Ingegneria del Software a.a. 2013-14
Prova Scritta del 10 febbraio 2014

BOZZA DI SOLUZIONE

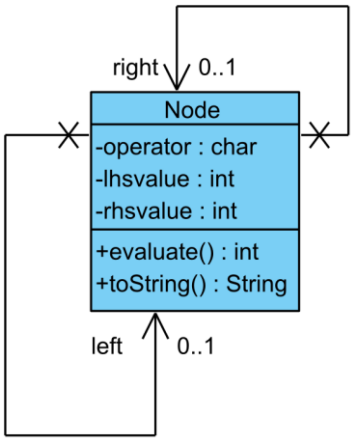
Tempo a disposizione: 3 ore

Esercizio 1

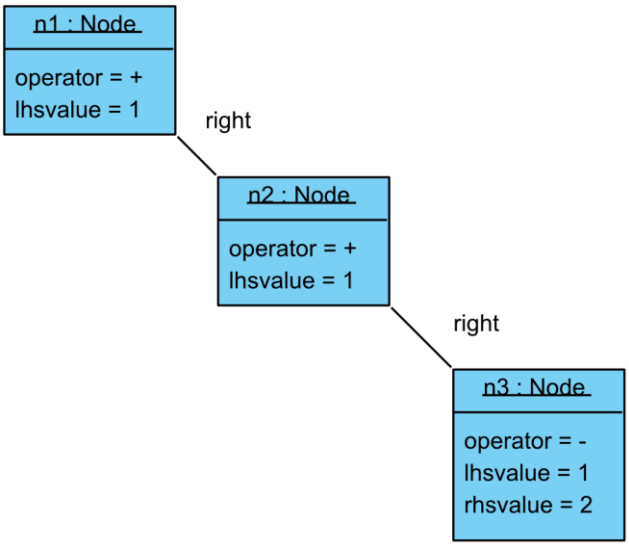
La classe Node rappresentata nel seguito è un implementazione ad albero binario delle espressioni aritmetiche (es. 3+4+2). Tramite l'operazione evaluate() è possibile valutare un espressione mentre toString() esegue la stampa a video. **Requisiti:** Nel caso venga valutata un'espressione con un operatore diverso da + e – il risultato **deve** essere zero. Se invece si prova a stampare un'espressione con un operatore diverso da + e – allora il risultato **deve** essere la stringa "error".

```
public class Node {
        char operator;
        int lhsValue, rhsValue;
        Node left, right;

        public int evaluate() {
                int leftVal = (left == null) ? lhsValue : left.evaluate();
                int rightVal = (right == null) ? rhsValue : right.evaluate();

                if (operator == '+') {
                        return leftVal + rightVal;
                } else if (operator == '-') {
                        return leftVal - rightVal;
                } else return 0;
        }
        public String toString() {
                String leftVal = (left == null) ? lhsValue+"" : left.toString();
                String rightVal = (right == null) ? rhsValue+"" : right.toString();

                if (operator == '+') {
                        return leftVal + "+" + rightVal;
                } else if (operator == '-') {
                        return leftVal + "-" +rightVal;
                } else return "error";
        }
}
```

1) Rappresentare con un class diagram la classe Node

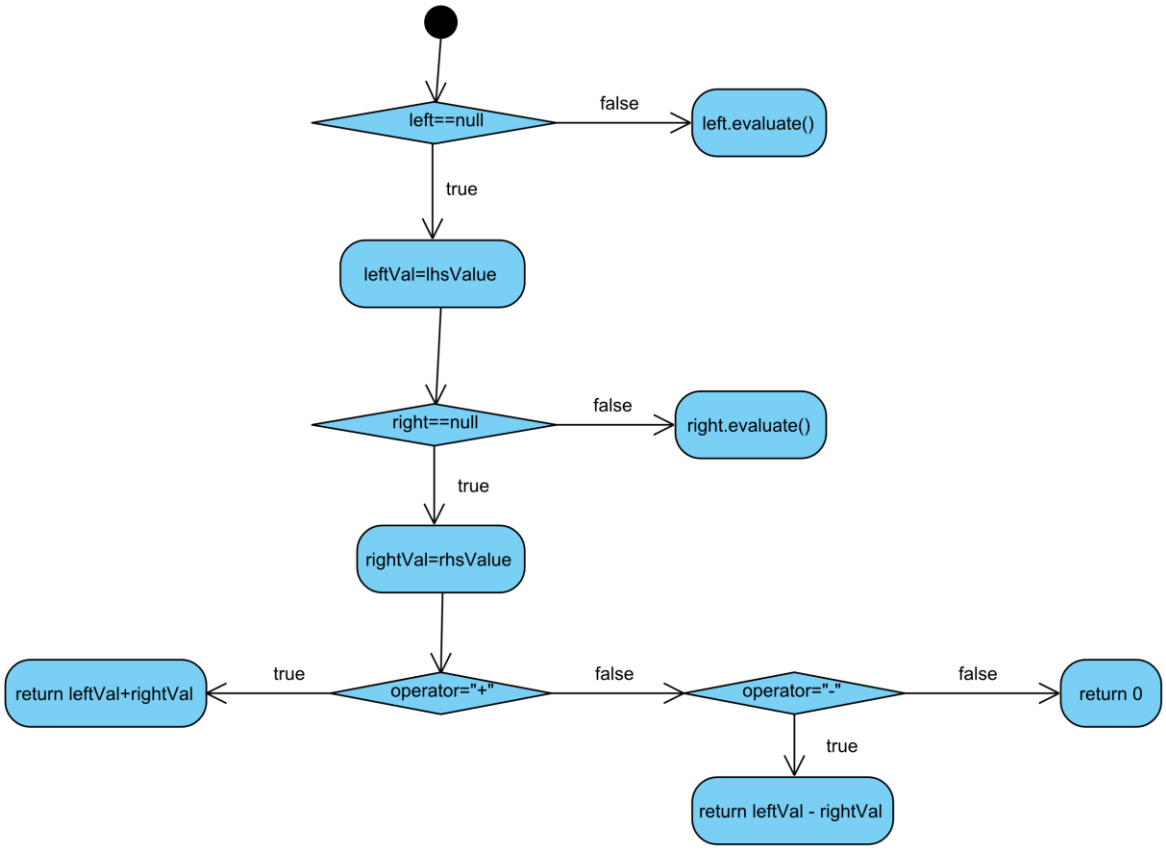


2) Rappresentare con un object diagram l'espressione **1+1+1-2**



3) Scrivere dei casi di test **JUnit** per la classe Node che abbiano branch coverage = 100%

Il CFG dei metodi evaluate() e toString() è lo stesso. Per avere branch coverage 100% occorre selezionare degli input che coprano tutti gli archi. Scriviamo solo i casi di test solo per evaluate(). Analogo per toString().



Aggiungiamo il costruttore alla classe Node:

```
public Node(char operator, int lhsValue, int rhsValue, Node left, Node right) {
            this.operator = operator;
            this.lhsValue = lhsValue;
            this.rhsValue = rhsValue;
            this.left = left;
            this.right = right;
}


public class testComplete100 {
      Node node1, node2, node3, node4, node5;

      @Before
      public void setUp() {
            node1 = new Node('+', 3, 4, null, null); // 3 + 4
            node2 = new Node('-', 4, 2, null, null); // 4 - 2
            node3 = new Node(':', 2, 0, null, node2); // 2 : 3 + 4
            node4 = new Node(':', 0, 2, node2, null); // 3 + 4 : 2
      }

      @Test
      public void testEvaluatePlus() {
            assertEquals(7, node1.evaluate());
      }

      @Test
      public void testEvaluateMinus() {
            assertEquals(2, node2.evaluate());
      }

      @Test
      public void testEvaluateError1() {
```

```
		assertEquals(0, node3.evaluate());
	}

	@Test
	public void testEvaluateError2() {
		assertEquals(0, node4.evaluate());
	}
}
```

4) Il codice non soddisfa i requisiti; scrivere dei casi di test **JUnit** che evidenziano le differenze tra codice e requisiti richiesti

   Il problema è ben evidenziato dai seguenti casi di test JUnit dove i casi di test `testEvaluateStrangeOperator2()` e `testToStringStrangeOperator3()` non "passano" perchè i velori ritornati sono rispettivamente differenti da zero e error

```
public class testNode {
	Node node4, node5;

	@Before
	public void setUp() {
		node4 = new Node('&', 3, 6, null, null); // 3 & 6
		node5 = new Node('+', 0, 2, node4, null); // 3 & 6 + 2

	}

	@Test
	public void testEvaluateStrangeOperator1() {
		assertEquals(0, node4.evaluate());
	}

	@Test
	public void testEvaluateStrangeOperator2() { // ritorna 2
		assertEquals(0, node5.evaluate());
	}

	@Test
	public void testToStringStrangeOperator3() { // ritorna "error + 2"
		assertEquals("error", node5.toString());
	}

}
```

5) **Affermazione**: esistono dei problemi/difficoltà a modificare la classe Node perché i due metodi evaluate() e toString() "smells". Commentare l'**affermazione** precedente considerando le seguenti domande: "Che modifiche devono essere apportate al codice (e dove) per aggiungere degli operatori diversi da + e -?" e "perché queste modifiche possono causare dei problemi?"

   Gli smell derivano dal fatto che: 1) abbiamo diversi "if" in cascata e 2) abbiamo due cloni di codice che in qualche modo sono accoppiati quando dobbiamo eseguire una modifica del codice (co-change coupling). L'aggiunta di un nuovo operatore (es. "*") implica la modifica di entrambi i metodi. Questo può portare ad errori se la modifica viene fatta solo ad un metodo o ad entrambi i metodi ma in modo diverso. Inoltre molti if in cascata complicano la fase di testing e rendono il codice difficile da comprendere.

6) Che refactoring, tra quelli visti a lezione, applicheresti per migliorare la classe Node (si intende la versione corretta della classe)?

   Il problema della classe Node è che sono state utilizzate delle istruzioni condizionali quando invece era meglio ragionare in termini di polimorfismo. Pertanto il refactoring da applicare è il "replace conditional with polimorphism"

7) Abbozzare il codice risultante dopo che è stata eseguita l'operazione di refactoring (se necessario utilizzare anche un class diagram)

```
public interface Node {
	int evaluate();
	String toString();
}

public class ValueNode implements Node {

	private int value;

	public ValueNode(int value) {
		this.value = value;
	}

	public int evaluate() {
		return value;
	}

	public String toString() {
		return value + "";
	}
}

public class Addition implements Node {

	Node left, right;

	public Addition(Node left, Node right) {
		super();
		this.left = left;
		this.right = right;
	}

	public int evaluate() {
		return left.evaluate()
				+ right.evaluate();
	}

	public String toString() {
		return left.toString() + " + " + right.toString();
	}
}

public class Subtraction implements Node {

	Node left, right;

	public Subtraction(Node left, Node right) {
		super();
		this.left = left;
```

```
            this.right = right;
        }

        public int evaluate() {
            return left.evaluate() - right.evaluate();
        }

        public String toString() {
            return left.toString() + " - "
                        + right.toString();
        }

}
```

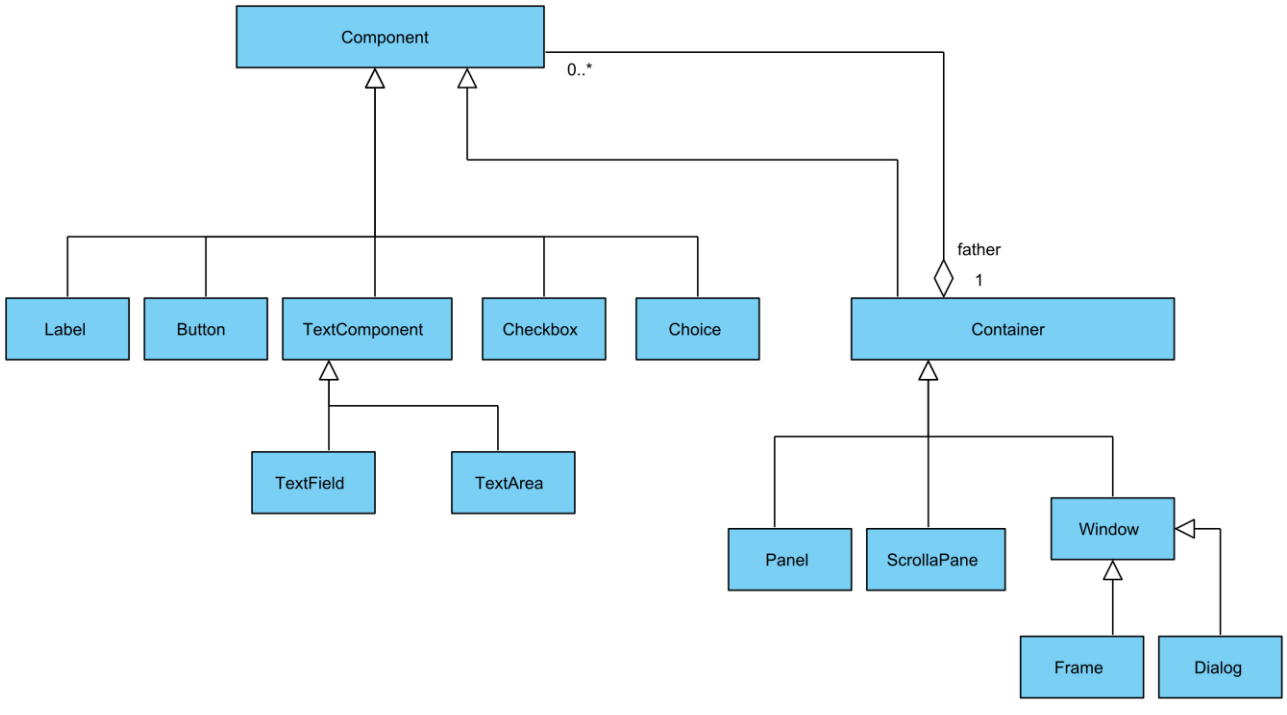
Esercizio 2

Si supponga di dover progettare un nuovo Drawing tool chiamato EasyDrawing simile ad una versione molto semplificata di Paint (sistema Window). EasyDrawing può gestire due tipi di oggetti i Container e i Component. Un Container può essere specializzato in tre tipi (Panel, ScrollPane e Window) e può contenere dei Component che sono di diversi tipi (Label, Button, TextComponent, Checkbox e Choice). Un Container può anche contenere uno o più Container. Considerare anche che: un oggetto Window può essere un Frame o un Dialog e che un TextComponent si può specializzare in TextField e TextArea.

1) Quale dei design pattern visti a lezione è adatto a rappresentare le relazioni che intercorrono tra Container e Component. Perché?

Design pattern Composite. Compone oggetti in strutture ad albero per rappresentare gerarchie tutto-parti

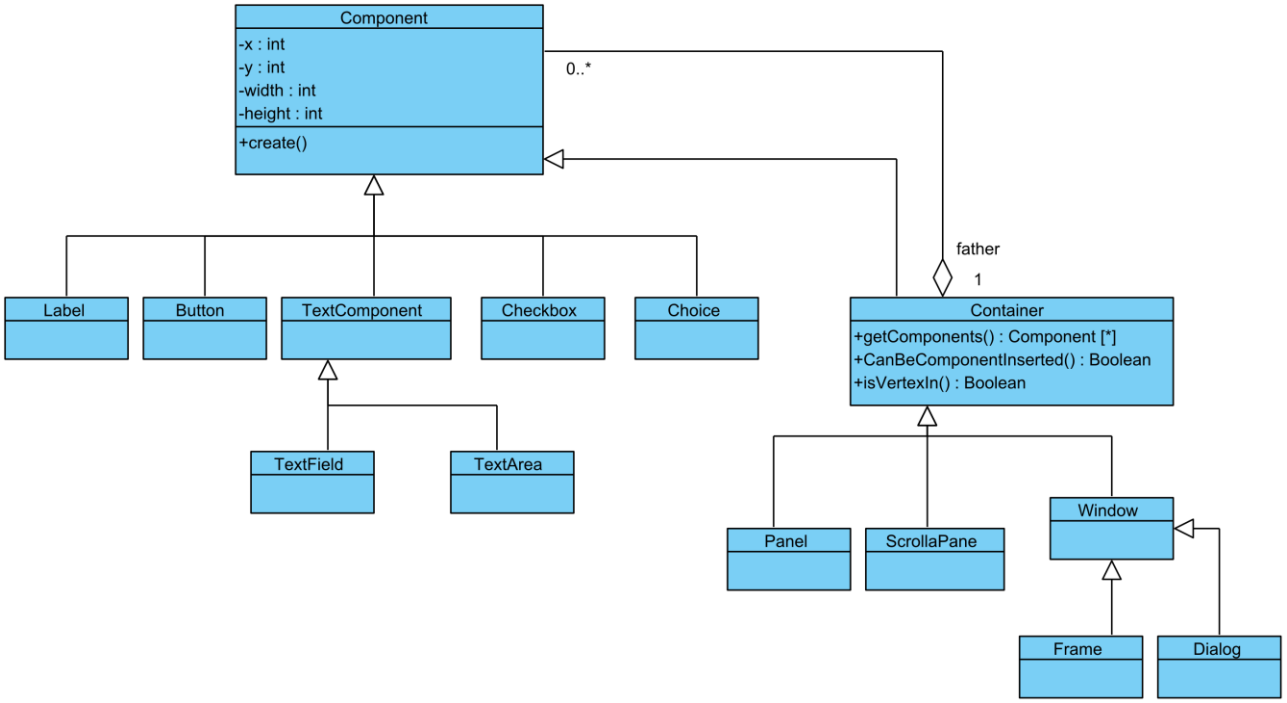
2) Rendere esplicita con un Class diagram la descrizione data sopra



3) Descrivere in pseudocodice (o con un activity diagram) l'operazione create() che crea e visualizza a video un Component; supponiamo per semplicità che la forma dei Component sia solo rettangolare. Tale operazione può inserire un Component in un Container solo se esso non si sovrappone parzialmente ad un altro Component precedentemente inserito nello stesso Container. Descrivere i parametri che vengono passati all'operazione create() e descrivere la logica che permette di rivelare eventuali sovrapposizioni. Infine viene richiesto di posizionare tale operazione nel Class diagram presentato al punto 2), eventualmente considerando casi di Overrading. Chiaramente è possibile definire operazioni ausiliarie che possono essere richiamate da create()

- **Un component è costituito da un vertice x, y (punto in alto a sinistra), width (larghezza) e height (altezza)**
- **(**1) Un componente C può essere creato solo se i suoi vertici <u>non</u> cadono all'interno di un componente già nel container oppure se nessun vertice dei componenti nel container cade all'interno di C**
- **L'operazione getComponents() contenuta in Container ritorna la lista di componenti contenuti nel container.**
- **L'operazione isVertexIn() contenuta in container prende come parametri in input un vertice e un componente e ritorna true se il vertice cade all'interno del componente, false altrimenti**
- **Il construttore crete() prima di creare un componente chiama l'operazione canBeComponetInserted() passando "se stesso" (father.cabBecomponnetInserted(this)), vedi diagramma delle classi sotto. Solo se detta funzione ritorna true il componente è creato e visualizzato a video**

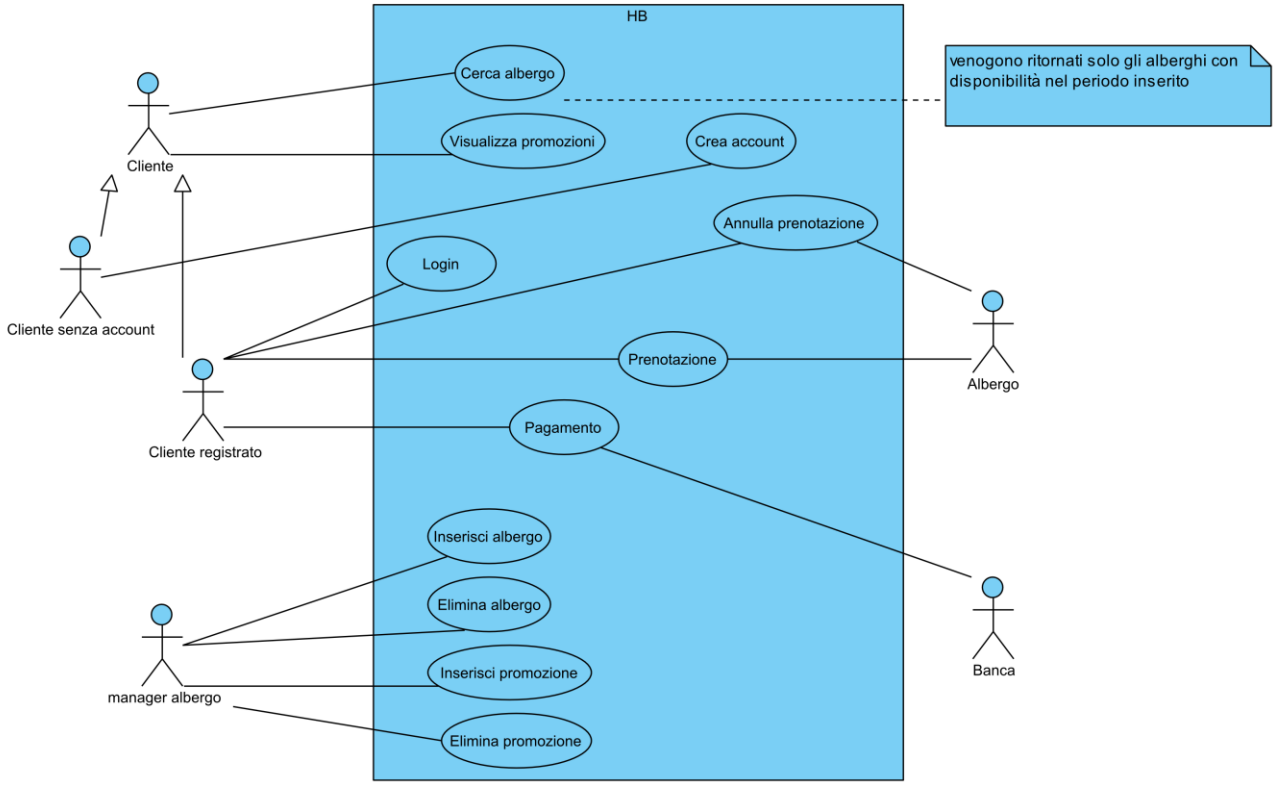
```
Boolean canBeComponentInserted(c:Component) {
   For each c' in c.getFather().getComponents() { // controllo **1
     If ( isVertexIn(c.getX(), c.getY(), c') or isVertexIn(c.getX()+c.getWidth, c.getY(), c') or
        isVertexIn(c.getX(), c.getY()+c.getHeight(), c') or isVertexIn(c.getX()+c.getWidth, c.getY()+c.getHeight(), c')
       or
       isVertexIn(c'.getX(), c'.getY(), c) or isVertexIn(c'.getX()+c'.getWidth, c'.getY(), c) or
       isVertexIn(c'.getX(), c'.getY()+c'.getHeight(), c) or isVertexIn(c'.getX()+c'.getWidth, c'.getY()+c'.getHeight(), c))
     Return false;
   }
  Return true;
}
```



Esercizio 3

Si supponga di dover progettare un nuovo **holiday rental booking website** chiamato **HB** (simile a Booking.com)**,** ovvero un applicazione Web dove è possible cercare e prenotare una stanza di un hotel in un qualsiasi luogo del globo.

1) Rappresentare con uno use case diagram gli attori e gli use case (cioè le funzionalità offerte) principali di HB. Prestare attenzione ad eventuali attori esterni.



2) Scegliere uno use case tra quelli rappresentati al punto 1) e specificarlo utilizzando il formalismo (la notazione) visto a lezione

Caso d'uso: Cerca Albergo
Breve descrizione: Il sistema individua alcuni alberghi in base ai criteri di ricerca specificati dal Cliente e li mostra al Cliente
Attori primari: Cliente
Attori secondari: nessuno
Precondizioni: nessuna
Sequenza degli eventi prinicpale:

1. **Il caso d'uso inizia quando il cliente seleziona "Cerca albergo"**
2. **Il sistema chiede al Cliente i criteri di ricerca (vedi **2)**
3. **Il Cliente inserisce i criteri di ricerca**
4. **Il sistema ricerca gli alberghi che soddisfano i criteri specificati dal cliente**
5. **Se il sistema trova uno o più alberghi**
   - **5.1 Per ogni albergo trovato**
     - **5.1.1 il sistema mostra l'immagine dell'albergo**
     - **5.1.2 il sistema mostra le caratteristiche**
     - **5.1.3 il sistema mostra il prezzo**
     - **5.1.4 il sistema mostra un bottone per effettuare la prenotazione**
6. **Altrimenti**
   - **6.1 Il sistema comunica al Cliente che non ci sono alberghi**

Postcondizioni: nessuna
Sequenze eventi alternativa: Annulla Ricerca Albergo

(2)**