà di Java (altrimenti dei test che avrebbero successo su Windows fallirebbero su Linux o MacOS, e viceversa).

Utilizzare nomi sensati per tutto il vostro codice (classi, variabili di istanza, metodi, ecc.). Se utilizzate l'Inglese per tali nomi, assicuratevi che i nomi siano corretti in Inglese (senza errori di sintassi e in modo che abbiano un senso compiuto – ricordate, ad es., che *EmployeeShop* vuol dire "negozio degli impiegati" mentre probabilmente avreste voluto dire "impiegato del negozio", cioè *ShopEmployee*). Se non siete sicuri, utilizzate l'Italiano. Ovviamente dovete essere consistenti: o usate l'Inglese o l'Italiano, non un misto dei due.

Evitate nomi inutilmente fuorvianti, soprattutto per i nomi dei campi e dei metodi. Per esempio

```
Map<String, Student> listOfStudents
```

è oltremodo fuorviante: sembra che il campo rappresenti una lista di studenti quando in realtà è una mappa! Inoltre "listOf", oltre a essere fuorviante, è inutilmente specifico. Basterebbe chiamare il campo "students", o, se proprio volete dire che è una mappa, almeno "studentMap". Allo stesso modo il nome del campo non deve essere fuorviante per quanto riguarda i contenuti di una collection. Per esempio, entrambe le dichiarazioni sono fuorvianti

```
Collection<Student> persons
```

```
Collection<Person> students
```

Evitate inutili astrazioni. In generale, tenete a mente che per ogni astrazione ci dovrebbero essere almeno due implementazioni concrete (fa eccezione quanto detto sopra riguardo a parti non implementate a cui ci si riferisce solo tramite un'astrazione, eventualmente mockata solo nei test). Se avete un'astrazione e una sola implementazione concreta c'è qualcosa che non va. Per esempio, non ha senso avere *AbstractEmployee* se avete poi solo un'implementazione concreta *Employee*.

Il codice deve essere formattato con cura, correttamente e in modo consistente. Per indentare usate o TAB o spazi, ma non un mix dei due. Evitate righe vuote che non hanno senso, soprattutto in corrispondenza di parentesi graffe (dopo "{" e prima di "}"). Per separare i campi, usate una riga

vuota o nessuna riga vuota, in modo consistente. Questo è un esempio di codice formattato senza cura:

```
public class MyClass    implements  Expression {

    private String name;
    private int age;


    private final boolean checked;

    public MyClass(String name, int age) {

        this.name= name;
             this.age= age;

    }

    public String getName() {       return left;
    }
}
```

Prestate particolare attenzione a istruzioni che usano una fluent API (method chaining). Per quelle parti evitate di usare il formattatore automatico di Eclipse (oppure personalizzatelo in modo opportuno). Tali istruzioni dovrebbero avere un'invocazione di metodo su ogni riga (di solito, potrebbe fare eccezione la primissima invocazione, se necessario). Per esempio, invece di

```
students.stream().filter(…).map(…).collect(toList())
```

```
assertThat(…).isEmpty().containsExactly(…)
```

usate questa strategia di formattazione (che rende indiscutibilmente il codice più leggibile):

```
students.stream()
      .filter(…)
      .map(…)
      .collect(toList())
assertThat(…)
      .isEmpty()
      .containsExactly(…)
```

Ovviamente, se c'è una singola invocazione di metodo, dovrebbe stare tutto sulla stessa riga.

Evitate di lasciare commenti "// TODO" (tipicamente inseriti da Eclipse quando vi genera qualcosa). Allo stesso modo, NON lasciate istruzioni commentate (tipicamente istruzioni System.out.println che avete usato per "debug" – cosa che peraltro non dovreste mai fare in generale). Per esempio, evitate cose del genere

```
// System.out.println(students);
students.add(…);
// System.out.println(students);
```

Assicuratevi prima di consegnare di aver rimosso tali righe di codice commentate.

Evitate forme sintattiche “bizzarre”. Per esempio, non scrivete mai qualcosa del genere

```
if (exp)
	return true;
else
	return false;
```

La forma corretta è

```
return exp;
```

Ovviamente vale anche per espressioni del tipo “!exp”.

Evitate l’uso di “instanceof” e conseguente down cast. Ricordate che se dovete fare qualcosa in base al tipo runtime di un oggetto esiste un pattern apposito. In generale, il down cast dovrebbe essere usato solo in rarissimi casi e basandosi su particolari assunzioni (fate riferimento a una delle esercitazioni). Per sicurezza, quindi, NON usate MAI “instanceof” e down cast! Come detto a lezione, il down-cast nelle implementazioni di “equals” è inevitabile.

Non testate i getter e i setter a meno che non abbiano della logica da testare.

Usate AssertJ solo se usate le sue funzionalità non presenti in JUnit (se usate solo *assertThat(…).isEqualTo* non ha senso usare AssertJ).

Testate in modo appropriato le eccezioni: evitate la forma “manuale” di try … fail … catch …: usate invece *assertThrows* oppure il corrispettivo di AssertJ).

Se nei test chiamate un metodo che può sollevare un’eccezione e tale caso rappresenta un fallimento usate la clausola *throws* sulla firma del test: NON catturate l’eccezione e eseguite un *fail*.

Preferite soluzioni “dichiarative” (es., lambda e stream) ai cicli. In particolare, evitate cicli “while” e cicli “for (… ; … ; …)”. Ovviamente, fa eccezione l’eventuale uso del pattern *Iterator*. Scrivere codice come il seguente per iterare su una lista comporterà una seria penalità nella valutazione:

```
for (int i = 0; i < students.size(); i++) {
	Student s = students.get(i);
	...
}
```

Se usate direttamente l’implementazione di *Iterator* di Java, non potete dire che avete applicato il pattern *Iterator*. (Lo avete solo usato, quindi non deve apparire nella relazione fra i pattern che avete applicato.)

Evitate di mettere troppi campi in una classe. Questo renderebbe inutilmente il vostro codice più difficile da gestire soprattutto in un contesto didattico come il nostro, che non necessariamente deve ricreare completamente un caso d’uso reale.

Ricordate che le classi e i metodi dovrebbero in generale essere molto piccoli. Nel caso, utilizzate dei refactoring.

Evitate l’uso di System.out.print e System.err.print nel modo più assoluto.

Ricordate che stiamo usando un linguaggio OOP, quindi lo stato degli oggetti deve essere modellato con gli strumenti OOP. Per esempio, se vi trovate a rappresentare lo stato di un oggetto tramite una

mappa la cui chiave è una stringa che identifica un campo e il cui valore è il valore di quel campo, allora state sbagliando tutto!

Evitate l'uso di campi statici (ovviamente sono escluse le costanti) e di "singleton", se questi mantengono uno "stato" e possono essere "mutati". Ricordate che questi vi daranno seri problemi nei test!

Fate attenzione al codice duplicato: evitatelo ma senza rinunciare alla leggibilità e mantenibilità. In particolare, fate attenzione a codice che potrebbe sembrare duplicato ma che in realtà non lo è, e quindi non deve essere trattato come tale.

Fate attenzione a eventuali variabili di istanza (che, ricordate, devono tutte essere private) ereditate nelle classi derivate: chiedetevi se abbia senso che una classe derivata erediti certe variabili di istanza (che, ricordate, anche se non direttamente accessibili, fanno comunque parte dello stato a runtime degli oggetti corrispondenti), cioè chiedetevi se certe variabili di istanza debbano far parte di una classe base. Chiedetevi inoltre se una variabile di istanza in una classe base abbia esattamente la stessa semantica in tutte le possibili classi derivate.

Fate attenzione a dove definite *equals*, *hashCode* e *toString*. Questi metodi hanno senso solo in certe classi.

Evitate assolutamente di utilizzare la "concorrenza" (es., Thread, sincronizzazione, ecc.). La programmazione concorrente non fa parte di questo corso.

Evitate l'implementazione di codice computazionalmente troppo complicato. Come detto sopra, non è interessante la parte computazionale del progetto ma la parte di design OOP. Non dovreste quindi aver bisogno di implementare algoritmi di ordinamento, o algoritmi di ricerca su strutture dati. Se iniziate ad aver bisogno di tanti if...then...else, eventualmente annidati, state sbagliando tipologia di progetto. In tal caso, quasi sicuramente, il progetto sarà valutato insufficiente.

Se vedete che il vostro codice diventa troppo complesso e che vi riesce difficile capirlo, gestirlo o comprendere quello che succede durante l'esecuzione, probabilmente state sbagliando il design: è meglio iniziare da capo o rimuovere le parti complesse.

Se vi sembra che l'uso di un certo pattern nel vostro codice sia un po' "forzato" probabilmente state sbagliando nell'applicarlo o state sbagliando la modellazione o il design: è meglio iniziare da capo.

Infine, in generale, se nel corso (durante le lezioni, nelle slide e nelle esercitazioni) vi è stato detto di NON fare certe cose (ad es., perché violano dei principi OOP, ecc.), allora NON fatele nel progetto, in quanto la valutazione potrebbe essere insufficiente o molto bassa.