

# Corso Web MVC

## Java EE – Tomcat

Emanuele Galli

[www.linkedin.com/in/egalli/](http://www.linkedin.com/in/egalli/)

# Java Enterprise Edition (Java EE)

- Prima di JDK 5 nota come J2EE
- In transizione da Oracle verso Eclipse Foundation
  - Jakarta EE <https://jakarta.ee/>
- Estende la Standard Edition (la versione corrente è basata su JDK 8) con specifiche per lo sviluppo enterprise
  - Distributed computing, web services, ...
- Applicazioni JEE sono eseguite da un reference runtime (application server o microservice)

# Apache Tomcat

- Web server che implementa parzialmente le specifiche Java EE
  - <https://tomcat.apache.org/>
- La versione 9 richiede Java SE 8 e supporta
  - Java Servlet 4.0
  - Java Server Pages 2.3
  - Java Expression Language 3.0
  - Java Web Socket 1.1
- Gestisce il ciclo di vita delle servlet, multithreading, sicurezza, ...
- Dalla shell, folder bin, eseguire lo script di startup (set JAVA\_HOME)

# Eclipse per Java EE

- Plug-in della Web Tools Platform
  - Eclipse Java EE Developer Tools
  - Eclipse JST Server Adapters
  - Eclipse Web Developer Tools
  - Maven (Java EE) integration for Eclipse WTP
  - HTML Editor
  - Eclipse XML Editors and Tools
- Java EE Perspective

# Eclipse Dynamic Web Project

- Principali setting nel wizard
  - Target runtime: Apache Tomcat 9
    - Window, Show View, Servers
    - Servers View → New → Server
  - Dynamic Web module version: 4
  - Configuration: Default
  - Generate web.xml DD (tick)
- Project Explorer
  - WebContent: HTML e JSP
  - Java Resources: Servlet
- Generazione del WAR
  - Export, WAR file

<https://github.com/egalli64/edwpot.git>

# Request – Response

- Il client manda una request al web server per una specifica risorsa
- Il web server genera una response
  - File HTML
  - Immagine, PDF, ...
  - Errore (404 not found, ...)
- Si comunica con il protocollo HTTP, di solito con i metodi GET e POST
  - GET: eventuali parametri sono passati come parte della request URL
  - POST: i parametri sono passati come message body (o “payload”)
- Associazione tra request e un nuovo thread di esecuzione della servlet

# Servlet vs JSP

- Servlet: Java puro (HTML visto come testo)

`extends HttpServlet`

`@WebServlet("/s07/timer")`

```
try (PrintWriter writer = response.getWriter()) {  
    // ...  
    writer.println("<h1>" + LocalTime.now() + "</h1>");  
    // ...  
}
```

`doGet()`

- JSP: HTML con dei frammenti Java al suo interno

scriptlet

```
<h1>  
    <%  
        out.print(LocalTime.now());  
    %>  
</h1>
```

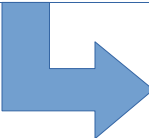
- Web client: HTML link, form o JavaScript XMLHttpRequest
  - wrapper JQuery via `ajax()` o shortcuts `load()`, `get()`, ...

# Servlet e JSP

- Servlet: gestisce l'interazione con il metodo HTTP e la logica del controller
- JSP: generazione di un documento HTML nella response

```
String user = request.getParameter("user");
Set<Character> set = new TreeSet<>();
// ...
request.setAttribute("set", set);
RequestDispatcher rd = request.getRequestDispatcher("/s08/checker.jsp");
rd.forward(request, response);
```

comunicazione  
via attributi  
nella request



```
<%
    Set<Character> set = (Set<Character>)request.getAttribute("set");
    Iterator<Character> it = set.iterator();
    while (it.hasNext()) {
        out.print(" " + it.next());
    }
%>
```



# Session

- Le connessioni HTTP sono stateless. HttpSession identifica una conversazione (cookie)

```
HttpSession session = request.getSession();
LocalTime start = (LocalTime) session.getAttribute("start");
// ...

if (start == null) {
    // ...
    session.setAttribute("start", LocalTime.now());
} // ...

if (request.getParameter("done") != null) {
    session.invalidate();
    // ... page generation with goodbye message
}

// ...
```

# Elementi JSP

direttiva

```
<%@ page contentType="text/html; charset=utf-8" pageEncoding="utf-8"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Hello JSP</title>
</head>
<body>
```

commento

```
<!-- Just as example -->
```

dichiarazione

```
<%!int unreliableCounter = 0;%>
```

```
<h1>
```

```
<%
```

scriptlet

```
out.print("Counter was " + unreliableCounter);
```

```
%>
```

```
now is
```

espressione

```
<%=++unreliableCounter%>
```

```
</h1>
```

```
<a href="..">back home</a>
```

```
</body>
```

```
</html>
```

Proprietà (!?) e metodi

Nel body del metodo  
che implementa  
JspPage.\_jspService()

Le espressioni JSP **non** sono  
terminate dal punto e virgola!  
(argomento di out.print())

# JSP useBean

servlet

```
request.setAttribute("user", new User(name, id));
```

User è  
un JavaBean

JSP script

```
<%  
    User user = (User) request.getAttribute("user");  
%>  
<%=user.getName()%>  
<%=user.getId()%>
```

page  
request  
session  
application

JSP  
standard  
action

```
<jsp:useBean id="user" class="dd.User" scope="request">  
    <jsp:setProperty name="user" property="name" value="Bob" />  
    <jsp:setProperty name="user" property="id" value="42" />  
</jsp:useBean>  
<jsp:getProperty name="user" property="name" />  
<jsp:getProperty name="user" property="id" />
```

Default values

# JSP useBean /2

http:// ... /s12/fetch.jsp?name=Tom&id=42

accesso esplicito  
ai parametri della  
request

```
<jsp:useBean id="user" class="dd.User">  
  <jsp:setProperty name="user" property="name" param="name" />  
  <jsp:setProperty name="user" property="id" param="id" />  
</jsp:useBean>
```

accesso implicito

```
<jsp:setProperty name="user" property="name" />  
<jsp:setProperty name="user" property="id" />
```

deduzione implicita

```
<jsp:setProperty name="user" property="*" />
```

```
<jsp:getProperty name="user" property="name" />  
<jsp:getProperty name="user" property="id" />
```

# JSP Expression Language

```
request.setAttribute("doc", new Document("JSP Cheatsheet", new User("Tom", 42)));
```

JavaBean aggregato come attributo nella request da servlet a JSP

```
<p>Doc title: ${doc.title}</p>  
<p>Doc user: ${doc.user.name}</p>  
<p>Doc title again: ${requestScope.doc.title}</p>
```

oggetti impliciti EL  
per gli scope

```
http:// ... /s13/direct.jsp?x=42&y=a&y=b
```

Chiamata diretta a un JSP

oggetti impliciti EL  
per i parametri

```
<p>${param.x}</p>  
<p>${paramValues.y[1]}</p>
```

pageScope  
requestScope  
sessionScope  
applicationScope

# Servlet e parametri

- Dalla request
  - `getParameter()`
    - Ritorna il valore del parametro come `String`
    - Chiamato su un array, ritorna il primo valore
  - `getParameterValues()`
    - Ritorna i valori associati al parametro come array di `String`
  - Se la request non ha quel parametro → `null`

# Servlet forward e redirect

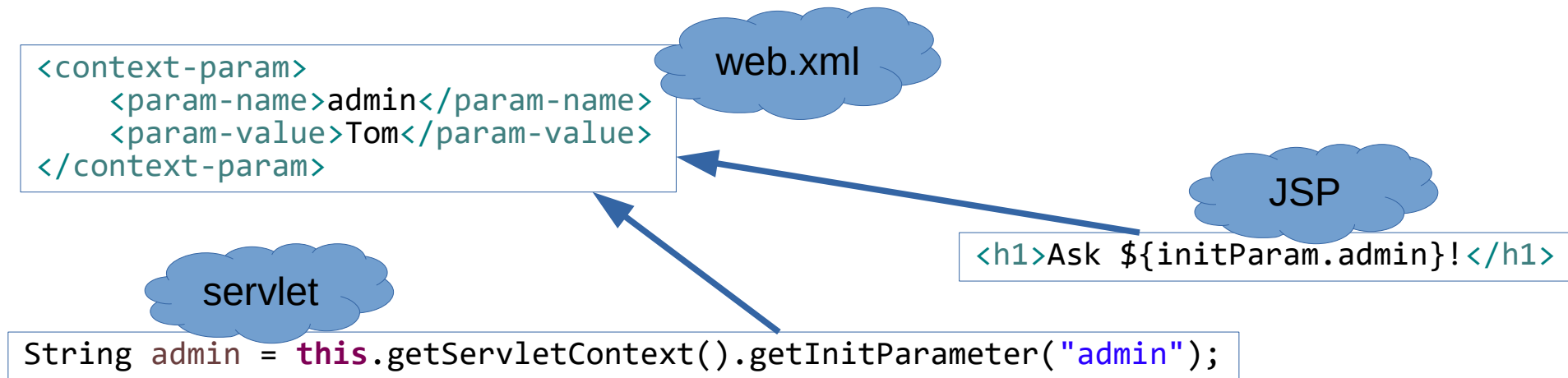
- forward()
  - Metodo di RequestDispatcher
  - getRequestDispatcher() sulla request
  - La risorsa target può essere servlet, JSP, HTML
- sendRedirect()
  - Metodo della response
  - URL tipicamente esterno al sito corrente

```
RequestDispatcher rd = request.getRequestDispatcher(destination);  
rd.forward(request, response);
```

```
response.sendRedirect("https://tomcat.apache.org/");
```

# context-param

- Parametri visibili in tutta la webapp
- Definiti in WEB-INF/web.xml





# Pagine di errore

- Nel web.xml si specifica il mapping tra tipo di errore e pagina associata

```
<error-page>
  <error-code>404</error-code>
  <location>/s17/404.jsp</location>
</error-page>
<error-page>
  <error-code>500</error-code>
  <location>/s17/500.jsp</location>
</error-page>
```

Page not found

Internal error

Da JSP si può accedere  
all'eccezione che ha causato  
l'internal error

Oggetto implicito EL

```
${pageContext.exception["class"]}
${pageContext.exception["message"]}
```

# JSTL: JSP Standard Tag Library

- Jar jstl-api e jstl nel WEB-INF/lib del progetto
- Nel JSP direttiva taglib per la libreria da usare
  - core (c), formatting (fmt), SQL (sql), functions (fn), ...

```
https://repo1.maven.org/maven2/javax/servlet/jsp/jstl/javax.servlet.jsp.jstl-api/1.2.2/  
https://repo1.maven.org/maven2/org/glassfish/web/javax.servlet.jsp.jstl/1.2.2/
```

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
```

```
<c:if test="${param.x != null}">  
  <p>Parameter x is ${param.x}</p>  
</c:if>
```

# JSTL core loop

- Su un iterable `c:forEach`, su stringa tokenizzata `c:forEachTokens`

```
User[] users = new User[] { /* ... */ };  
Double[] values = new Double[12]; // ...  
String names = "bob,tom,bill";
```

La servlet crea alcuni attributi nella request e passa il controllo a un JSP per la visualizzazione

```
<c:forEach var="user" items="${users}">  
  <p>${user.name},${user.id}</p>  
</c:forEach>
```

```
<c:forEachTokens var="token" items="${names}" delims=",">  
  <p>${token}</p>  
</c:forEachTokens>
```

```
<c:forEach var="value" items="${values}" begin="0" end="11" step="3" varStatus="status">  
  <p>  
    ${status.count}: ${value}  
    <c:if test="${status.first}">(first element)</c:if>  
    <c:if test="${status.last}">(last element)</c:if>  
    <c:if test="${not(status.first or status.last)}">(index is ${status.index})</c:if>  
  </p>  
</c:forEach>
```

# Altri JSTL core tag

- choose-when: switch (e if-else)
- out: trasforma HTML in testo semplice
- redirect: ridirezione ad un'altra pagina
- remove: elimina un attributo
- set: set di un attributo nello scope specificato
- url: generazione di URL basato sulla root

# JDBC e Tomcat con Eclipse

- Run → Run Configuration... → Apache Tomcat  
Classpath tab, User Entries, Add External JARs... (ojdbc8.jar)
- Nel folder Servers, Tomcat aggiornare **context.xml** per la resource
- Nel **web.xml** del progetto, aggiungere una reference alla resource
- Ora il **data source** è utilizzabile da servlet e JSP

```
<Resource name="jdbc/hr" type="javax.sql.DataSource" driverClassName="oracle.jdbc.OracleDriver"
auth="Container" url="jdbc:oracle:thin:@localhost:1521/orclpdb" username="hr" password="hr" />
```

```
<resource-ref>
  <res-ref-name>jdbc/hr</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
```

**JSTL sql taglib**

```
<sql:query dataSource="jdbc/hr" var="regions">
  select * from regions
</sql:query>
```

# Context lifecycle servlet listener

- Servlet chiamata all'inizializzazione e distruzione della web app
- `@WebListener` implements `ServletContextListener`
  - `void contextInitialized(ServletContextEvent sce)`
    - `sce.getServletContext().setAttribute("start", LocalTime.now());`
  - `void contextDestroyed(ServletContextEvent sce)`
    - Eventuale cleanup delle risorse allocate all'inizializzazione

```
<h1>The web app started at ${applicationScope.start}</h1>
```

# Filter

- In ingresso: audit, log, security check
- In uscita: modifica della response generata

filtro su tutte le request

O magari "\*.jsp"

```
@WebFilter(dispatcherTypes = { DispatcherType.REQUEST }, urlPatterns = { "/" })  
public class FilterAllReq implements Filter {  
    // ...  
  
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)  
        throws IOException, ServletException {  
        // ...  
        chain.doFilter(request, response);  
        // ...  
    }  
}
```

La logica può andare prima o dopo il doFilter() [IN o OUT]

# Build automation con Maven

- Build automation
  - Compilazione del codice sorgente
  - Packaging dell'eseguibile
  - Esecuzione automatica dei test
- UNIX make, Ant, Maven, Gradle
- Apache Maven, supportato da tutti i principali IDE per Java
  - pom.xml (POM: Project Object Model)
  - I processi seguono convenzioni stabilite, solo le eccezioni vanno indicate
  - Le dipendenze implicano il download automatico delle librerie richieste



# Install in Maven

- Da [maven.apache.org](http://maven.apache.org) si può scaricare Maven in formato zip (o tar.gz)
- Basta estrarre l'archivio in una directory dedicata per poter eseguire Maven: “`mvn`” in “`bin`”
- Per verificare che Maven funzioni correttamente: `mvn --version`
- Richiede Java, potrebbe essere necessaria la definizione di `JAVA_HOME`
  - Es: `set JAVA_HOME=C:\Program Files\Java\jre1.8.0_221`
- Il repository di Maven viene installato per l'utente corrente in “`.m2`”
- Si può installare un file (jar o altro) nel proprio repository di Maven. Esempio:
  - `mvn install:install-file -Dfile=/app/Administrator/product/18.0.0/dbhomeXE/jdbc/lib/ojdbc8.jar -DgroupId=com.oracle -DartifactId=jdbc -Dversion=8 -Dpackaging=jar`
  - Il risultato è che `ojdbc8.jar` viene copiato in `.m2\repository\com\oracle\jdbc\8\jdbc-8.jar`

# Nuovo progetto Maven in Eclipse

- Creare un progetto Maven
  - File, New, Project → Wizard “Maven Project”
  - È necessario specificare solo `group id` e `artifact id`
  - Il progetto risultante è per Java 5
- Nel POM specifichiamo le nostre variazioni
  - Properties
  - Dependencies
- A volte occorre forzare l'update del progetto dopo aver cambiato il POM
  - Alt-F5 (o right-click sul nome del progetto → Maven, Update project)

# Properties

- Nel elemento properties del POM si definiscono costanti
- Esempio: quali versioni usare nel progetto per
  - Java (source e target)
  - Il plugin che gestisce i jar in maven

```
<properties>  
  <maven.compiler.source>1.8</maven.compiler.source>  
  <maven.compiler.target>1.8</maven.compiler.target>  
  
  <maven-jar-plugin.version>3.1.1</maven-jar-plugin.version>  
</properties>
```

# Aggiungere una dependency

- Default (“central”) repository:
  - <https://repo.maven.apache.org/maven2>
- Ricerca di dipendenze:
  - <https://search.maven.org/>, <https://mvnrepository.com/>
- Es: JUnit
  - <https://search.maven.org/artifact/junit/junit/4.12/jar>
  - <https://search.maven.org/artifact/org.junit.jupiter/junit-jupiter-engine/5.3.2/jar>

```
<dependency>  
  <groupId>junit</groupId>  
  <artifactId>junit</artifactId>  
  <version>4.12</version>  
</dependency>
```

```
<dependency>  
  <groupId>org.junit.jupiter</groupId>  
  <artifactId>junit-jupiter-engine</artifactId>  
  <version>5.3.2</version>  
</dependency>
```

Tra le <dependencies>

Vogliamo usare Junit  
solo in test,  
perciò aggiungiamo:  
**<scope>test</scope>**

# Design pattern

- È una **soluzione** verificata a un **problema comune**
- Progettazione più flessibile e modulare
- Documentazione del codice più intuitiva
- Testi storici
  - A pattern language – Christopher Alexander – 1977
  - Using Pattern Languages for Object-Oriented Programs – Kent Beck, Ward Cunningham – 1987
  - Design Patterns – Erich Gamma et al. (GoF: Gang of Four) – 1994

# Definizione

- Nome
  - Descrive il pattern e la sua soluzione in un paio di parole
- Problema
  - Contesto e ragioni per applicare il pattern
- Soluzione
  - Elementi del design, relazioni, responsabilità e collaborazioni
- Conseguenze
  - Risultato, costi e benefici, impatto sulla flessibilità, estensibilità, portabilità del sistema
  - Possibili alternative

# Classificazione

- Scopo
  - Creazionali
    - Creazione di oggetti
  - Strutturali
    - Composizione di classi e oggetti
  - Comportamentali
    - Interazione tra oggetti o classi
    - Flusso di controllo
- Raggio d'azione
  - Classe (statico)
    - Ereditarietà
  - Oggetto (dinamico)
    - Associazione
    - Interfacce

# Creazionali

- Singleton (uno e uno solo)
- Factory method (da classi derivate)
- Abstract factory (da famiglie di classi)
- Builder (più step di costruzione)
- Prototype (clone)
- ...



# Strutturali

- Façade (un oggetto rappresenta un sistema)
- Composite (albero di oggetti semplici e composti)
- Decorator (aggiunge metodi dinamicamente)
- Adapter (adatta l'interfaccia a un'altra esigenza)
- Proxy (un oggetto rappresenta un altro oggetto)
- ...

# Comportamentali

- Mediator (interfaccia di comunicazione)
- Observer / Pub-Sub (notifica di cambiamenti di stato)
- Memento (persistenza dello stato)
- Iterator (accesso sequenziale agli elementi)
- Strategy (algoritmo incapsulato in una classe)
- ...

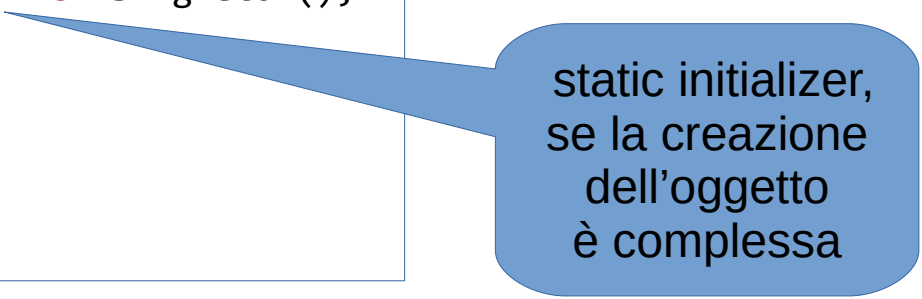
# Altri pattern

- Funzionali
  - Generator
- Concorrenza
  - Future e promise
- Architetture
  - Model View Controller (MVC)
  - Client/Server

# Singleton

- È necessario che esista un'unica istanza di una classe
- Ctor privato (o protetto), un metodo statico (factory) è responsabile dell'istanziazione e dell'accesso
- Semplice implementazione eager in Java

```
private static final Singleton instance = new Singleton();  
  
private Singleton() {  
}  
  
public static Singleton getInstance() {  
    return instance;  
}
```



static initializer,  
se la creazione  
dell'oggetto  
è complessa

# Singleton (lazy)

```
private static Lazy instance;  
private Lazy() {}  
public static synchronized Lazy getInstance() {  
    if(instance == null) {  
        instance = new Lazy();  
    }  
    return instance;  
}
```

Istanza creata solo su richiesta  
ma la sincronizzazione costa.  
Alternativa: lock tra doppio  
check su instance volatile

Instance creata alla  
prima chiamata

```
private LazyInner() {}  
private static class Helper {  
    private static final LazyInner INSTANCE = new LazyInner();  
}  
public static LazyInner getInstance() {  
    return Helper.INSTANCE;  
}
```

# Strategy

- Modifica dinamicamente un algoritmo
- Il comportamento viene delegato a un'altra classe
- Esempio Java: Comparator per sorting
  - Data una List di Integer
  - Sort con Comparator custom per ordine particolare (prima i numeri dispari e poi quelli pari)

# Strategy con Comparator

```
List<Integer> data = Arrays.asList(42, 7, 5, 12);  
data.sort(new OddFirst());  
System.out.println(data);
```

```
class OddFirst implements Comparator<Integer> {  
    @Override  
    public int compare(Integer lhs, Integer rhs) {  
        if (lhs % 2 == 1 && rhs % 2 == 0) {  
            return -1;  
        }  
        if (lhs % 2 == 0 && rhs % 2 == 1) {  
            return 1;  
        }  
        return lhs.compareTo(rhs);  
    }  
}
```

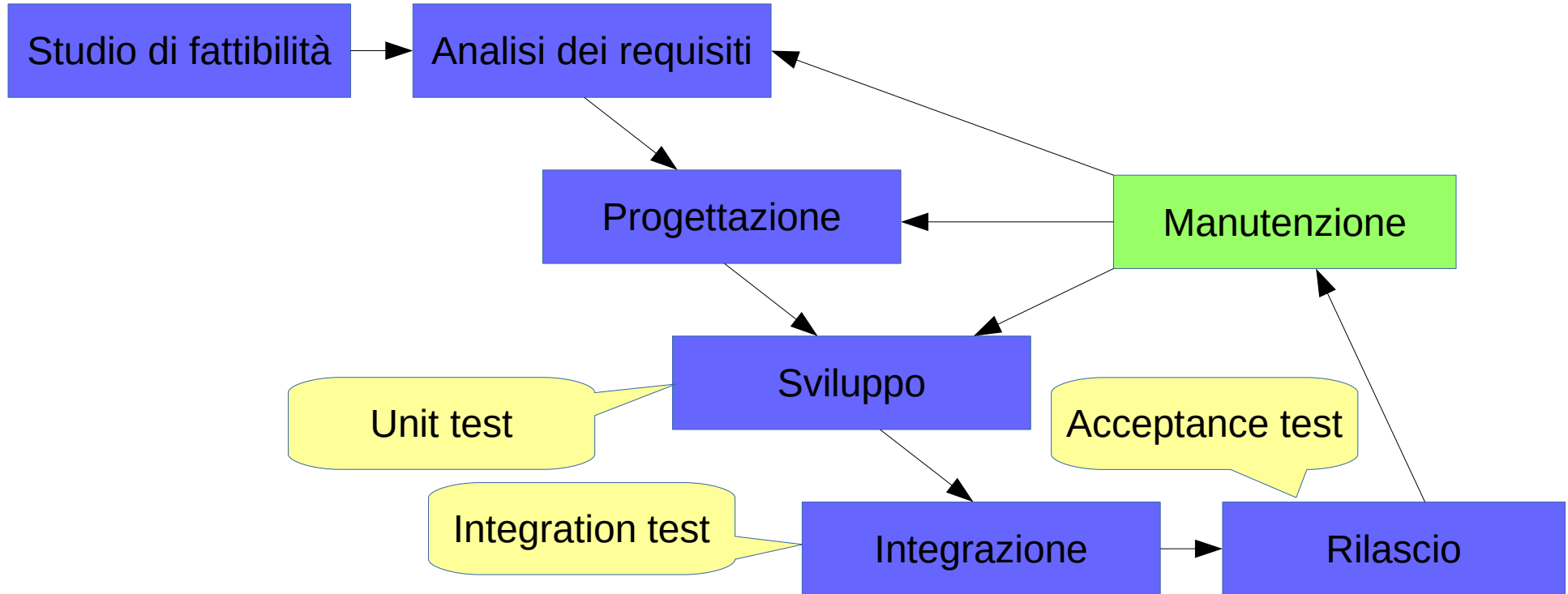
# Ciclo di vita del software

Come gestire la complessità di un progetto?

- Divide et impera
- Struttura
- Documentazione
- Milestones
- Comunicazione e interazione tra partecipanti



# Modello a cascata (waterfall)



# Modello agile

