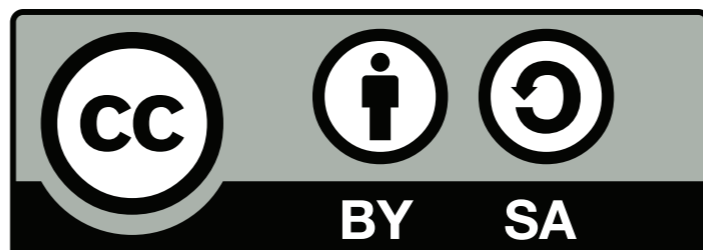


Tecnologia e Applicazioni Internet 2008/9

Lezione 0 - Modern Application Design

Matteo Vaccari

<http://matteo.vaccari.name/>
matteo.vaccari@uninsubria.it



Le vostre aspettative?

Argomenti

- Progettazione applicativa moderna
- Test unitario e funzionale di applicazioni web
- Uso del database in Java
- Java Servlet API
- JavaScript
- Ajax
- Architetture REST

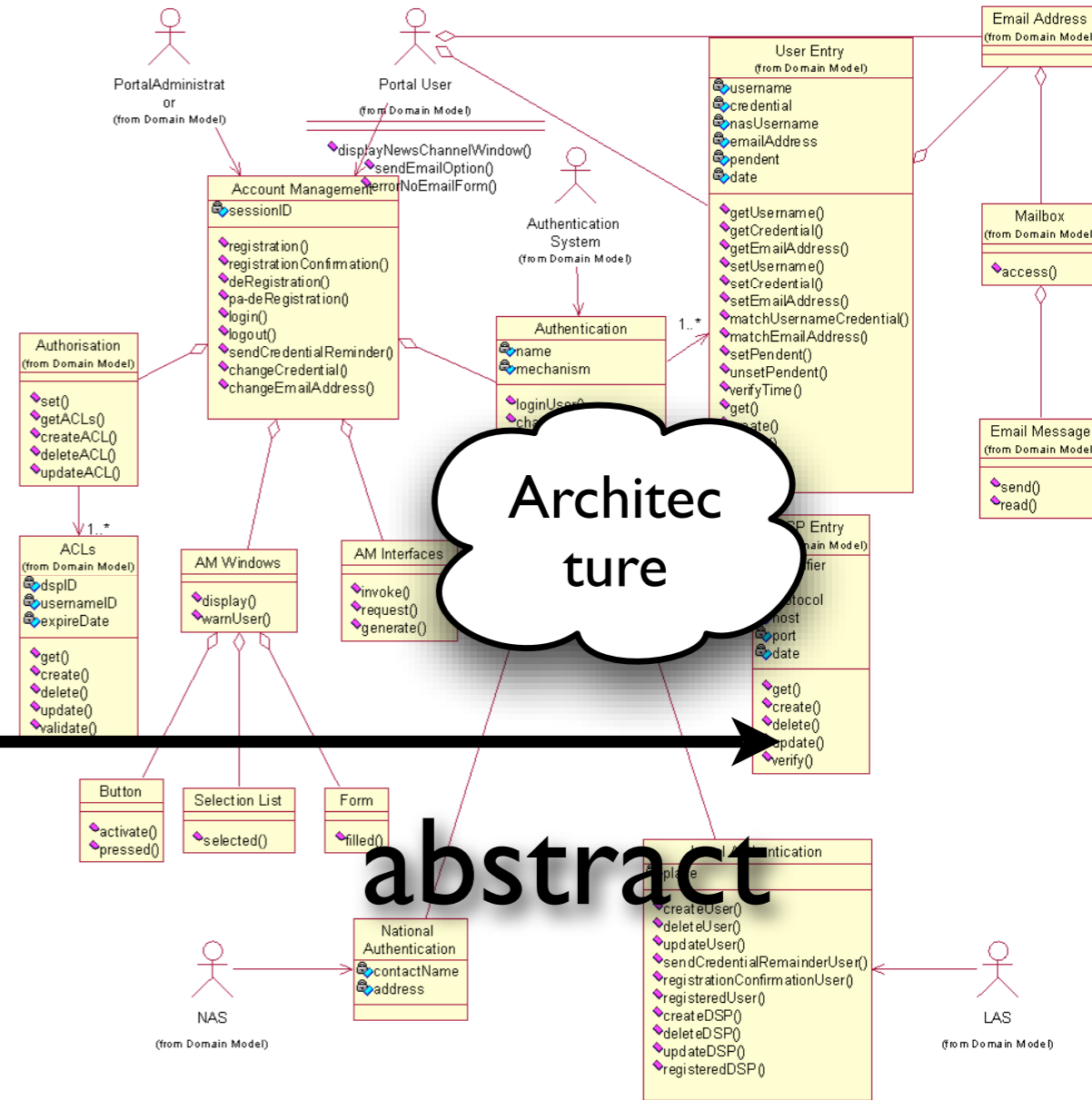
Application Design

What is design?

```
public class XpathTest {  
    @Test  
    public void testXpath() throws Exception {  
        Document doc = documentFromString("<a><b><c>foo</c></b></a>");  
        assertEquals(0, getNodeList(doc, "//foo").getLength());  
        assertEquals(1, getNodeList(doc, "//c").getLength());  
        assertEquals("foo", getNodeContent(doc, "/a/b/c"));  
    }  
  
    private String getNodeContent(Document doc, String xpath) throws XPathExpressionException {  
        NodeList nodes = getNodeList(doc, xpath);  
        assertEquals("expected exactly 1 node at xpath " + xpath, 1, nodes.getLength());  
        return nodes.item(0).getTextContent();  
    }  
  
    private NodeList getNodeList(Document doc, String xpath) throws XPathExpressionException {  
        // see http://www.w3.org/TR/xpath-20/2.2.2.html  
        XPathFactory xPathFactory = XPathFactory.newInstance();  
        XPath xp = xPathFactory.newXPath();  
        return xp.evaluate(xpath, doc, XPathConstants.NODESET);  
    }  
  
    private Document documentFromString(String string) throws Exception {  
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();  
        factory.setValidating(false);  
        factory.setNamespaceAware(true);  
        return factory.newDocumentBuilder().parse(new ByteArrayInputStream(string.getBytes()));  
    }  
  
    @Test  
    public void testTemplate() throws Exception {  
        StringTemplate template = new StringTemplate("Hello, $name$!");  
        template.setAttribute("name", "world");  
        assertEquals("Hello, world!", template.toString());  
    }  
}
```

Source code

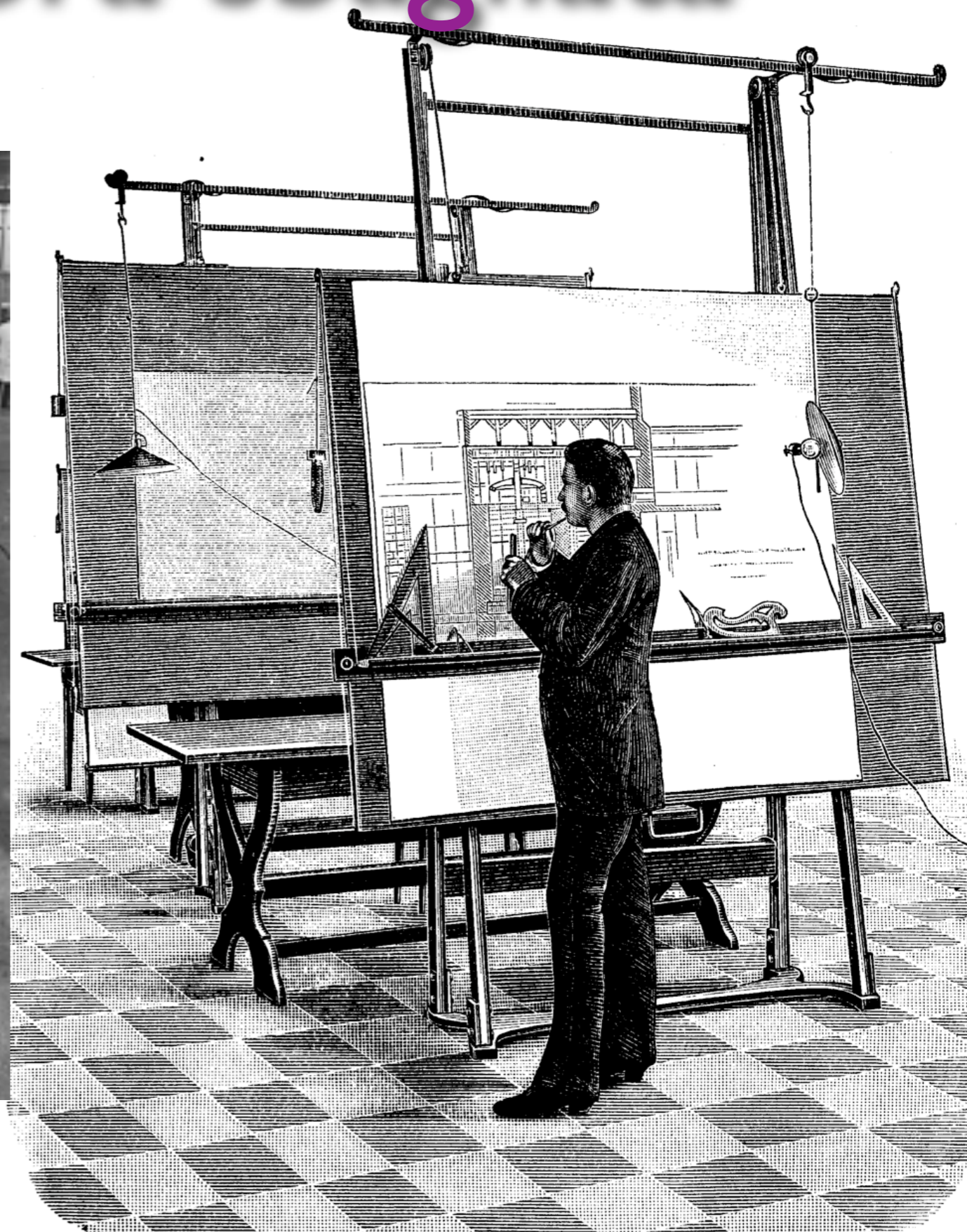
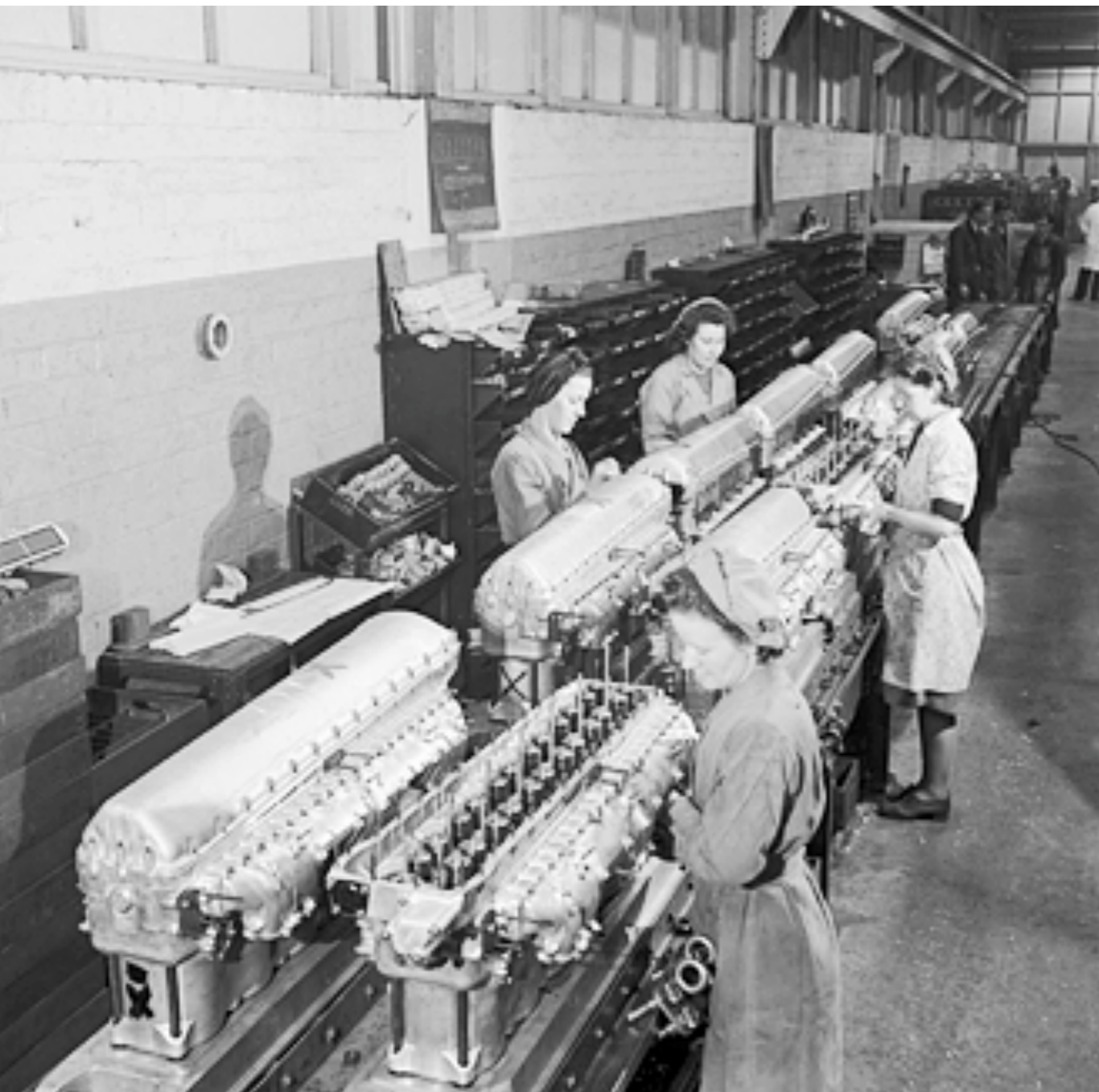
detailed



abstract

Il **design** del software è un continuo, che va dai modelli architetturali astratti fatti di diagrammi e descrizioni a parole, via via fino al design più dettagliato, che è rappresentato dal codice sorgente. Il codice sorgente è il design più preciso.

Una metafora sbagliata



La produzione di software non assomiglia alla manifattura. In particolare è pernicioso l'idea che i programmatori siano "operai" poco educati che eseguono i progetti prodotti da "architetti" più capaci. Nella produzione di software i programmatori sono **progettisti** a tutti gli effetti. Gli unici "operai" nel mondo del software sono i compilatori, che producono il codice eseguibile dai sorgenti.

SW Development != Manufacturing

```
        PreparedStatementData preparedStatementData = new PreparedStatementData(sql, pa
        executer.executeUpdateStatement(preparedStatementData);
    }

    protected static void execute(String sql) {
        execute(sql, list());
    }

    @Override
    public boolean equals(Object other) {
        return EqualsBuilder.reflectionEquals(this, other);
    }

    @Override
    public int hashCode() {
        return HashCodeBuilder.reflectionHashCode(this);
    }

    @Override
    public String toString() {
        return ReflectionToStringBuilder.reflectionToString(this);
    }

    protected void insert(String tableName, List columnNames, List parameters) {
        String sql = JdbcQueryBuilder.createInsertStatement(tableName, columnNames);
        PreparedStatementData data = new PreparedStatementData(sql, parameters);
        getExecuter().executeInsertStatement(data );
    }

    public abstract void save();
```

The most accurate *design model* of software is *the code*

Modular programming

- Decompose the system in **modules**
- Modules should have:
 - a **clear interface**
 - low **coupling**
 - high **cohesion**
 - a single reason for **change**

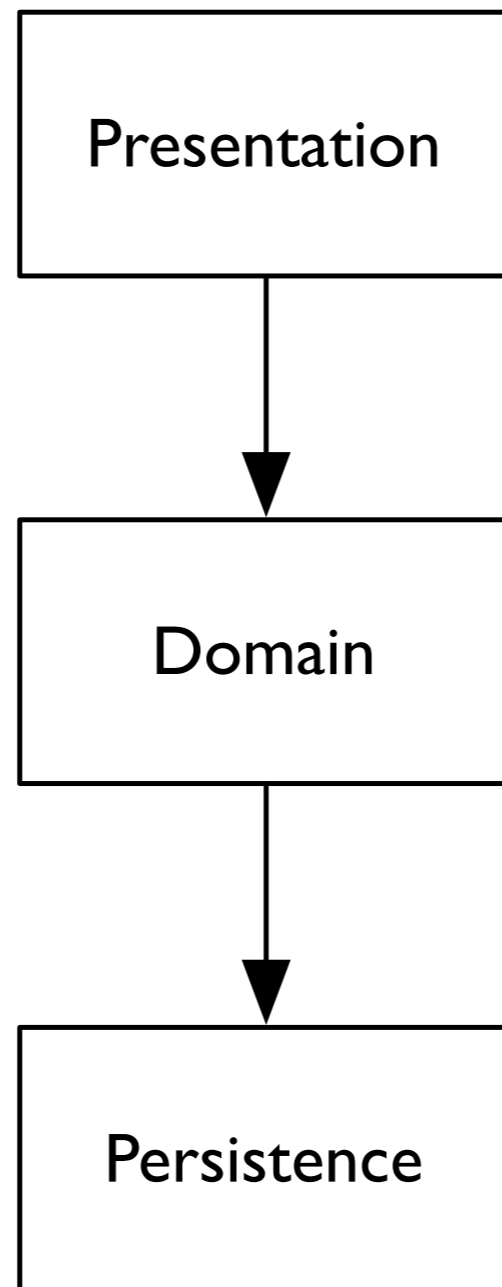
Modular programming

- Decompose the system in modules: yes but how?

*...it is almost always incorrect to begin the decomposition of a system into modules on the basis of a flowchart. We propose instead that one begins with a **list of difficult design decisions or design decisions which are likely to change**. Each module is then designed to hide such a decision from the others.*

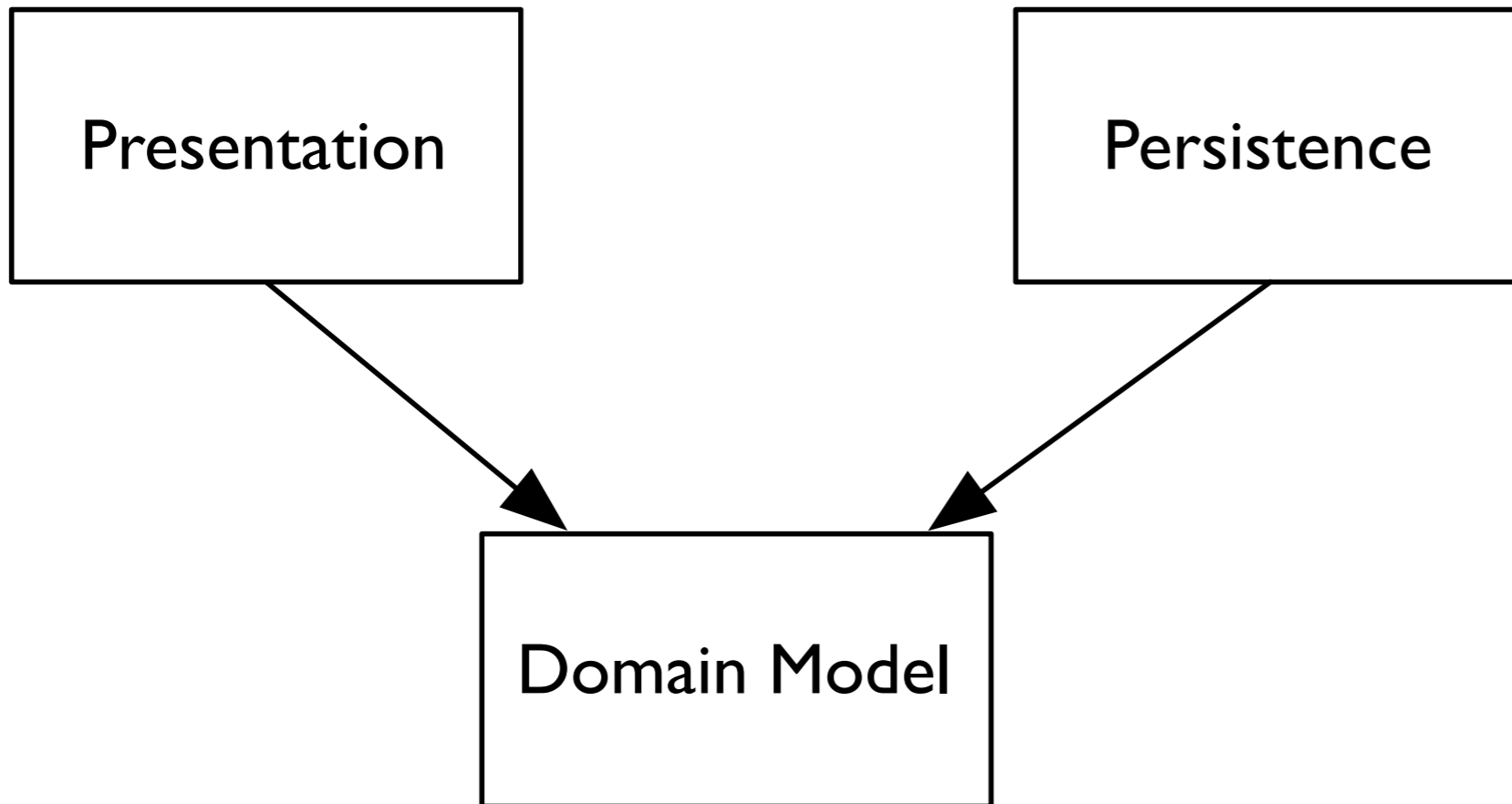
-- David Parnas, 1972

The standard three-layer architecture



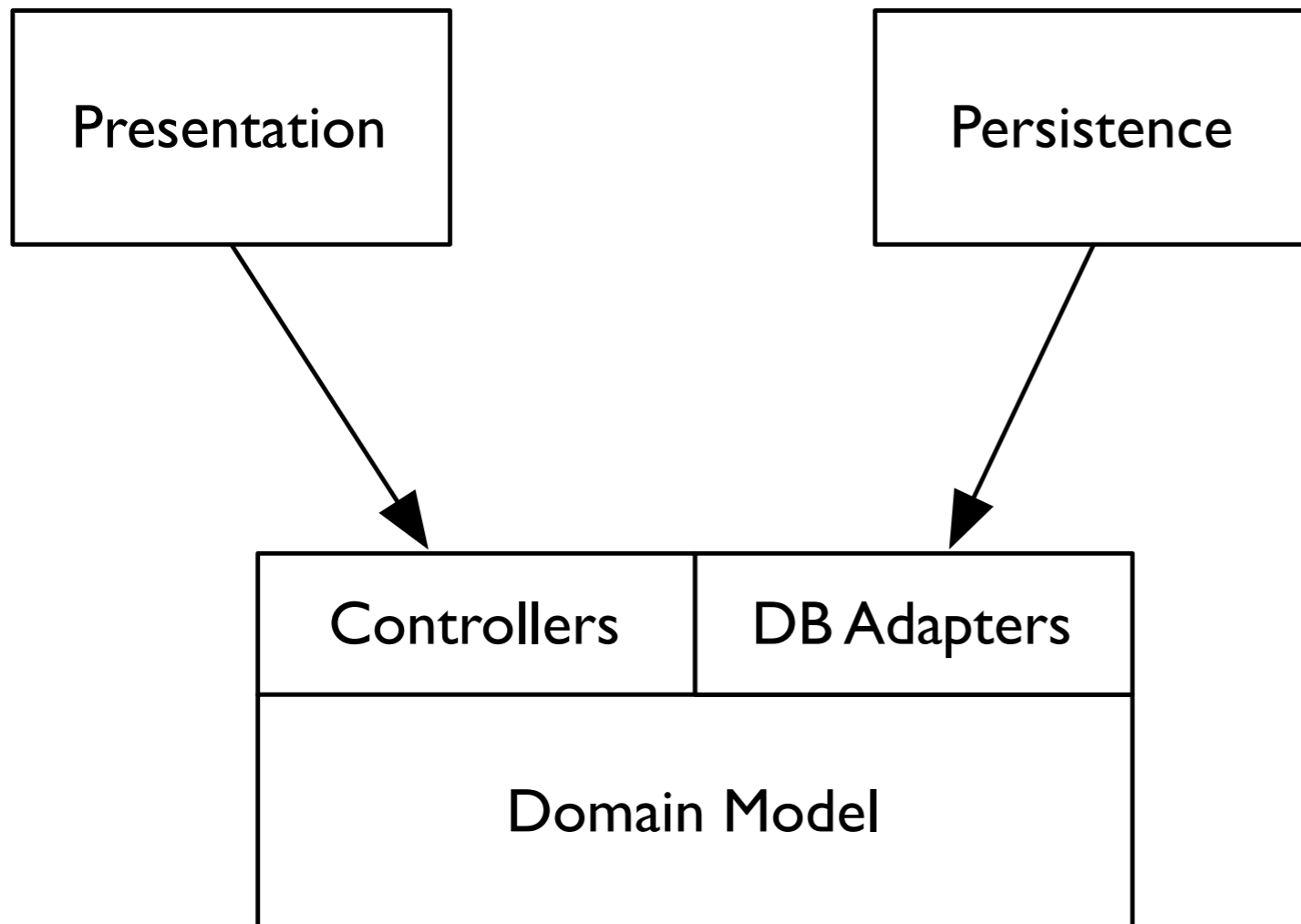
- Boxes are *layers* (sets of classes)
- Arrows mean “depends on”

A better variation



- Everything depends on the domain model
- The domain model depends on nothing

In greater detail



- Controllers are called by the presentation layers
- DB Adapters are called by the domain layer

Che cos'è il domain model?

Ma lo prence de lí laiéri, la piú magna e nobile parte de lo progetto tutto, serà lo laiéro mediano. Codesto, nomato la "logica de l'affari", contarrà tutte le parti complesse de lo sistema et sophisticate ancora. Et invero, hesso laiéro spartirà lo programmatore probbo et astuto de lo cacchione emprovisato et stolto.

“Leonardo da Vinci” (in realtà Paolo Perrotta all’Italian Agile Day 2009 :-)

Perché imparare a fare design?

*Le tecniche di programmazione istintiva (cut&paste, code&fix, ecc...) danno all'inizio del progetto una **falsa** sensazione di velocità. Perché falsa? Perché:*

*1. La velocità è destinata a **diminuire progressivamente**. Queste tecniche sono degenerative, raggiungono velocemente degli obiettivi (implementazione di funzionalità) nel breve periodo, ma degradano la qualità della base di codice, il che comporta un rallentamento durante l'implementazione della funzionalità successiva, ecc... fino al raggiungimento del collasso del codice (ovvero il momento in cui i programmatori o scappano, o si impuntano per una riscrittura dell'intero progetto)*

...

Gabriele Lana, <http://milano-xpug.pbwiki.com/Velocita>

Perché imparare a fare design?

...

2. La velocità iniziale *non può essere aumentata*. Le tecniche di cui sopra non sono fisicamente migliorabili attraverso l'esperienza, la pratica o l'impegno (forse solo leggermente). L'unico modo di aumentare la velocità è di aumentare il numero delle persone coinvolte nella scrittura del progetto; peccato che questo aumenti anche la velocità di degradazione del codice, che porta al peggioramento della situazione in breve tempo.

Gabriele Lana, <http://milano-xpug.pbwiki.com/Velocita>

Perché imparare a fare design?

*The fantasy of **cutting quality to deliver faster** lives on in part because we do not measure that long period at the end where we are taking out the critical bugs that we put in. We assume, wrongly, that that period is necessary, part of the universe. ...*

Low quality has an immediate and visible negative impact on delivering “fast enough”. That negative impact is the stabilization period. That is lost time due directly to low quality. ...

*To a first approximation, then, looking at a random project, **if we want to speed up, we must increase quality.***

Ron Jeffries, <http://is.gd/kZcp>

What is a good design?

- Simple
- No duplication
- Testable

Why testable?

- Test-driven-development is a *design* technique
- Automated tests enable refactoring

Come funziona JUnit?

Il pattern AAA

- *Arrange* creo una collezione di oggetti
- *Act*: eseguo un'operazione sull'oggetto da testare
- *Assert*: asserisco che sia successo quello che mi aspettavo

Test-Driven Development

Write a test

```
public class AdderTest {  
    @Test  
    public void testTwoPlusThree() {  
        Adder a = new Adder();  
        assertEquals(5, a.add(2, 3));  
    }  
}
```

Now it compiles

```
public class AdderTest {  
    @Test  
    public void testTwoPlusThree() {  
        Adder a = new Adder();  
        assertEquals(5, a.add(2, 3));  
    }  
}  
  
public class Adder {  
    public int add(int a, int b) { return 0; }  
}
```

Red bar!

```
public class AdderTest {  
    @Test  
    public void testTwoPlusThree() {  
        Adder a = new Adder();  
        assertEquals(5, a.add(2, 3));  
    }  
}  
  
public class Adder {  
    public int add(int a, int b) { return 0; }  
}
```

Expected 5, was 0

Do the simplest thing

```
public class AdderTest {  
    @Test  
    public void testTwoPlusThree() {  
        Adder a = new Adder();  
        assertEquals(5, a.add(2, 3));  
    }  
}  
  
public class Adder {  
    public int add(int a, int b) { return 5; }  
}
```

Refactor

```
public class AdderTest {  
    @Test  
    public void testTwoPlusThree() {  
        Adder a = new Adder();  
        assertEquals(5, a.add(2, 3));  
    }  
}
```

```
public class Adder {  
    public int add(int a, int b) { return a+b; }  
}
```



The procedure

1. Write a test

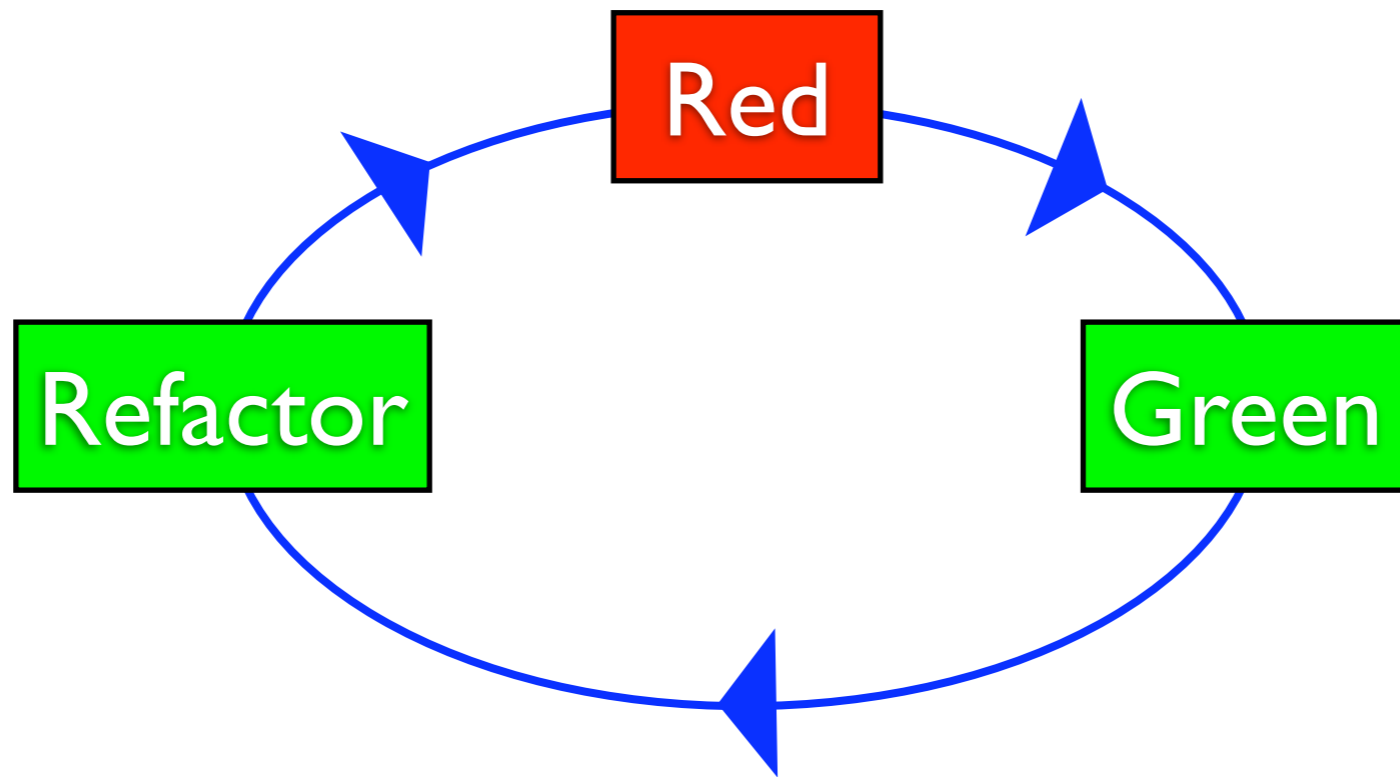
2. Make it compile



Expected 5, was 0

3. Make it pass

4. Refactor



Repeat every 2-10 min.

Clean code, why?

- *Design* is the great accelerator:
- If you drop quality for speed, *you will get neither*
- If you aim for quality...
 - ... and you know how to get it...
 - ... you will also be *fast!*

Test first, why?

- You think code from the point of view of the *caller*
- This perspective makes for better **design**
- Test coverage is a *useful byproduct*

Refactor, why?

- Because I can: the tests support refactoring
- Refactoring is when I do **design**
- I don't claim I can *guess* the right design at first
- Design **emerges**, with thought, care and small steps

Scrivere codice testabile

Presentazioni di Miško Hevery

Dependency Injection Frameworks

- Spring
- Guice
- Picocontainer

Example of Spring “Application Context”

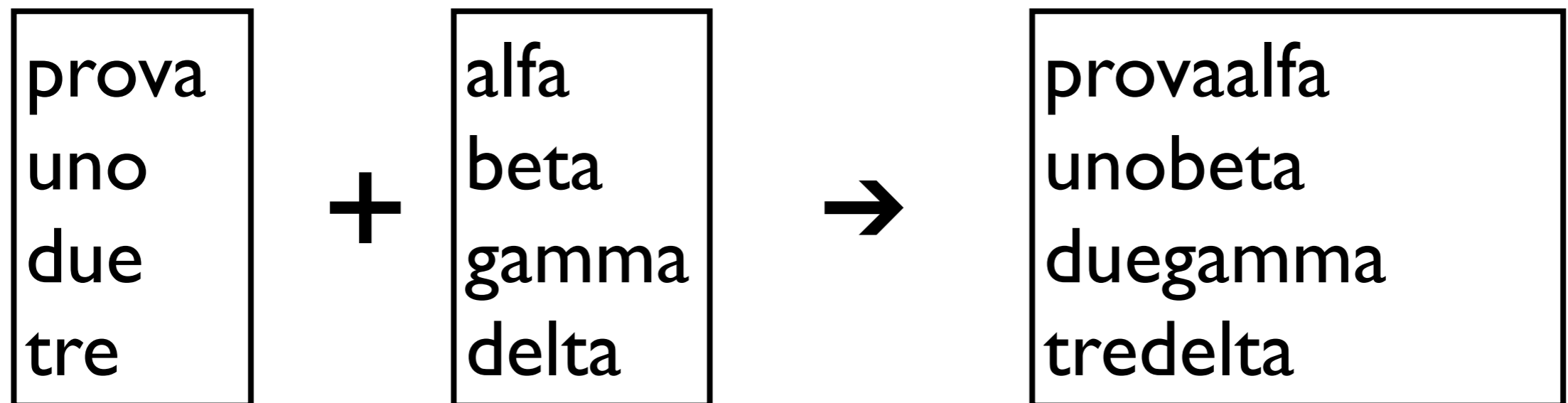
```
<beans>
  <bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource"
destroy-method="close">
  <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
  <property name="url" value="jdbc:mysql://localhost:3306/mydb"/>
  <property name="username" value="someone"/>
</bean>
<bean id="exampleDataAccessObject" class="example.ExampleDataAccessObject">
  <property name="dataSource" ref="myDataSource"/>
</bean>
<bean id="exampleBusinessObject" class="example.ExampleBusinessObject">
  <property name="dataAccessObject">
    <ref bean="exampleDataAccessObject"/>
  </property>
  <property name="exampleParam">
    <value>10</value>
  </property>
</bean>
</beans>
```

Problemi con Spring

- L'application context implementa l'antipattern "service locator"
- La "configurazione" diventa codice applicativo in XML

Esempio

Problema: incollare file riga per riga



Soluzione “veloce”

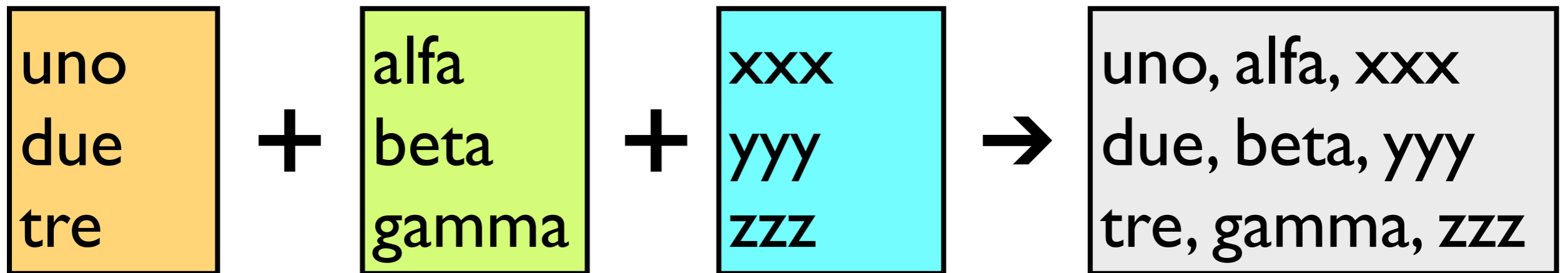
```
import java.io.*;

public class Main {

    public static void main(String ... args) throws IOException {
        BufferedReader first = new BufferedReader(new FileReader(args[0]));
        BufferedReader second = new BufferedReader(new FileReader(args[1]));
        BufferedWriter out = new BufferedWriter(new FileWriter(args[2]));
        String stringFromFirst, stringFromSecond;
        while ((stringFromFirst = first.readLine()) != null
            && (stringFromSecond = second.readLine()) != null) {
            out.append(stringFromFirst);
            out.append(stringFromSecond);
            out.newLine();
        }
        first.close();
        second.close();
        out.close();
    }
}
```

Quando abbiamo affrontato il problema anni fa, lo abbiamo risolto con una soluzione “veloce”

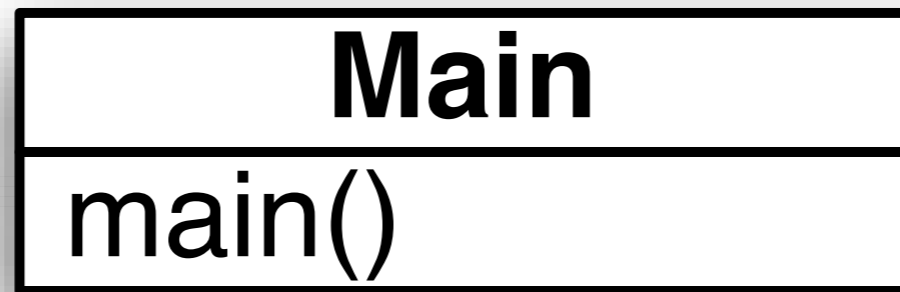
Nuovi requisiti!



- numero arbitrario di file in ingresso
- output separato da virgole

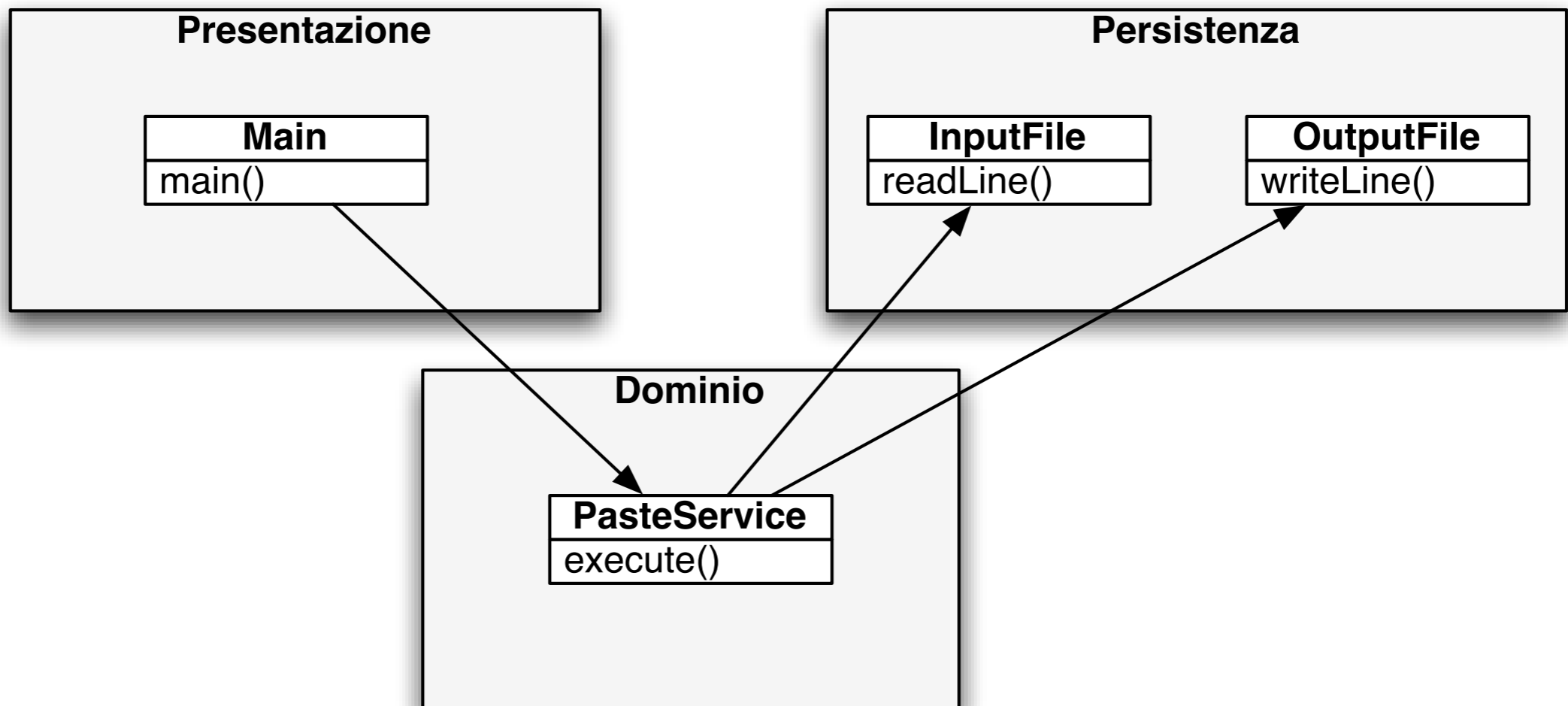
La soluzione “veloce” può avere risolto il problema originariamente. Ma quando dobbiamo, tempo dopo, risolvere un problema simile ma più complesso, adattare il vecchio software può risultare difficile.

Modello UML della versione “veloce”



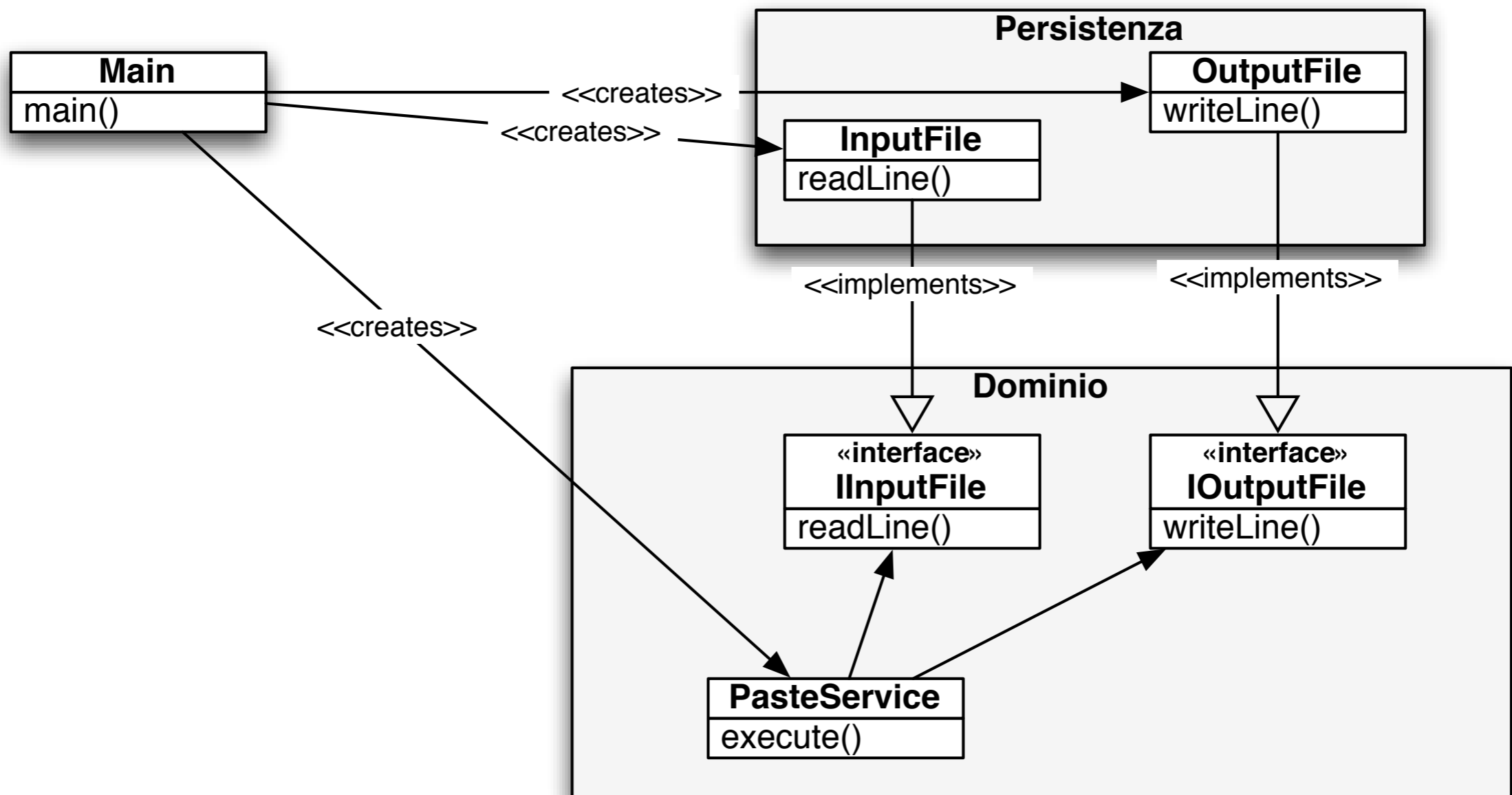
Non c'è molta struttura.

Primo tentativo di riprogettazione



Molto meglio. C'è però un problema: il PasteService invoca direttamente i metodi della persistenza. Il dominio dipende dal

Invertiamo le dipendenze



```
import java.io.IOException;

public class PasteService {

    public void execute(String firstFileName, String secondFileName, String outputFileName)
        throws IOException {

        InputFile first = new InputFile(firstFileName);
        InputFile second = new InputFile(secondFileName);
        OutputFile out = new OutputFile(outputFileName);

        String stringFromFirst, stringFromSecond;
        while ((stringFromFirst = first.readLine()) != null
            && (stringFromSecond = second.readLine()) != null) {
            out.writeLine(stringFromFirst + stringFromSecond);
        }

        first.close();
        second.close();
        out.close();
    }
}
```