MODEL-VIEW PATTERNS

INGEGNERIA DEL SOFTWARE

Università degli Studi di Padova

Dipartimento di Matematica

Corso di Laurea in Informatica, A.A. 2017 – 2018

rcardin@math.unipd.it

MODEL VIEW CONTROLLER

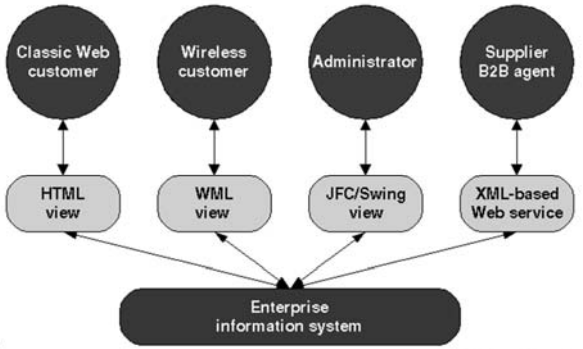
Relazioni tra		Campo di applicazione		
		Creational (5)	Structural (7)	Behavioral (11)
	Class	Factory method	Adapter (Class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter(Object) Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsability Command Iterator Mediator Memento Observer State Strategy Visitor
	Architetturali			
	Model view *			

INTRODUZIONE E CONTESTO

- Pattern architetturale
  - Inizialmente utilizzato per GUI Smalltalk-80
    - … ora *pattern* base dell'architettura J2EE, .NET, RoR ...
    - ... e dei maggiori *framework* JS: AngularJS, BackboneJS, ...
- Contesto d'utilizzo
  - Applicazioni che devono presentare attraverso una UI un insieme di informazioni
    - Le informazioni devono essere costantemente aggiornate
  - *Separation of concerns*
    - Le persone responsabili dello sviluppo hanno compentenze differenti

PROBLEMA

- Supporto a diverse tipologie di utenti con diverse interfacce
  - Rischio di duplicazione del codice ("*cut and paste*")

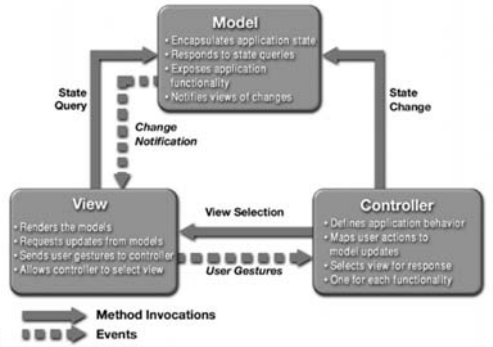


Necessità

- Accesso ai dati attraverso "viste" differenti
  - Ad esempio: HTML/Js, JSP, XML, JSON…
- I dati devono poter essere modificati attraverso interazioni differenti con i client
  - Ad esempio: messaggi SOAP, richieste HTTP, …
- Il supporto a diverse viste non deve influire sulle componenti che forniscono le funzionalità base.

Soluzione e Struttura

- Disaccoppiamento (*separation of concerns*)
  - ***Model***: dati di *business* e regole di accesso
  - ***View***: rappresentazione grafica
  - ***Controller***: reazione della UI agli *input* utente (*application logic*)



Soluzione e Struttura

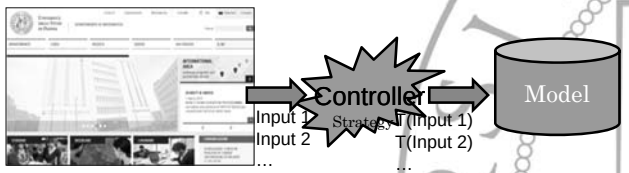
- *Model*
  - Definisce il modello dati
    - Realizza la ***business logic***
    - Dati e le operazioni su questi
  - Progettato mediante tecniche *object oriented*
    - Design pattern
  - Notifica alla *view* aggiornamenti del modello dati
    - *Observer pattern*
    - *View* deve visualizzare sempre dati aggiornati!

Soluzione e Struttura

- *View*
  - Gestisce la logica di presentazione verso i vari utenti
    - Metodi di interazione con l'applicazione
  - Cattura gli input utente e delega al *controller* l'elaborazione
  - Aggiornamento
    - "***push model***"
      - La view deve essere costantemente aggiornata
        - Utilizzo *design pattern* Observer
        - MVC in un solo ambiente di esecuzione (i.e. Javascript)
    - "***pull model***"
      - La view richiede aggiornamenti solo quando è opportuno
      - MVC su diversi ambienti di esecuzione
        - Strategia JEE (JSP, Servlet) classico, Spring, Play!, …

Soluzione e Struttura

- *Controller*
  - Trasforma le interazioni dell'utente (*view*) in azioni sui dati (*model)*
    - Realizza l'***application logic***
    - Esiste un *Controller* per ogni *View*
    - *Design patten* Strategy
      - Modifica degli algoritmi che permettono l'interazione utente con il *model*.

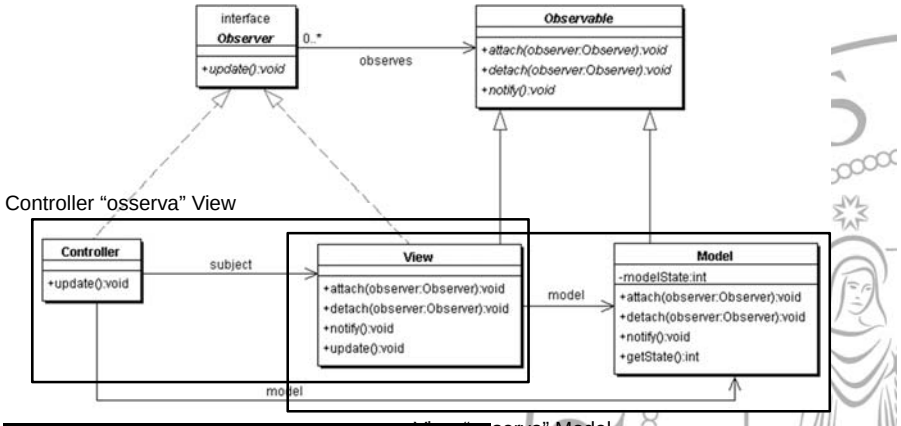


Strategie

- Nativo (*push model*)
  - *Web based (single page application)*
    - *View*: Javascript e *template*
    - *Controller*: Javascript *(routing)*
    - *Model:* Javascript
      - Sincronizzazione con *backend* tramite API REST/SOAP
- *Web based (server, pull model)*
  - *View*: JSP, ASP, ...
  - *Controller*: Servlet
    - Una sola *servlet* come controller (*Front Controller pattern*)
  - *Model*: EJB / Hibernate / MyBatis

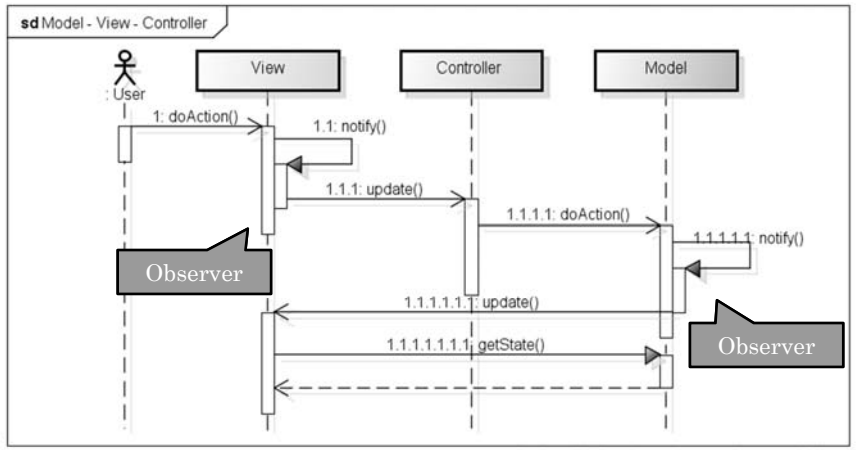
Collaborazioni

- *Push model*



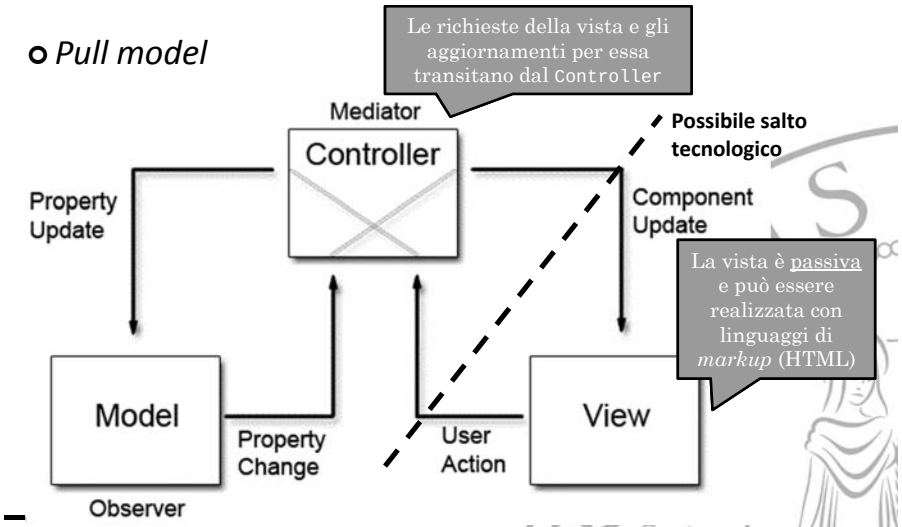
Collaborazioni

- *Push model*



COLLABORAZIONI

- *Pull model*



CONSEGUENZE

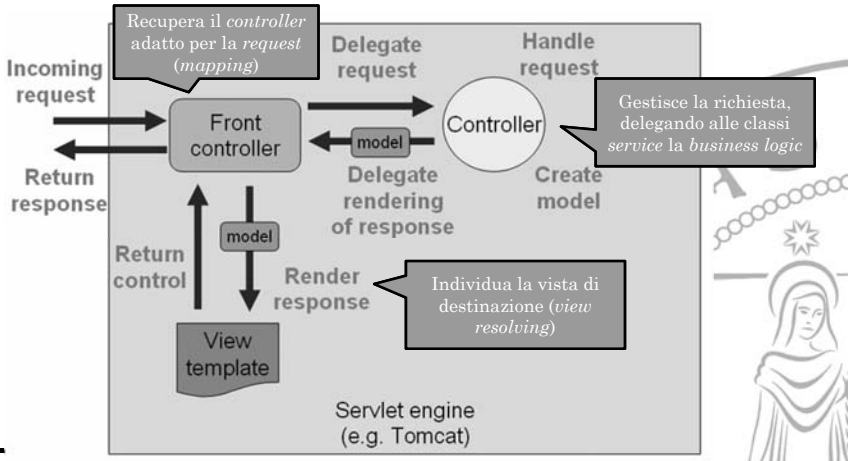
- Riuso dei componenti dei *model*
  - Riutilizzo dello stesso *model* da parte di differenti *view*
  - Miglior manutenzione e processo di test
- Supporto più semplice per nuovi tipi di *client*
  - Creazione nuova *view* e *controller*
- Maggiore complessità di progettazione
  - Introduzione molte classi per garantire la separazione

ESEMPIO PULL MODEL: SPRING MVC

- Componente per lo sviluppo di applicazione *web*
  - *Model*
    - *Service classes: layer* della logica di *business* del sistema
  - *View*
    - *Layer* di visualizzazione/presentazione dati
      - Utilizza la tecnologia JSP e *Tag library*
  - *Controller*
    - *Layer* che gestisce/controlla flussi e comunicazioni
      - *Dispatcher* delle richieste (**Front controller**)
      - *Controller* che implementano la logica applicativa
  - *Pull model* MVC

ESEMPIO PULL MODEL: SPRING MVC

- Architettura



ESEMPIO PULL MODEL: SPRING MVC

- `org.springframework.web.servlet.DispatcherServlet`
  - Front controller
    - Recupera *controller* (handler mapping) e viste (view resolving)
  - Da configurare nel file *web.xml*

```
<servlet>
   <servlet-name>disp</servlet-name>
   <servlet-class>
      org.springframework.web.servlet.DispatcherServlet
   </servlet-class>
   <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
   <servlet-name>disp</servlet-name>
   <url-pattern>/</url-pattern>
</servlet-mapping>
```

Default servlet, non preclude alcun formato nella risposta, ma gestisce anche i contenuti statici.

  - Configurazione XML (*Web Application Context*)
    - `<servlet-name>-servlet.xml`

ESEMPIO PULL MODEL: SPRING MVC

- *Controller* e annotazioni
  - Racchiudono la logica dell'applicazione *web*
  - `DefaultAnnotationHandlerMapping`
    - *Mapping* delle richieste utilizzando `@RequestMapping`
    - Sfrutta l'*autowiring* e l'*autodiscovering* dei *bean*
    - POJO, più semplice da verificare (i.e. Mockito)

```
<beans>
   <bean id="defaultHandlerMapping"
    class="org.springframework.web.portlet.mvc.annotation.
           DefaultAnnotationHandlerMapping" />

   <mvc:annotation-driven/>
   <context:component-scanbase-
          package="com.habuma.spitter.mvc"/>
   [...]
</bean>
```

`<name>-servlet.xml`

ESEMPIO PULL MODEL: SPRING MVC

- *Controller* e annotazioni

```
@Controller
public ClassHomeController {
    // Business logic
    private SpitterService spitterService;

    @Inject
    public HomeController(SpitterService spitterService) {
       this.spitterService = spitterService;
    }

    @RequestMapping({"/","/home"})
    public String showHomePage(Map<String,Object> model) {

       model.put("spittles",spitterService.getRecentSpittles(
          DEFAULT_SPITTLES_PER_PAGE));
       return "home";
    }
}
```

Dichiarazione del Controller

Injection della *business logic*

Dichiarazione URL gestiti

Modello ritornato alla view

Scelta della prossima view

ESEMPIO PULL MODEL: SPRING MVC

- `@RequestParam`
  - Permette il recupero dei parametri da una richiesta

```
@RequestMapping(value="/spittles",method=GET)
public String listSpittlesForSpitter(
   @RequestParam("spitter") String username, Model model) {

   Spitterspitter=spitterService.getSpitter(username);
   model.addAttribute(spitter);
   model.addAttribute(
       spitterService.getSpittlesForSpitter(username));
   return "spittles/list";
}
```

- `org.springframework.ui.Model`
  - Mappa di stringhe – oggetti
  - *Convention over configuration* (CoC)
  - Da utilizzare con Controller annotati

Esempio Pull Model: Spring MVC

- Componente *View*
  - Scelte da un risolutore (*resolver*) secondo il tipo di ritorno del metodo del *Controller*
    - `XmlViewResolver` Usa un file di configurazione xml per la risoluzione delle view
    - `ResourceBundleViewResolver` Usa un resource bundle (una serie di file con estensione *.properties*) per risolvere le view
    - `UrlBasedViewResolver` Esegue una risoluzione diretta del nome simbolico della view in una URL
    - `InternalResourceViewResolver` Il nome logico viene utilizzato direttamente come nome della view.

Esempio Pull Model: Spring MVC

- Componente *View*

`InternalResourceViewResolver`

```
[...]
<bean id="viewResolver"
 class="org.springframework.web.servlet.view.InternalResourceViewResolver">
   <property name="prefix" value="/WEB-INF/jsp/" />
   <property name="suffix" value=".jsp" />
</bean>
[...]
```

`XmlViewResolver`

```
<bean class="org.springframework.web.servlet.view.XmlViewResolver">
   <property name="location">
      <value>/WEB-INF/spring-views.xml</value>
   </property>
</bean>
```

```
<beans xmlns="…">
   <bean id="WelcomePage"
         class="org.springframework.web.servlet.view.JstlView">
      <property name="url" value="/WEB-INF/pages/WelcomePage.jsp" />
   </bean>
</beans>
```

`spring-views.xml`

Esempio Pull Model: Spring MVC

- Componente *View*
  - Pagina JSP (HTML + *scripting Java*)

Librerie di direttive per manipolare i *bean*

```
<%@ taglib prefix="sf" uri="http://www.springframework.org/tags/form" %>
<html>
  <head></head>
  <body>
    <div>Salve, menestrello. Inserisci di seguito il nome
         del cavaliere di cui vuoi narrare le gesta:
      <sf:form method="POST" modelAttribute="knightOfTheRoundTable">
        <sf:input path="name" size="15" />
        <sf:button>Inizia</sf:button>
       </sf:form>
    </div>
  </body>
</html>
```

Nome del *bean* che il Controller deve gestire. Il valore viene inserito nell'attributo «name»

  - Utilizzo di librerie (JSTL) per la manipolazione dei *bean*
  - Il *server* compila la pagina *(servlet)* in HTML semplice

Esempio Push Model: Backbone

- Componente *Model*
  - Dati di *business* (anche aggregati → *collection*)
    - `Backbone.Collection`
  - Notifica i propri osservatori delle modifiche

```
var Photo = Backbone.Model.extend({

   // Default attributes for the photo
   defaults: {
     src: "placeholder.jpg",
     caption: "A default image",
     viewed: false
   },

   // Ensure that each photo created has an `src`.
   initialize: function() {
      this.set( { "src": this.defaults.src} );
   }
});
```

Modello dati semplice

Costruttore

Esempio Push Model: Backbone

- Componente *View*

Constructor injection

```
var buildPhotoView = function ( photoModel, photoController ) {
  // ...
  var render = function () {
    photoEl.innerHTML = _.template( "#photoTemplate" , {
      src: photoModel.getSrc()
    });
  };

  photoModel.addSubscriber( render );

  photoEl.addEventListener( "click", function () {
    photoController.handleEvent( "click", photoModel );
  });
  // ...
  return {
    showView: show,
    hideView: hide
  };
};
```

Templating

Osservazione **modello** e comunicazione attiva con ***controller***

Operazioni esposte dalla vista (*module pattern*)

Esempio Push Model: Backbone

- Componente *Controller*
  - **Router**: collante tra View e Model
    - Inoltre instradano l'applicazione fra le diverse viste

```
var PhotoRouter = Backbone.Router.extend({
  // Handles a specific URL with a specific function
  routes: { "photos/:id": "route" },

  // Function specification
  route: function(id) {
    // Retrieving information from model
    var item = photoCollection.get(id);
    // Giving such information to view
    var view = new PhotoView({ model: item });
    something.html( view.render().el );
  }
}):
```

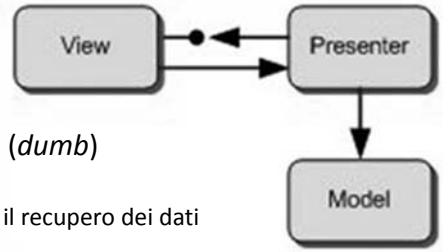
Associazione fra URL e funzioni

1. Recupera le informazioni dal modello
2. Imposta le informazioni nella vista

    - È possibile usare Controller da altre librerie

Model View Presenter

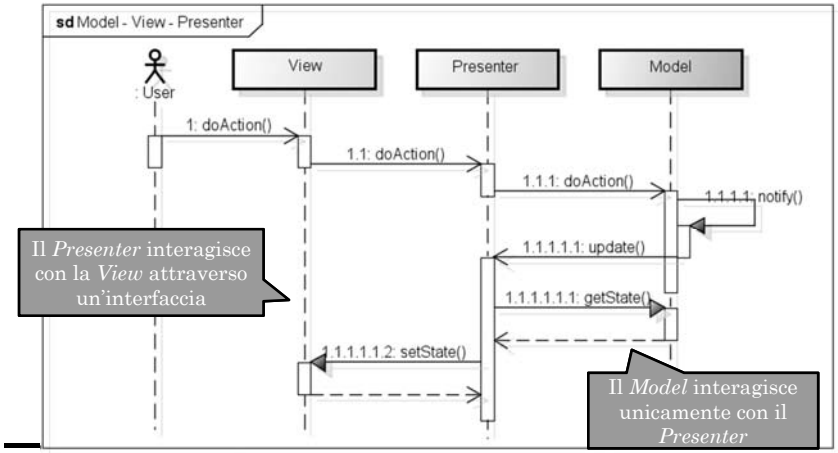
- Presenter (*passive view*)
  - *Man in the middle*
  - Osserva il modello
  - *View business logic*
  - Aggiorna e osserva la vista (*dumb*)
    - Interfaccia di comunicazione
      - Metodi *setter* e *getter* per il recupero dei dati



- View
  - Si riduce ad un *template* di visualizzazione e ad un'interfaccia di comunicazione
    - Può essere sostituita da un *mock* in fase di test
    - In Js si espone un protocollo

Model View Presenter

- *Passive View*



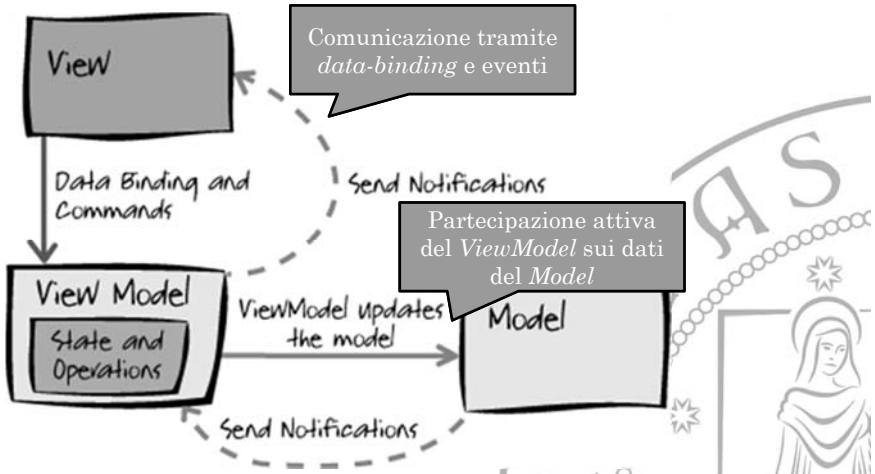
Il *Presenter* interagisce con la *View* attraverso un'interfaccia

Il *Model* interagisce unicamente con il *Presenter*

MODEL VIEW VIEWMODEL

- Separazione sviluppo UI dalla *business logic*
- ViewModel
  - Proiezione del modello per una vista
    - Solamente la validazione rimane nel modello
  - *Binding* con la vista e il modello
    - Dati e operazioni che possono essere eseguiti su una UI
- View
  - Dichiarativa (utilizzando linguaggi di *markup*)
  - *Two-way data-binding* con proprietà del ViewModel
  - Non possiede più lo stato dell'applicazione.

MODEL VIEW VIEWMODEL

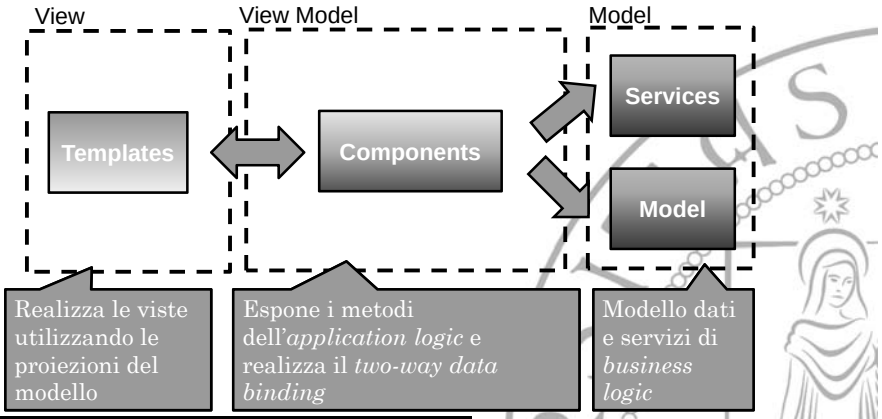


ESEMPIO MVVM: ANGULAR

- Javascript *framework*
  - *Client-side*
  - *Component based*
    - Facilita la divisione dell'applicazione in unità specializzate
  - MVVM (*two-way data binding*)
  - Utilizza HTML come linguaggio di *templating*
    - Non richiede operazioni di DOM *refresh*
      - Controlla attivamente le azioni utente, eventi del *browser*
  - *Dependence injection*
  - Fornisce ottimi strumenti di *test*
    - Jasmine (http://jasmine.github.io/)

ANGULAR

- *Model – View – View Model*



ANGULAR

- Viste e *templating*
  - Approccio dichiarativo: HTML
  - Direttive: *widget,* DOM «aumentato»
  - `Markup {{ }}`
    - Effettua il *binding* agli elementi del *view-model*
  - Tanti singoli file HTML che modellano una parte dell'applicazione

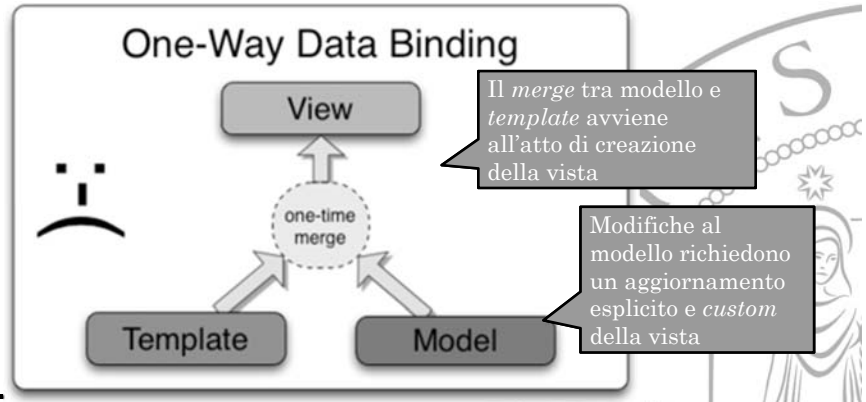
```
<ul>
  <li *ngFor="let hero of heroes" (click)="selectHero
    {{hero.name}}
  </li>
</ul>
<app-hero-detail *ngIf="selectedHero" [hero]="selectedHero">
</app-hero-detail>
```

heroes.component.html

Il codice HTML aumentato è compilato da Angular

ANGULAR

- *One-way data binding*
  - *…not the right way…*

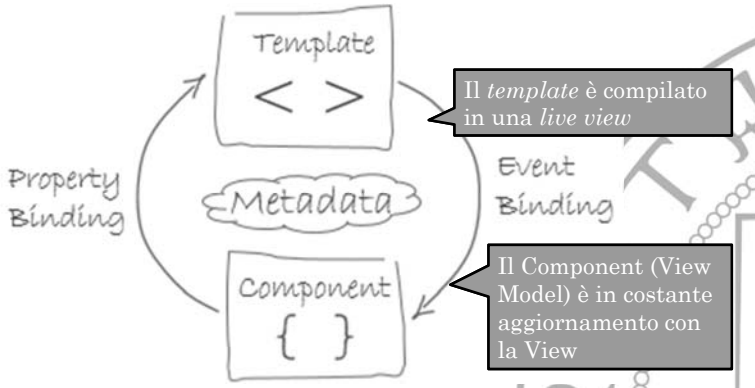


Il *merge* tra modello e *template* avviene all'atto di creazione della vista

Modifiche al modello richiedono un aggiornamento esplicito e *custom* della vista

ANGULAR

- *Two-way data binding*
  - *…the Angular way!*



Il *template* è compilato in una *live view*

Il Component (View Model) è in costante aggiornamento con la View

ANGULAR

- *Component*
  - View-Model di una vista
    - E' una semplice class Javascript marcata con `@Component`
  - Contiene l'*application logic*
    - Deve essere il più possibile snello
  - Espone dati e metodi alla vista
  - Realizza il *two-way data binding*

```
@Component({
  selector:    'app-hero-list',
  templateUrl: './hero-list.component.html',
  providers:  [ HeroService ]
})
export class HeroListComponent implements OnInit {
  /* . . . */
}
```

Elemento HTML associato al componente

Template HTML (vista)

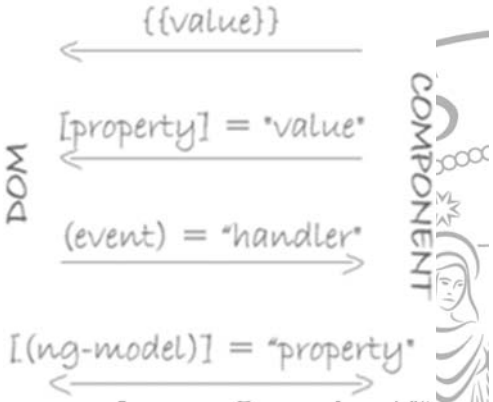
ANGULAR

- *Component*
  - Esportato come modulo (`NgModule`): visibilità

```
@Component({ /* ... */ })
export class HeroListComponent implements OnInit {
  // Properties visible from the template
  heroes: Hero[];
  selectedHero: Hero;

  // Dependency injection
  constructor(private service: HeroService) { }

  // Lifecycle hook (inversion of control)
  ngOnInit() {
    this.heroes = this.service.getHeroes();
  }

  // A method callable from the template
  selectHero(hero: Hero) { this.selectedHero = hero; }
}
```


ANGULAR

- *Data binding*
  - Il *component* fornisce un valore alla vista
  - La vista imposta un nuovo valore
  - La vista può invocare funzioni in risposta ad eventi



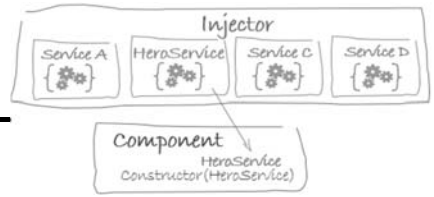
ANGULAR

- Servizi
  - Racchiudono la *business logic*
    - Richiamati dai *Component* e da altri *Service*
    - Effettuano chiamate HTTP, applicano algoritmi, ...
  - Vengono risolti utilizzando *dependency injection*

```
export class HeroService {
  private heroes: Hero[] = [];
  constructor(private backend: BackendService,  private logger: Logger) { }
  getHeroes() {
    this.backend.getAll(Hero).then( (heroes: Hero[]) => {
      this.logger.log(`Fetched ${heroes.length} heroes.`);
      this.heroes.push(...heroes); // fill cache
    });
    return this.heroes;
  }
}
```

Angular fornisce una serie di servizi standard (HTTP, logging, ...)

ANGULAR



- *Dependency injection*
  - Utilizzata per fornire ai *Component* i *Service* che necessitano
  - *Constructor injection*

```
constructor(private service: HeroService) { }
```

  - Un *provider* è il tipo dedicato a fornire un'istanza di una classe all'*injector*
    - *Injector* è un container di *Service*

```
@Component({
  selector:    'app-hero-list',
  templateUrl: './hero-list.component.html',
  providers:  [ HeroService ]
})
```

Censiti direttamente fra i metadati del *Component*

RIFERIMENTI

- Design Patterns, Elements of Reusable Object Oriented Software, GoF, 1995, Addison-Wesley
- GUI Arichitectures http://martinfowler.com/eaaDev/uiArchs.html
- MVC http://www.claudiodesio.com/ooa&d/mvc.htm
- Core J2EE MVC *design pattern* http://java.sun.com/blueprints/patterns/MVC-detailed.html
- Core J2EE *Front controller pattern* http://java.sun.com/blueprints/corej2eepatterns/Patterns/FrontController.html
- Learning Javascript Design Patterns http://addyosmani.com/resources/essentialjsdesignpatterns/book/
- Developing Backbone.js Applications http://addyosmani.github.io/backbone-fundamentals/
- Angular Fundamentals https://angular.io/guide/architecture

GITHUB REPOSITORY

https://github.com/rcardin/swe