PROGRAMMAZIONE DI SISTEMI MOBILI IOS, OBJECTIVE-C: IPHONE, IPAD, IPOD..

Prof. Paolo Nesi

Parte: 7b – Programmazione Objective C

Dipartimento di Ingegneria dell'Informazione, University of Florence
Via S. Marta 3, 50139, Firenze, Italy
tel: +39-055-4796523, fax: +39-055-4796363

DISIT Lab

http://www.disit.dinfo.unifi.it/

paolo.nesi@unifi.it

http://www.disit.dinfo.unifi.it/nesi

(derivate dalle slide di Leonardo Sequi)

Objective-C

- **Objective-C** è il principale **linguaggio di programmazione** che si utilizza per creare applicazioni per **OSX**.

- È la **base** da apprendere per utilizzare le librerie (framework) che Apple mette a disposizione e che consentono lo sviluppo di applicazioni su **OSX, iPhone, iPod Touch e iPad.**

- E' un linguaggio di **programmazione ad oggetti (OOP)** molto simile a altri linguaggi come Java o C++.

Cenni sugli oggetti

- **CLASSI** e **OGGETTI** sono elementi astratti che permettono di rappresentare oggetti reali (o immaginari) in oggetti software.

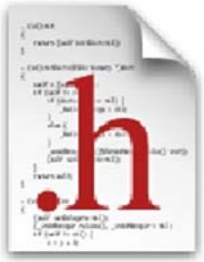
- Ogni **Oggetto** appartiene a una **Classe** e i nostri programmi saranno un **insieme di oggetti che dialogano tra loro** inviandosi messaggi e ricevendo risposte attraverso metodi e funzioni.

La prima classe

- In Objective-C per definire un **oggetto**, quindi la classe cui esso appartiene, abbiamo bisogno di due file:
  - uno (.h) che definisce **l'interfaccia** della classe
  - l'altro (.m) che ne definisce **l'implementazione**

- **L'interfaccia** descrive le azioni (i metodi e funzioni) della classe e **nasconde l'implementazione** che definisce il codice vero e proprio, ovvero ciò che le azioni realmente eseguono.

Class



#import

@interface

```
//      Person.h


@interface   Person   :   NSObject

@end
```

@implementation

```
//      Person.m

#import   "Person.h"

@implementation   Person

@end
```

Class @interface

```
#import <Foundation/Foundation.h>

@interface BankAccount : NSObject
{
    NSInteger _balance;
}

- (NSInteger) withdraw:(NSInteger)amount;
- (void) deposit:(NSInteger)amount;

@end
```

Class @interface

```
#import <Foundation/Foundation.h>

count : NSObject

lance;

- (NSInteger) withdraw:(NSInteger)amount;
- (void) deposit:(NSInteger)amount;

@end
```

Base types import

Class @interface

```
#import <Foundation/Foundation.h>

@interface BankAccount : NSObject
{

- (void) withdraw:(NSInteger)amount;
- (void) deposit:(NSInteger)amount;

@end
```

Class definition start

Class @interface

```
#import <Foundation/Foundation.h>

@interface BankAccount : NSObject
{
    NSInte
}

- (NSInteger) withdraw:(NSInteger)amount;
- (void) deposit:(NSInteger)amount;

@end
```

class name

Class @interface

```
#import <Foundat...>

@interface BankAccount : NSObject
{
    NSInteger _balan...
}

- (NSInteger) withdraw:(NSInteger)amount;
- (void) deposit:(NSInteger)amount;

@end
```

extends

parent class

Class @interface

```
#import <Foundation/Foundation.h>

@interface BankAccount : NSObject
{
    NSInteger _balance;
}

- (NSInteger er)amount;
- (void) dep unt;

@end
```

Instance variables

Class @interface

```
#import <Foundation/Foundation.h>

@interface BankAccount : NSObject
{
    NSInteger _balance;
}

- (NSInteger) withdraw:(NSInteger)amount;
- (void) deposit:(NSInteger)amount;

@end
```

methods declarations

Class @interface

```
#import   <Foundation/Foundation.h>

@interface   BankAccount   :   NSObject
{
      NSInteger    balance;
```

Class definition end

```
nteger)amount;
amount;
```

@end

Class @implementation

```
#import "BankAccount.h"

@implementation BankAccount

- (id) init {
    self = [super init];
    return self;
}

- (NSInteger) withdraw:(NSInteger)amount {
    return amount;
}

- (void) deposit:(NSInteger)amount {
    _balance += amount;
}
@end
```

Class @implementation

```
#import "BankAccount.h"

@implementation BankAccount

- (id) init {
    self = [super init];
    return self;
}

- (NSInteger) withdraw:(NSInteger)amount {
    return amount;
}

- (void) deposit:(NSInteger)amount {
    _balance += amount;
}
@end
```

Interface import

Class @implementation

```
#import "BankAccount.h"

@implementation BankAccount
```

Class implementation start

```
}

- (NSInteger) withdraw:(NSInteger)amount {
    return amount;
}

- (void) deposit:(NSInteger)amount {
    _balance += amount;
}
@end
```

Class @implementation

```
#import "BankAccount.h"

@implementation BankAccount

- (id) init {
    self = [super init];
    return self;
}

- (NSInteger) withdraw:(NSInteger)amount {
    return amount;
}

- (void) deposit:(NSInteger)amount {
    _balance += amount;
}
@end
```

methods with bodies

Class @implementation

```
#import "BankAccount.h"

@implementation BankAccount

- (id) init {
    self = [super init];
    return self;
}

- (NSInteger) withdraw:(NSInteger)amount {
    return amount;
}

{
}
@end
```

Class implementation end

Instance Variable Declaration

```
@interface MyClass : NSObject
{




}
```

Instance Variable Declaration

```
@interface MyClass : NSObject
{
    @private
    // Can only be accessed by instances of MyClass
    NSInteger _privateIvar1;
    NSString *_privateIvar2;

}
```

Instance Variable Declaration

```
@interface MyClass : NSObject
{
    @private
    // Can only be accessed by instances of MyClass
    NSInteger _privateIvar1;
    NSString *_privateIvar2;

    @protected // Default
    // Can only be accessed by instances of MyClass or MyClass's subclasses
    NSInteger _protectedIvar1;
    NSString *_protectedIvar2;









}
```

Instance Variable Declaration

```
@interface MyClass : NSObject
{
    @private
    // Can only be accessed by instances of MyClass
    NSInteger _privateIvar1;
    NSString *_privateIvar2;

    @protected // Default
    // Can only be accessed by instances of MyClass or MyClass's subclasses
    NSInteger _protectedIvar1;
    NSString *_protectedIvar2;

    @package // 64-bit only
    // Can be accessed by any object in the framework in which MyClass is defined
    NSInteger _packageIvar1;
    NSString *_packageIvar2;



}
```

Instance Variable Declaration

```
@interface MyClass : NSObject
{
    @private
    // Can only be accessed by instances of MyClass
    NSInteger _privateIvar1;
    NSString *_privateIvar2;

    @protected // Default
    // Can only be accessed by instances of MyClass or MyClass's subclasses
    NSInteger _protectedIvar1;
    NSString *_protectedIvar2;

    @package // 64-bit only
    // Can be accessed by any object in the framework in which MyClass is defined
    NSInteger _packageIvar1;
    NSString *_packageIvar2;

    @public // Never use it !
    // Can be accessed by any object
    NSInteger _publicVar1;
    NSString *_publicVar2;

}
```

@class directive

- **@class** directive provides minimal information about a class.
- **@class** indicates that the name you are referencing is a class!
- The use of the @class is known as a f*orward declaration*

```
//  Rectangle.h
#import  "Shape.h"

@class  Point;

@interface  Rectangle  :  Shape

-  (Point  *)center;


@end
```

```
//  Rectangle.m
#import  "Rectangle.h"

#import  "Point.h"

@implementation  Rectangle

-  (Point  *)center  {
      //  ...
}
@end
```

Method & Message

Method Declaration

```
- (BOOL) writeToFile:(NSString *)path atomically:(BOOL)flag;
```

Method Declaration

```
- (BOOL) writeToFile:(NSString *)path atomically:(BOOL)flag;
```

Method scope

Can be either:

- **+** for a **class method**
- **-** for an **instance method**

Methods are always *public* !

"Private" methods defined in implementation

Method Declaration

```
- (BOOL) writeToFile:(NSString *)path atomically:(BOOL)flag;
```

Return type

Can be any valid data type,including:

void returns nothing

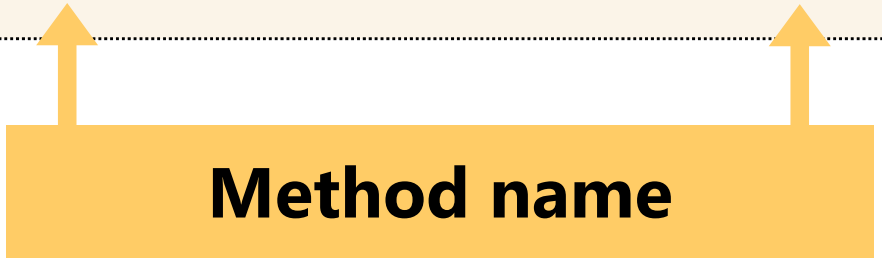
id a pointer to an object of any class

NSString * a pointer to an NSString

BOOL a boolean (YES or NO)

Method Declaration

```
- (BOOL) writeToFile:(NSString *)path atomically:(BOOL)flag;
```



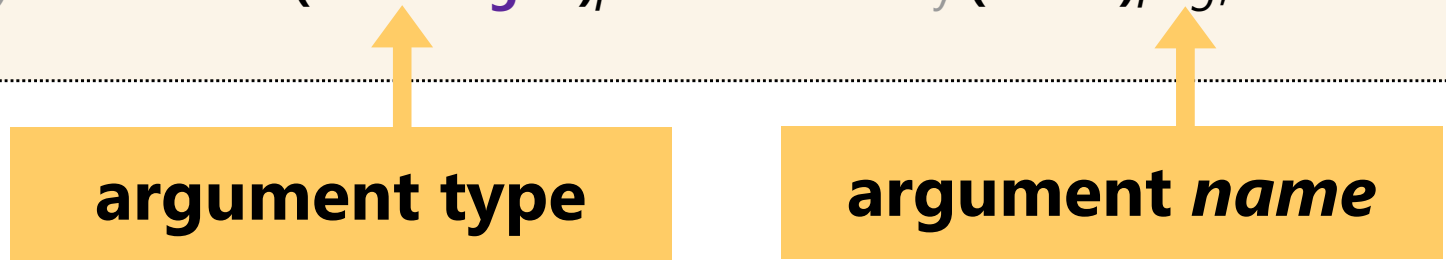
The method name is composed of all labels

Colons precede arguments,but are part of the method name

writeTofile:atomically:

Method Declaration

```
- (BOOL) writeToFile:(NSString *)path atomically:(BOOL)flag;
```



Arguments come after or within the method name

Message Passing

```
[data   writeToFile:@"/tmp/data.txt"   atomically:YES];
```

Square brackets syntax

Nested Message Passing: [[ ] [ ] [[ ]]]

```
[[store   data]   writeToFile:[@"/tmp/data.txt"   lowercaseString]
                    atomically:[[PMOption   sharedOption]   writeMode]
                       encoding:NSUTF8StringEncoding
                          error:&error];
```

Self & Super

- Methods have implicit reference to owning object called self (similar to Java and C# this,but self is a l-value)
- Additionally have access to superclass methods using super

```
- (void)viewWillAppear:(BOOL)animated  {
    [super  viewWillAppear:animated];
    [self  reloadData];
}
```

Properties

Properties

By default, the instance variables of an object are not accessible outside the block implementation of the object.

- **@public**
- **@property**

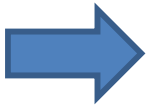
```
@property (attributes) variableType propertyName;
```

Attribute	You want it…
readwrite	When you want the property to be modifiable by people. The compiler will generate a getter and a setter for you. This is the default.
readonly	When you don't want people modifying the property. You can still change the field value backing the property, but the compiler won't generate a setter.
copy	When you want to hold onto a copy of some value instead of the value itself. For example, if you want to hold onto an array and don't want people to be able to change its contents after they set it. This sends a copy message to the value passed in, then keeps that.
retain	When you're dealing with object values. The compiler will retain the value you pass in and release the old value when a new one comes in.
assign	When you're dealing with basic types, like ints, floats, etc. The compiler just creates a setter with a simple myField = value statement. This is the default, but not usually what you want.

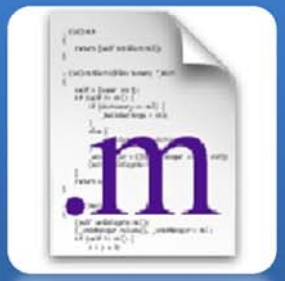
Properties

Example:

```
@property (nonatomic, copy) NSString* firstName;
```



```
- (NSString*) firstName;
- (void) setFirstName: (NSString*) firstName;
```



```
@synthesize firstName;
```

instructs the compiler to implement the methods

Object Life Cycle

Object Construction

- NSObject defines class method called **alloc**

  Dynamically allocates memory for object on the heap

  Returns new instance of receiving class

```
BankAccount *account = [BankAccount alloc];
```

- NSObject defines instance method called **init**

  Implemented by subclasses to initialize instance after memory has been allocated

  Subclasses commonly define several initializers *(default indicated in documentation)*

```
[account init];
```

- **alloc** and **init** calls are always nested into single line

```
BankAccount *account = [[BankAccount alloc] init];
```

Object Destruction

dealloc

- Never call explicitly
- Release all retained or copied instance variables *(* if notARC)*
- Calls [super dealloc] *(* if notARC)*

```
- (void)saveThis:(id)object {
    if (_instanceVariable != object ) {
        [_instanceVariable release];
        _instanceVariable = [object retain];
    }
}

- (void)dealloc {
    [_instanceVariable release];
    [super dealloc];
}
```

Memory Management

Memory Management

- Manual Reference Counting

  Higher level abstraction than malloc / free

  Straightforward approach,but must adhere to conventions and rules

- Automatic Reference Counting (ARC)

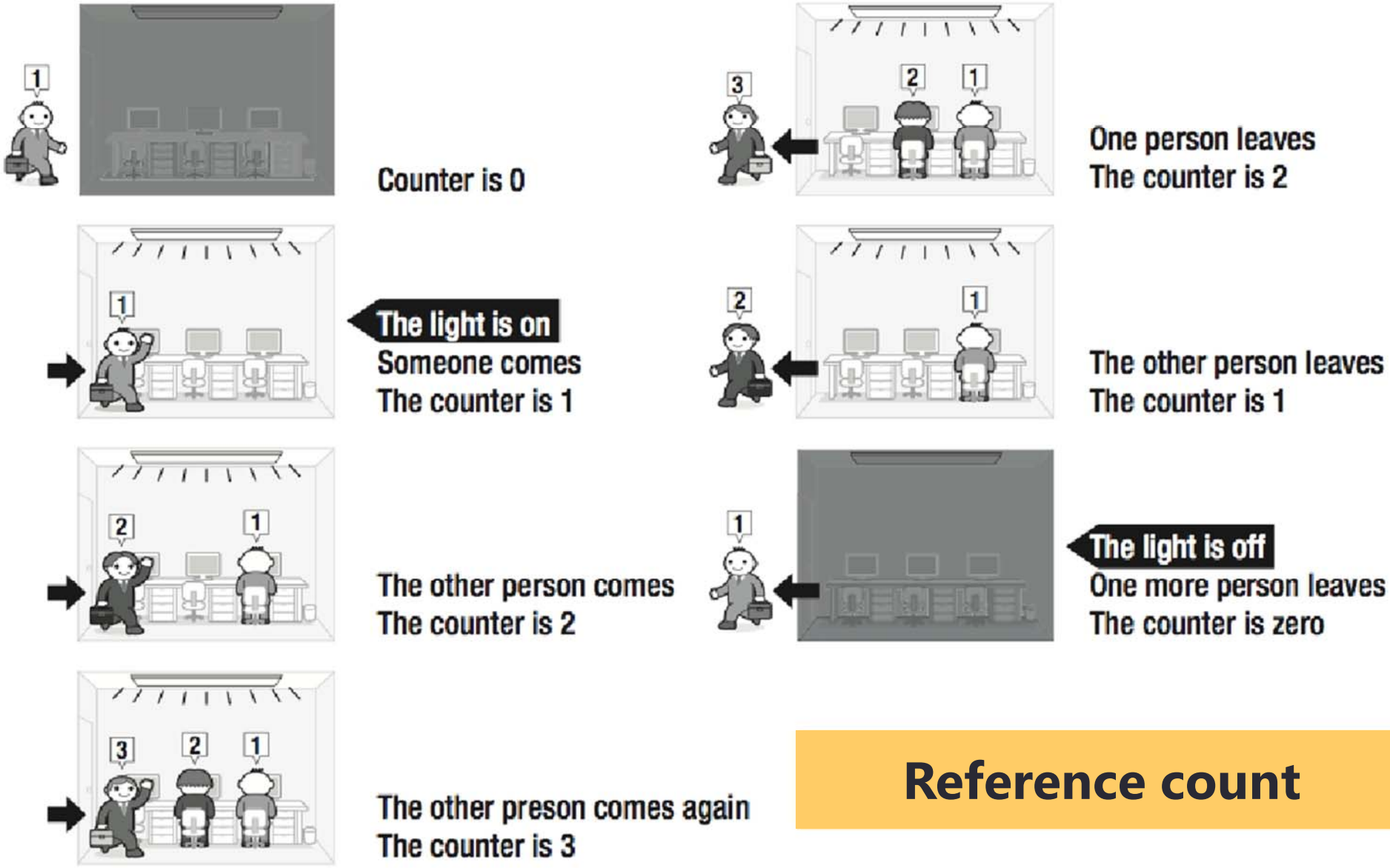
  Makes memory management the job of the compiler (and runtime)

  Available for:partially iOS 4+ or OS X 10.6+ / fully iOS 5+ or OS X 10.7+

- Garbage Collection

  Only available for OS X 10.5+, but depracated from 10.8+
  Not available on iOS due to performance concerns

Memory Management

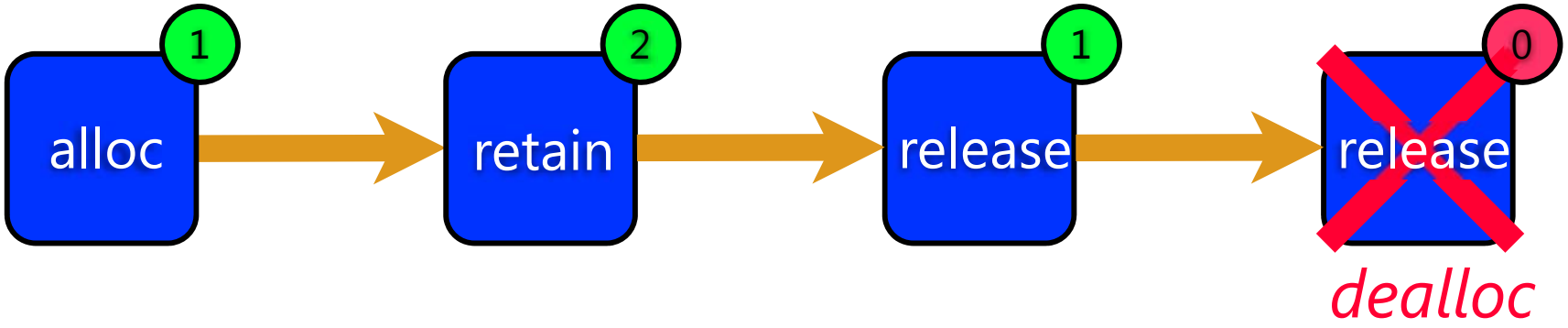


Reference count

Manual Reference Counting

(Only) Objective-C objects are reference counted:

- Objects **start** with retain count of **1** when created
- Increased with **retain**
- Decreased with **release**,**autorelease**
- When count equals **0**,*runtime* invokes ***dealloc***



Autorelease

- Instead of explicitly releasing something,you mark it for a **later** release
- An object called **autorelease pool** manages a set of objects to release when the pool is released
- Add an object to the release pool by calling **autorelease**

```
@autoreleasepool {
    // code goes here
}

NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
// code goes here
[pool release];
```

Autorelease

- Autorelease is **NOT** a Garbage Collector !
  It is deterministic ⌚
- Objects returned from methods are understood to be **autoreleased** if name is not in implicit retained set *(alloc,new,init or copy)*
- If you spawn your own **thread**,you'll have to create your own NSAutoreleasePool
- **Stack** based:autorelease pools can be nested

The Memory Management Rule

Everything that increases the reference count with alloc,copy,new or retain is in charge of the corresponding [auto]release.

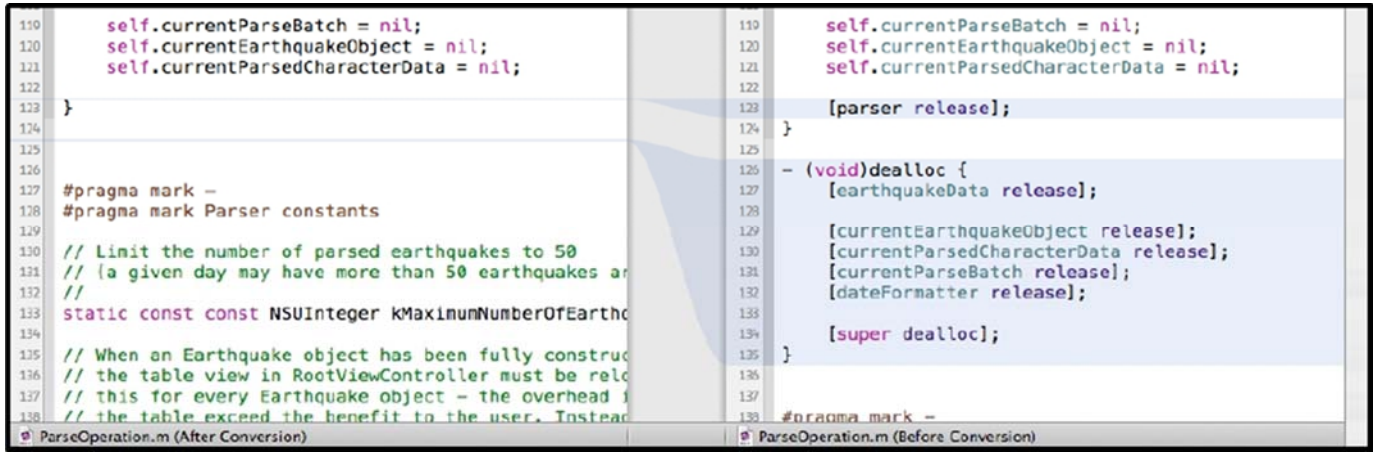
Automatic Reference Counting

"Automatic Reference Counting (ARC) in Objective-C makes memory management the job of the compiler.By enablingARC with the newApple LLVM compiler,you will never need to type retain or release again, dramatically simplifying the development process,while reducing crashes and memory leaks.The compiler has a complete understanding of your objects,and releases each object the instant it is no longer used,so apps run as fast as ever,with predictable,smooth performance."

(Apple,"iOS 5 for developers"– http://developer.apple.com/technologies/ios5)

Automatic Reference Counting

- The Rule is still valid,but it is managed by the **compiler**
- No more retain,[auto]release nor dealloc
- ARC is used in all new projects by default
- Apple provides a migration tool which is build into Xcode



Protocol

Protocol

- List of method declarations
  - Not associated with a particular class
  - Conformance,not class,is important
- Useful in defining
  - Methods that others are expected to implement
  - Declaring an interface while hiding its particular class
  - Capturing similarities among classes that aren't hierarchically related

Java / C# Interface done Objective-C style

Protocol

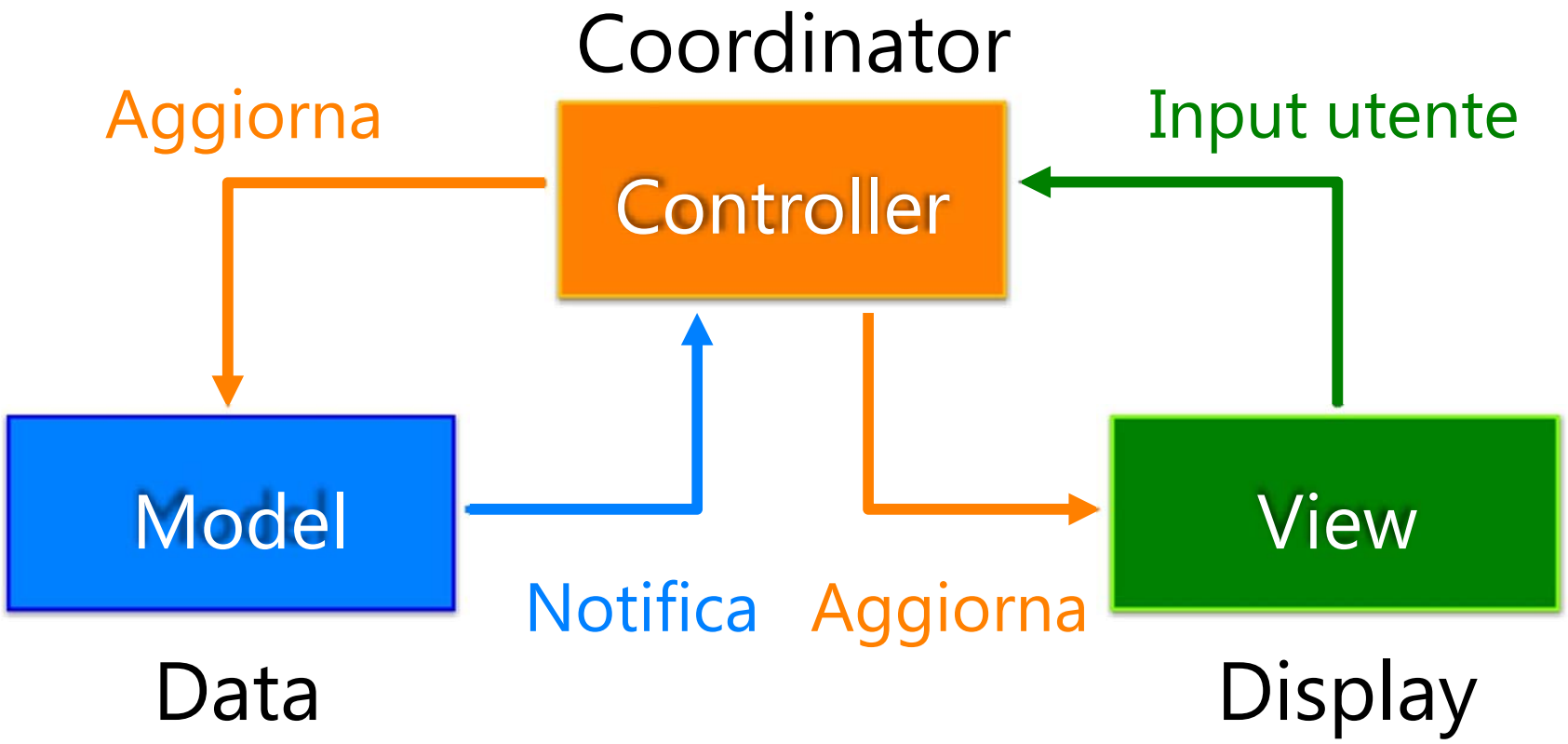
- Defining a Protocol

```
@protocol NSCoding

- (void)encodeWithCoder:(NSCoder *)aCoder;
- (id)initWithCoder:(NSCoder *)aDecoder;

@end
```

- Adopting a Protocol

```
@interface Person : NSObject<NSCoding> {
    NSString *_name;
}
// method & property declarations
@end
```

MVC



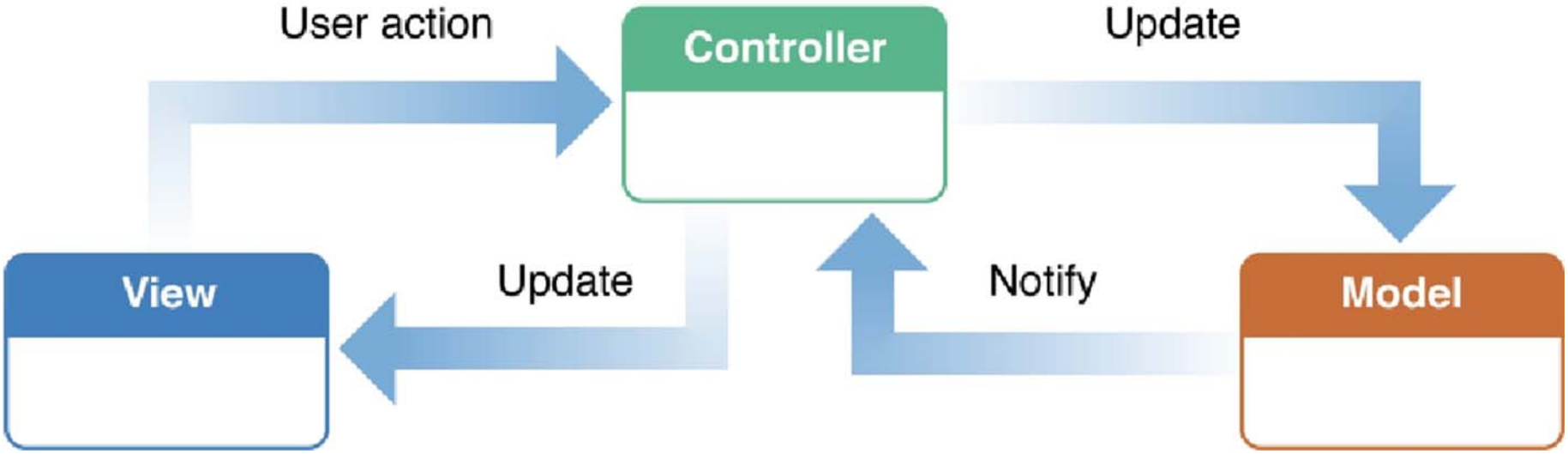
Il design pattern MVC assegna ad ogni oggetto un ruolo preciso:Model, View o Controller

Design Patterns

Design Patterns

- Model View Controller
- Protocol
- Delegation
- Notification Center

Model View Controller



- The Model-View-Controller design pattern (commonly known as MVC) assigns objects in an app one of three roles: model, view, or controller.

- The pattern defines not only the roles objects play in the app, it defines the way objects communicate with each othe

Model

A model object encapsulates the data of an app and defines the logic and computation that manipulate and process that data.

Example: a model object might represent a character in a game or a contact in an address book.

User actions in the view layer that create or modify data are **communicated through a controller object** and result in the creation or **updating of a model objec**t.

When a **model object changes** (for example, new data is received over a network connection), it **notifies a controller object**, **which updates the appropriate view objects**.

View

- A **view object** is an object in an app that **users can see**.
- A view object knows **how to draw itself and might respond to user actions.**
- A major purpose of view objects is to **display data from the app's model objects** and to enable the editing of that data.
- In an MVC app **view objects are typically decoupled from model objects.**
- **View objects learn about changes in model data through the app's controller** objects and communicate user-initiated changes—for example, text entered in a text field—through controller objects to an app's model objects.

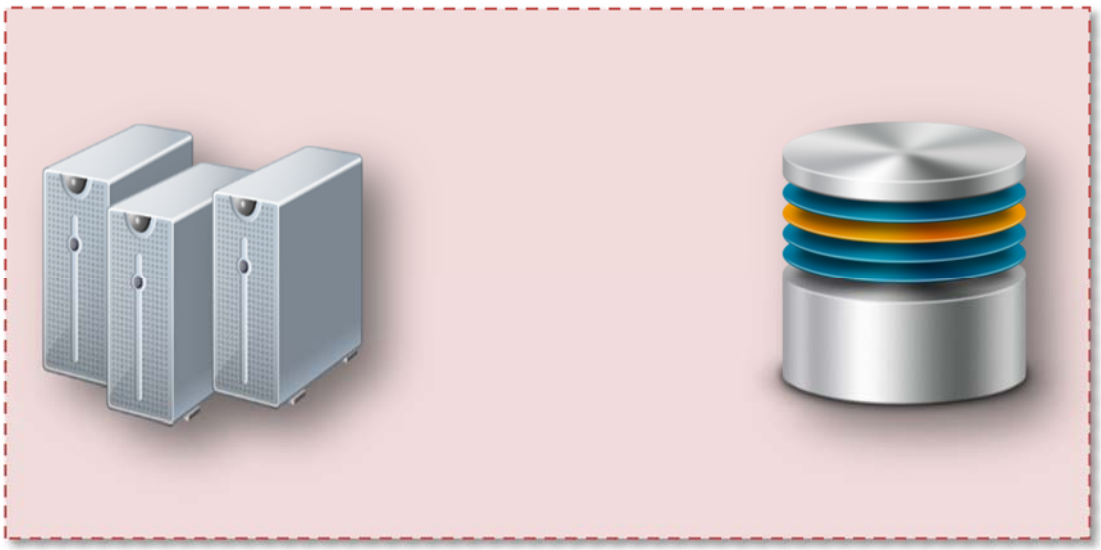
Controller

- A controller object acts as an **intermediary** between one or more of an app's **view objects** and one or more of its model objects.
- Controller objects can also **perform setup and coordinating tasks** for an app and manage the life cycles of other objects.
- **A controller object interprets user actions** made in view objects and **communicates new or changed data to the model layer.**
- When model objects change, a controller object communicates that new model data to the view objects so that they can display it.

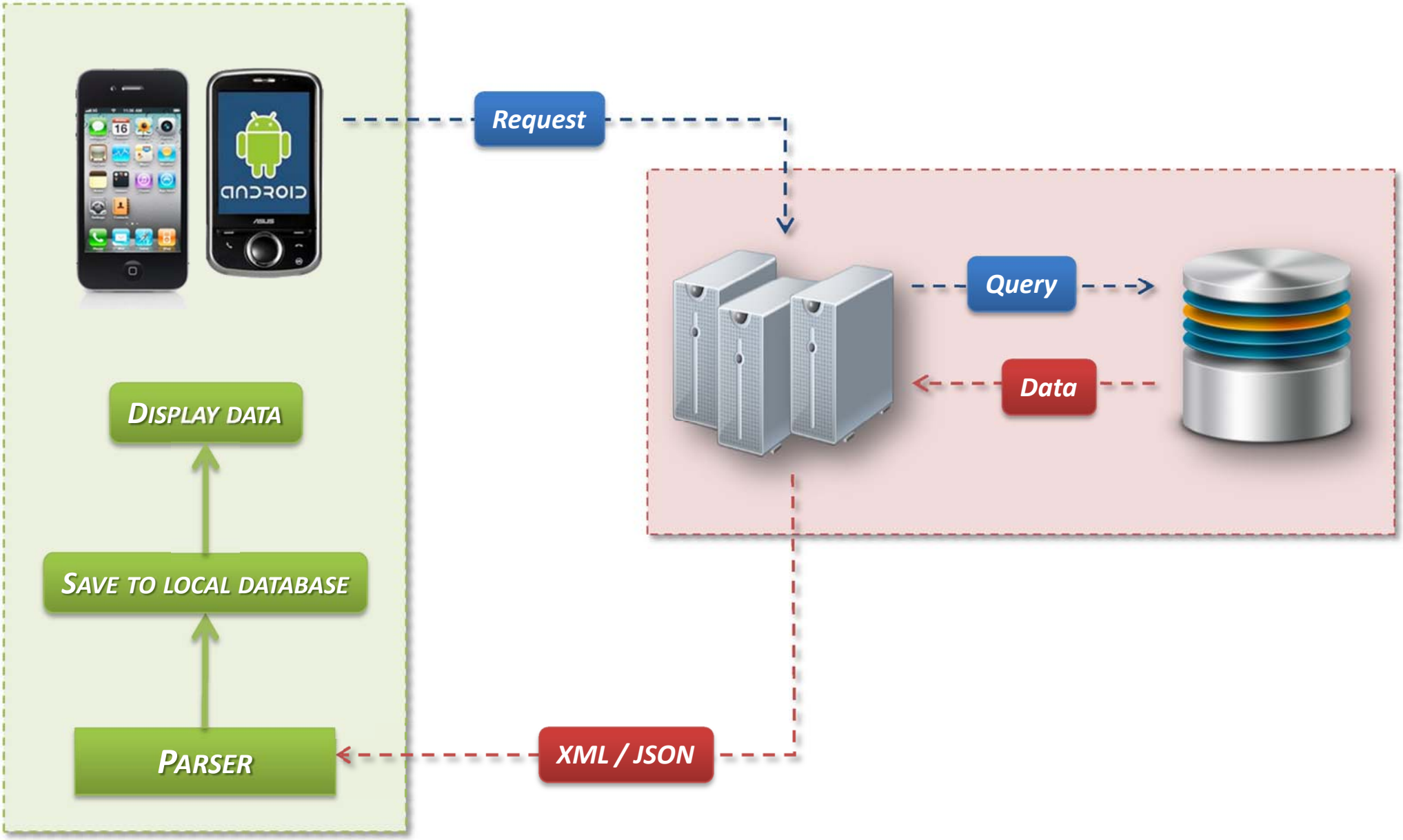
Controller

DISPLAY DATA

PROCESSING

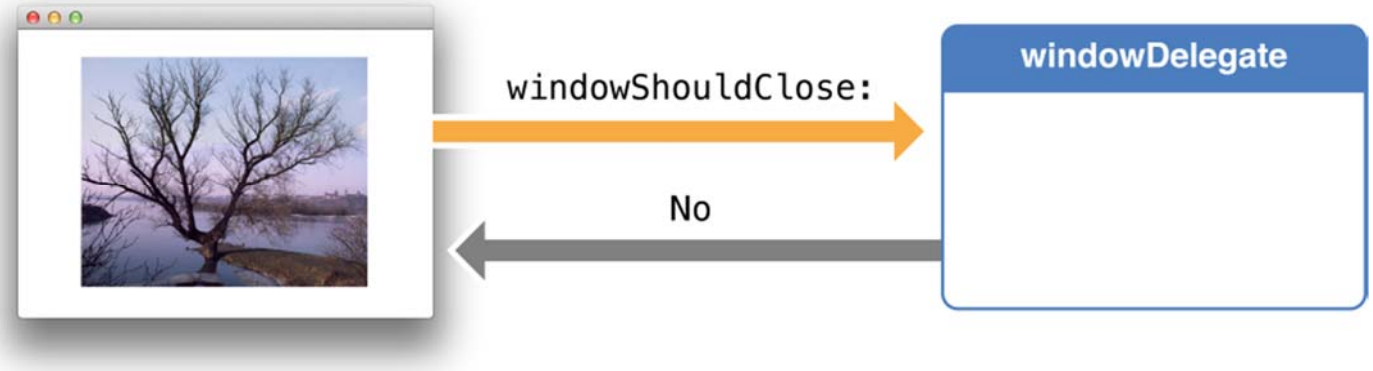


Controller


Request
Query
Data
DISPLAY DATA
SAVE TO LOCAL DATABASE
PARSER
XML / JSON

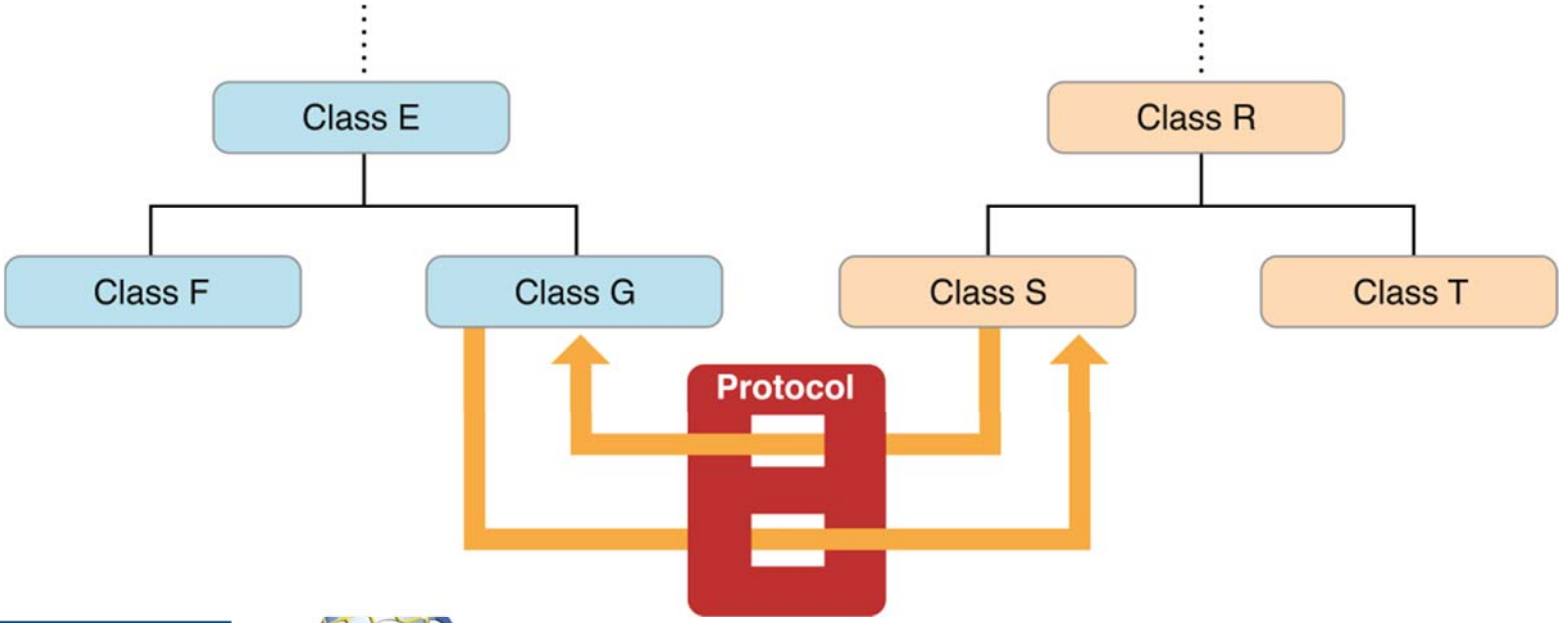
Delegation: Acting on Behalf of Another Object

- In delegation, an object called the ***delegate*** acts on behalf of, and at the request of, another object. That other, delegating, object is typically a framework object.
- At some point in execution, **it sends a message to its delegate**; the message tells the delegate that some event is about to happen and asks for some response.
- **The delegate implements the method invoked** by the message and returns an appropriate value.



Protocol

- A protocol is a declaration of a programmatic interface whose methods any class can implement. An instance of a class associated with the protocol calls the methods of the protocol and obtains values returned by the class formally *adopting* and implementing the protocol.
- A protocol is thus, as is delegation, an alternative to subclassing and is often part of a framework's implementation of delegation.



Example

```
1  UIAlertView *alert = [[UIAlertView alloc] initWithTitle:@"My Title"
2  message:@"My Message."
3  delegate:self
4  cancelButtonTitle:@"Cancel"
5  otherButtonTitles:@"Do Something",nil];
6
7  [alert show];
8  [alert release];
```


```
1  @interface ClassName : UIViewController
2  <UIAlertViewDelegate, UIActionSheetDelegate>
3  {
4  //Declarations
5  }
6
7  @end
```

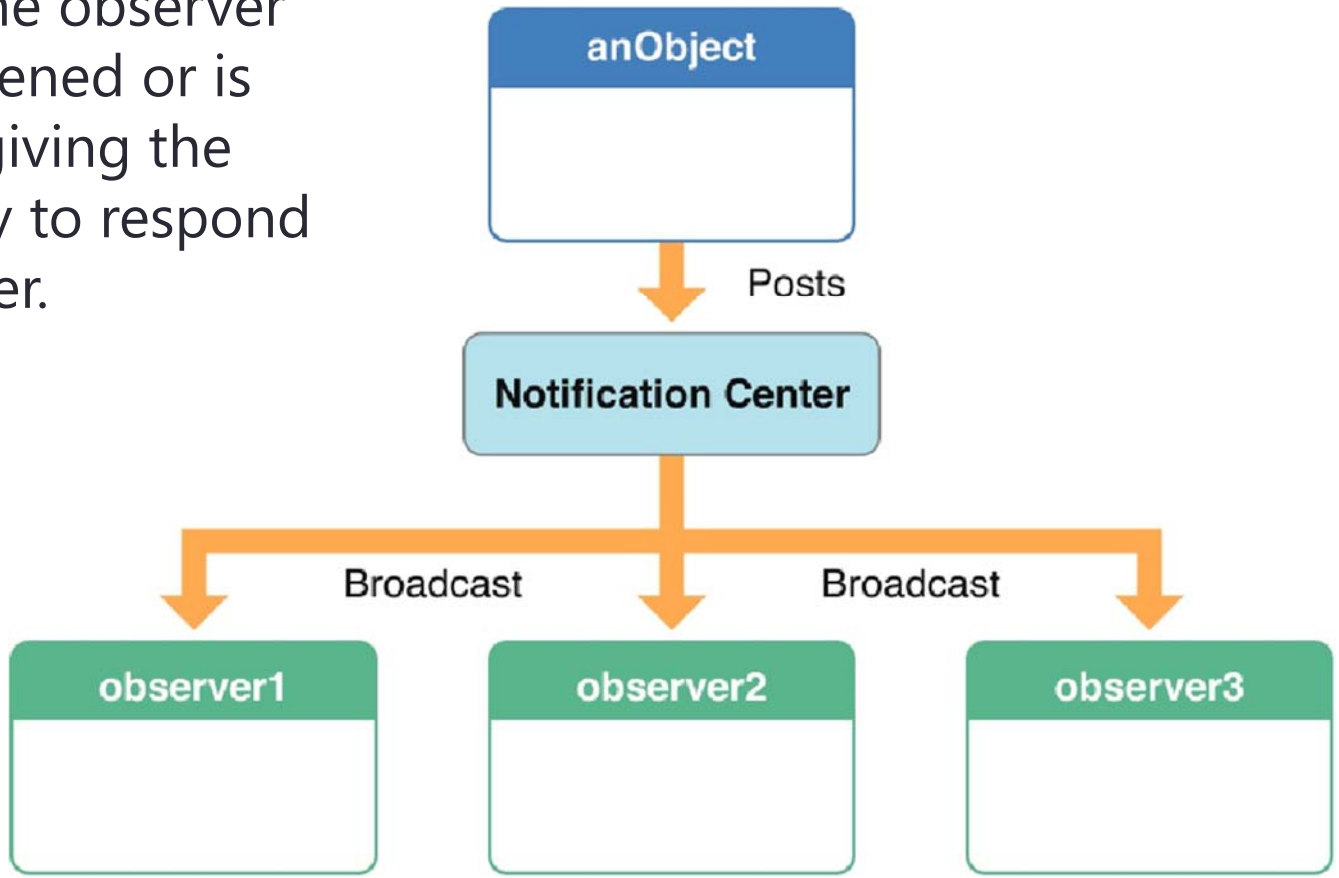

```
1  - (void)alertView:(UIAlertView *)alertView clickedButtonAtIndex:(NSInteger)buttonIndex {
2      if (buttonIndex == 0) {
3          NSLog(@"Button 0");
4      }
5      else if (buttonIndex == 1) {
6          NSLog(@"Button 1");
7      }
8      else {
9          NSLog(@"Unknown Button");
10     }
11 }
```

Notification Center

- A **notification center is a subsystem** of the Foundation framework that **broadcasts a message** — a **notification** — to all objects in an app that are registered observers of an event.

- A notification informs the observer that the event has happened or is about to happen, thus giving the observer an opportunity to respond in an appropriate manner.

- Notifications broadcast by the notification center are a way to increase cooperation and cohesion among the objects of an app.



Notification

- **Local notification**

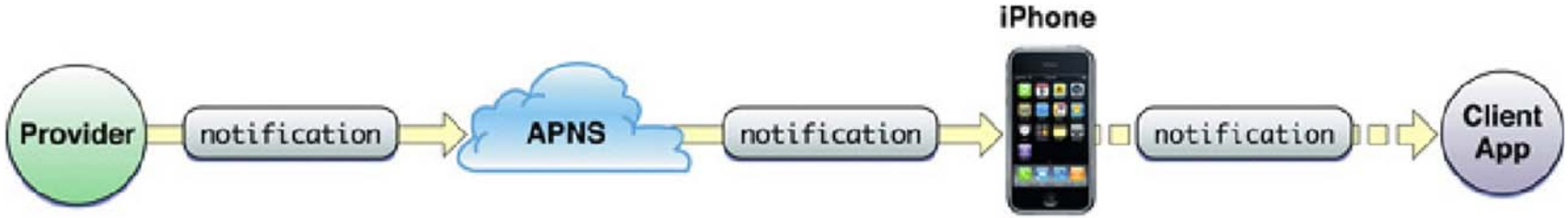
  Notifiche locali schedulate dall'app e gestite dal sistema operativo del singolo device.

- **Push Notification**

  Notifiche inviate dai serverApple (APNsApple Push Notification service) su tutti i device dove l'app è installata.

Push Notification

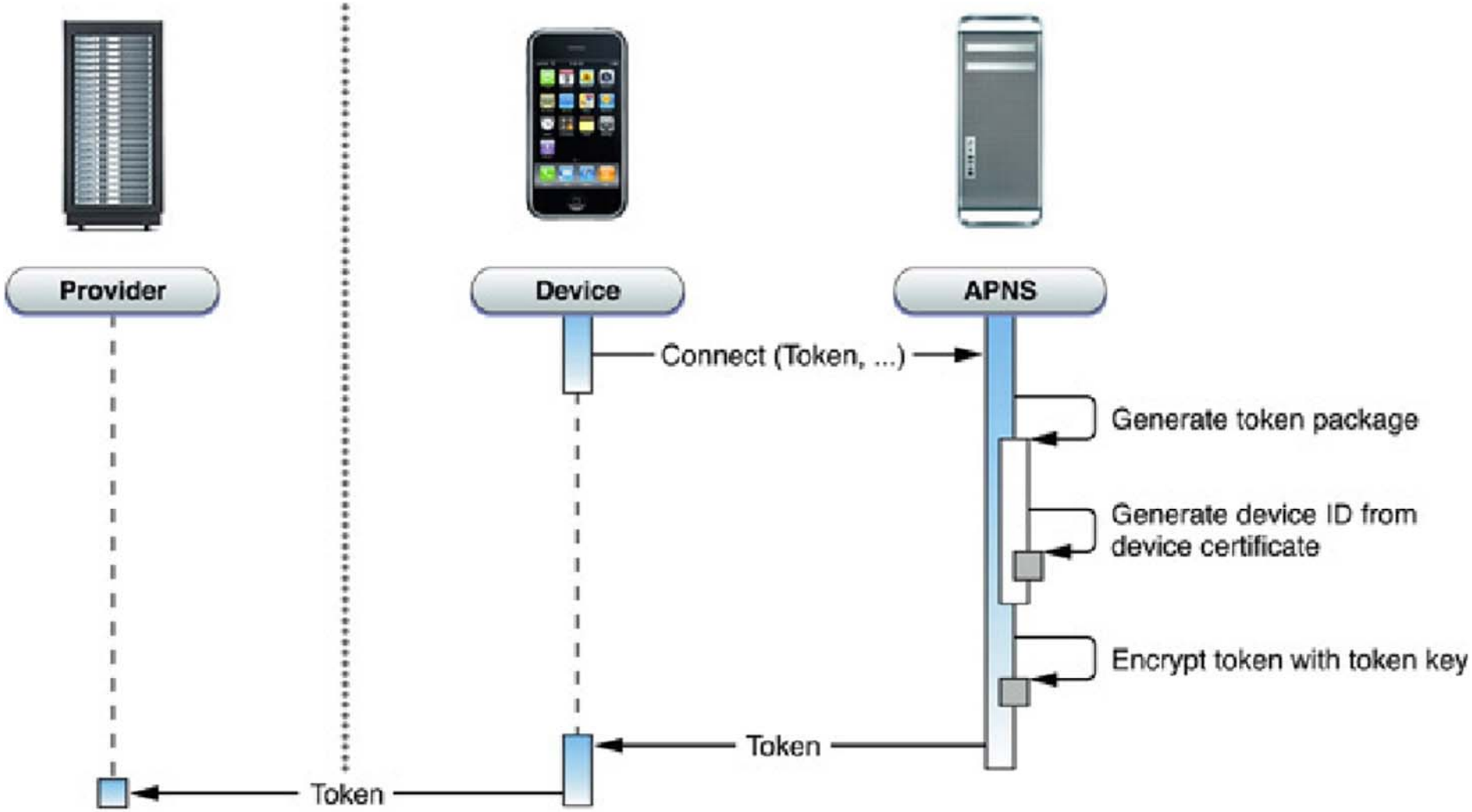
- **Apple Push Notification** service **transports and routes a notification** from a given provider to a given device.
- A notification is a short message consisting of two major pieces of data: the **device token** and the **payload**.
- The **device token** is analogous to a phone number; it contains information that enables APNs to locate the device on which the client application is installed. APNs also uses it to authenticate the routing of a notification.
- The **payload** is a JSON-defined property list that specifies how the user of an application on a device is to be alerted.



Token Generation and Dispersal

- Applications must **register** to receive push notifications
- The system receives the registration request from the application, connects with APNs, and forwards the request.
- **APNs generates a device token** using information contained in the unique device certificate. The **device token contains** an **identifier** of the device. It then encrypts the device token with a token key and returns it to the device.
- The device returns the device token to the requesting application as an NSData object. The application then must then deliver the device token to its provider in either binary or hexadecimal format.

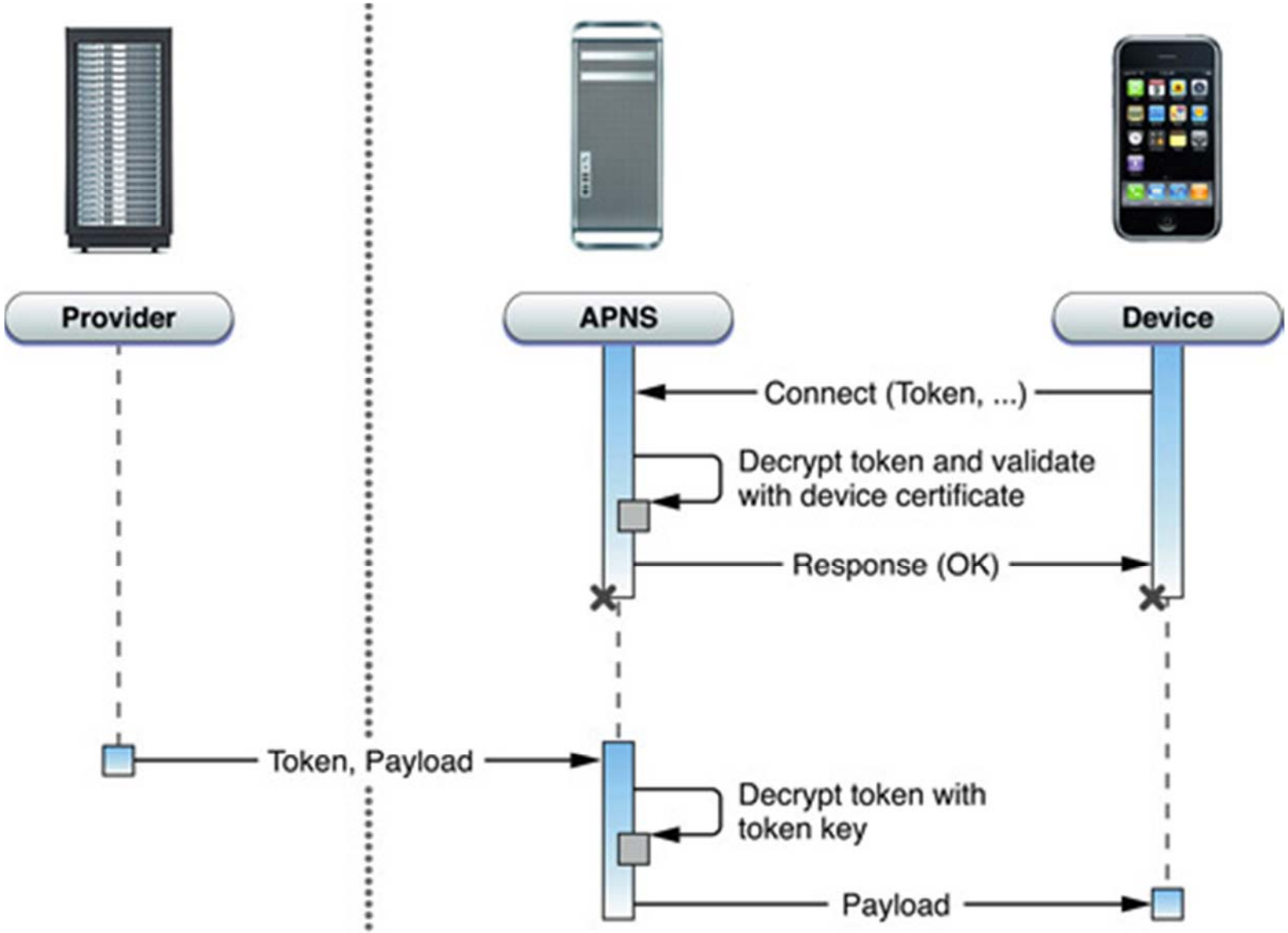
Token Generation and Dispersal



Token Trust (Notification)

- After the system obtains a device token from APNs it must provide APNs with the token every time it connects with it.
- APNs decrypts the device token and validates that the token was generated for the connecting device. To validate, APNs ensures that the device identifier contained in the token matches the device identifier in the device certificate.
- Every notification that a provider sends to APNs for delivery to a device must be accompanied by the device token it obtained from an application on that device.
- APNs decrypts the token using the token key, thereby ensuring that the notification is valid. It then uses the device ID contained in the device token to determine the destination device for the notification

Token Trust (Notification)



A Concrete Example

Mobile Emergency

Emergenze ospedaliere

Assenza di criteri standard per la creazione dei piani di emergenza → Incrementare la reattività e l'efficienza in caso di emergenza

Comunicazioni durante l'emergenza non gestite → Semplificare i meccanismi di comunicazione dell'informazione

Dispositivi mobili in ambito ospedaliero: accesso a cartelle cliniche, risultati di test e statistiche vitali dei pazienti tramite iPhone (M.Sinai Hospital, Toronto)

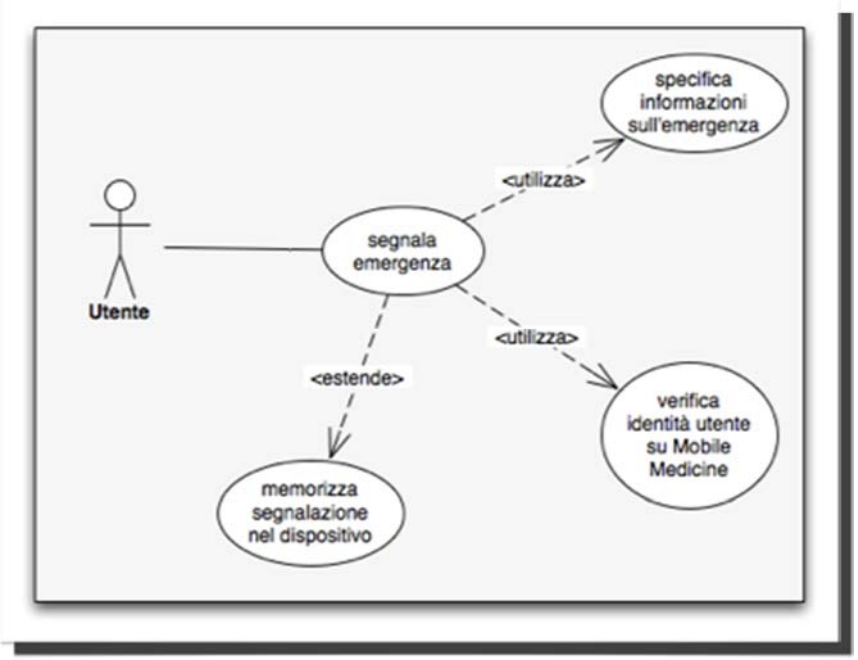
Mobile Emergency

ITALIA DEGLI INNOVATORI
Agenzia per la Diffusione delle Tecnologie per l'Innovazione
Presidenza del Consiglio dei Ministri

- **Segnalare** in modo dettagliato (luogo, stato pazienti, immagini, video) la presenza di una situazione di **emergenza**, gestire le segnalazioni effettuate
- **Consultare lo stato delle emergenze in corso**, con possibilità di ricevere notifiche dalla Centrale Operativa
- **individuare e segnalare la propria posizione indoor** tramite lettura di un codice **QR** posizionato all'interno dei locali della struttura
- individuare e segnalare la propria **posizione esterna** sfruttando il ricevitore GPS
- **Localizzare e raggiungere con un sistema di navigazione indoor/outdoor** le vie di uscita più vicine, le aree di raccolta definite dalla centrale operativa
- **Individuare gli operatori presenti all'interno dell'edificio**, conoscerne l'ultima posizione segnalata tramite la visualizzazione di una mappa, comunicare con essi tramite una sistema di messaggi in modalità broadcast o unicast
- Consultare una **checklist** sui compiti assegnati ai ruoli
- Richiedere informazioni sul Responsabile di un'emergenza o richiedere di diventarlo
- **Consultare mappe edificio e visualizzare i PDI presenti**

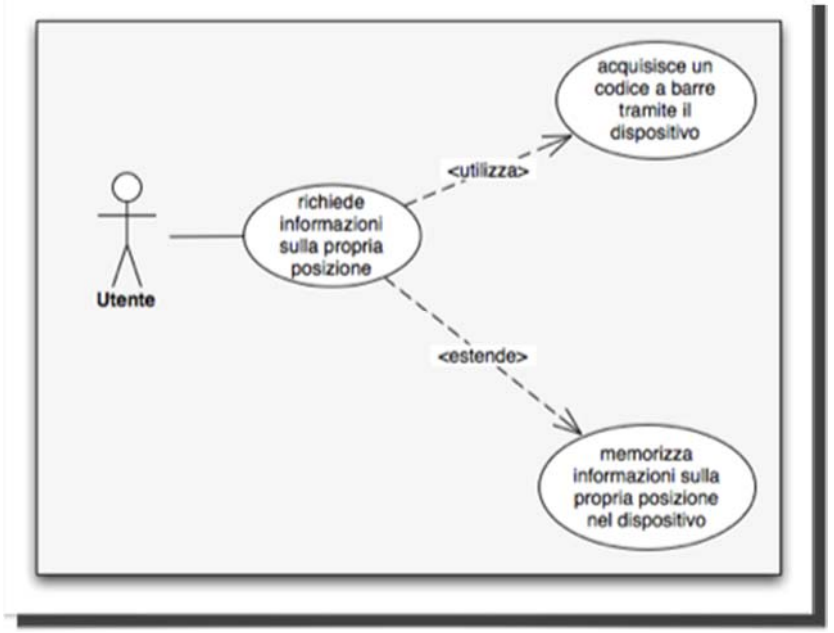
Available on the App Store

Analisi dei requisiti

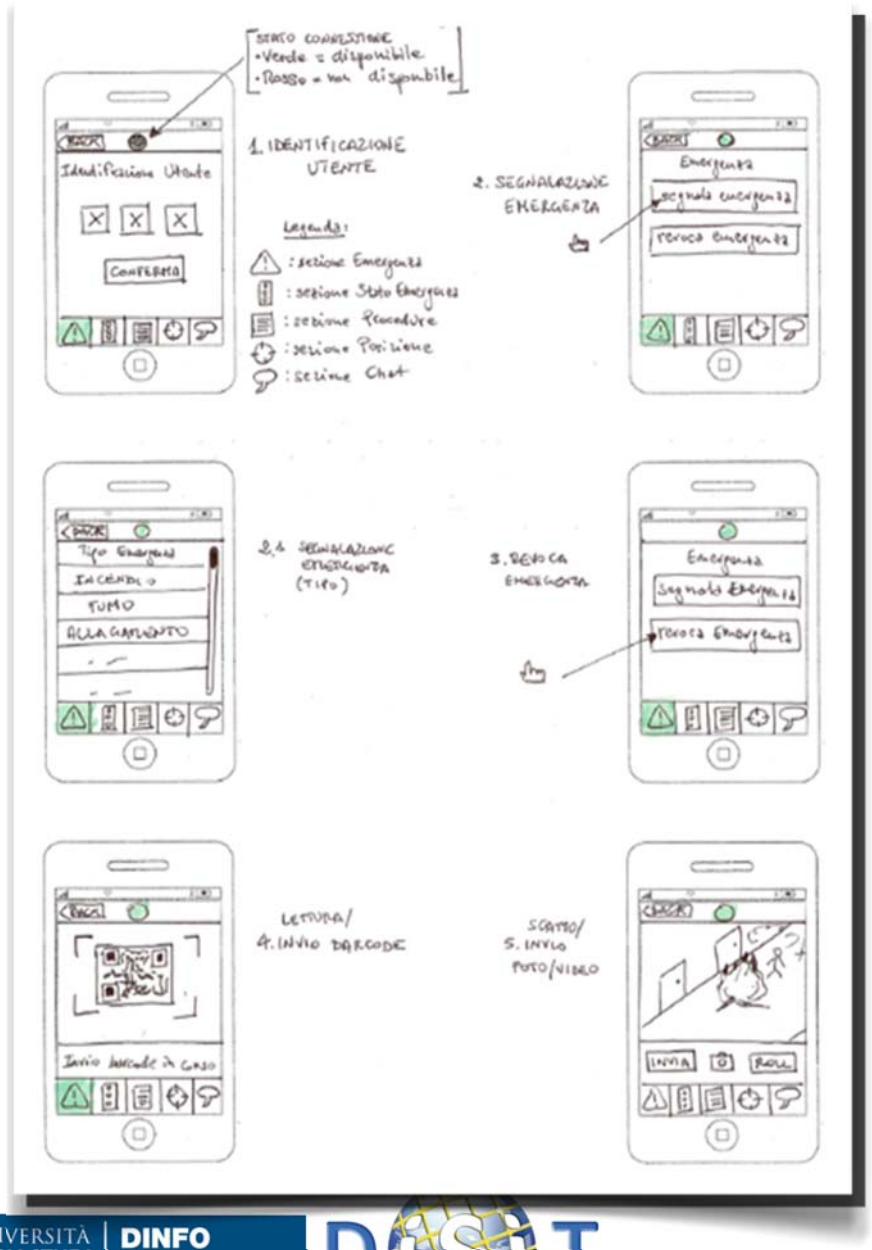


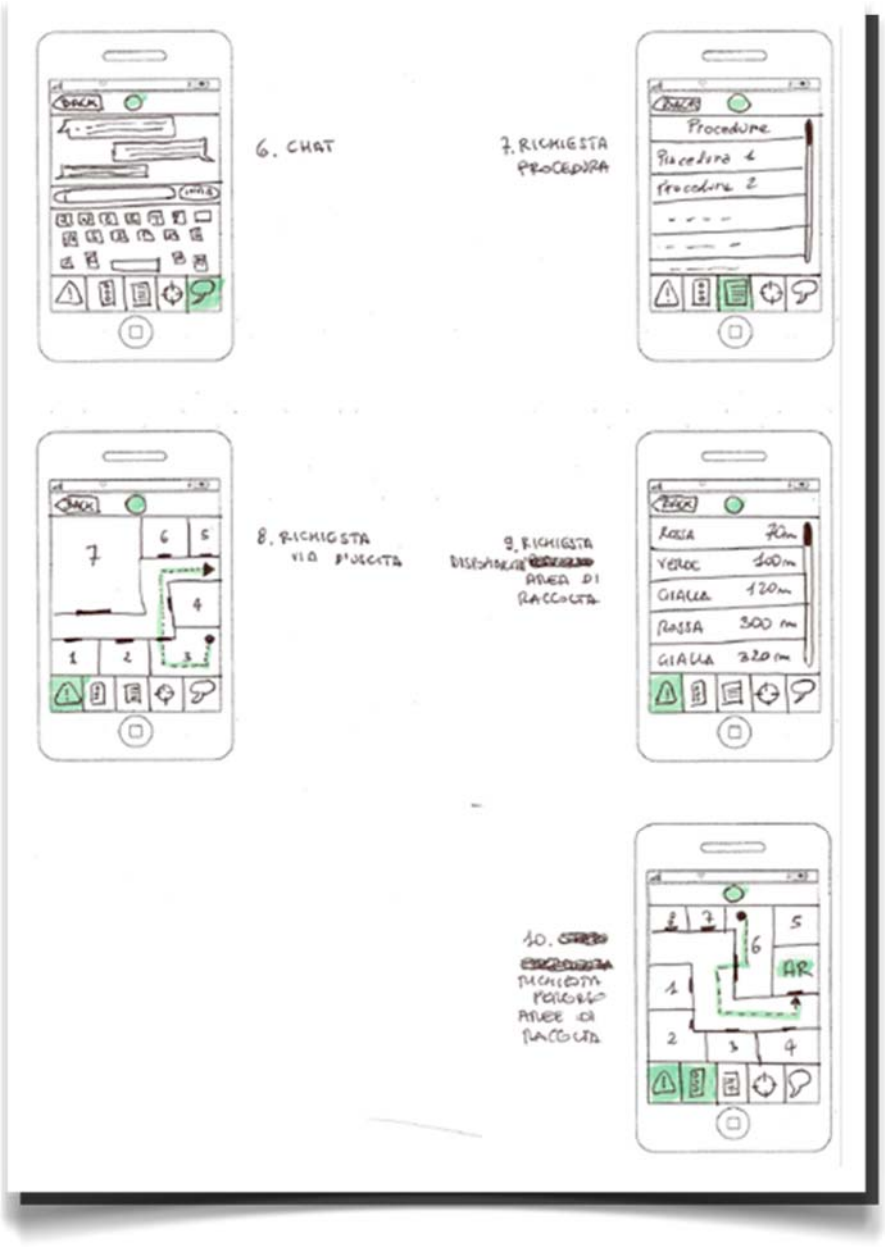
Segnalazione di una situazione di emergenza

Richiesta di informazioni sulla propria posizione

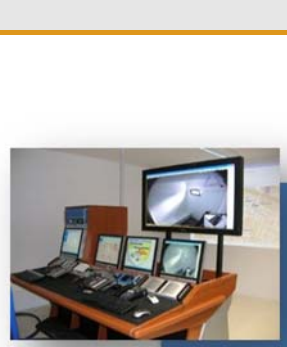


Progettazione della UI





Architettura del sistema



CENTRAL STATION

MOBILE EMERGENCY FRONTEND

Emergency Management

Maps Management

Messaging in Push

User Management

DATABASE

MOBILE MEDICINE

Best Practice Network and Services
http://mobmed.axmedis.org

MOBILE EMERGENCY PRO

User Interface

Emergency Manager

Media Acquisition and Delivering

Collaboration Discovering

QR acquisition

Map Consulting

Reach Emergency

PATHS MANAGER

Indoor Path Manager

Outdoor Path Manager

INDOOR NAVIGATION SYSTEM

COMMUNICATION MANAGEMENT

MOBILE DEVICES AND SENSORS

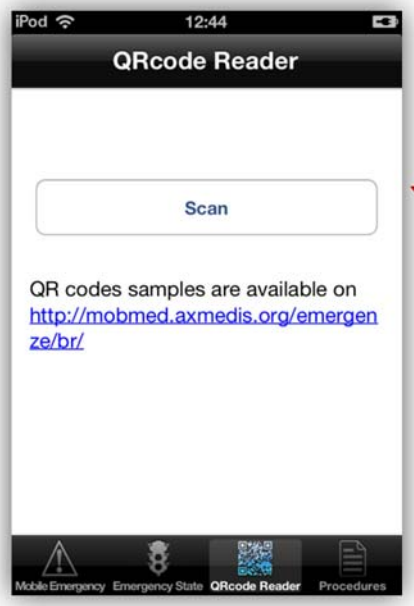
Mobile Medicine

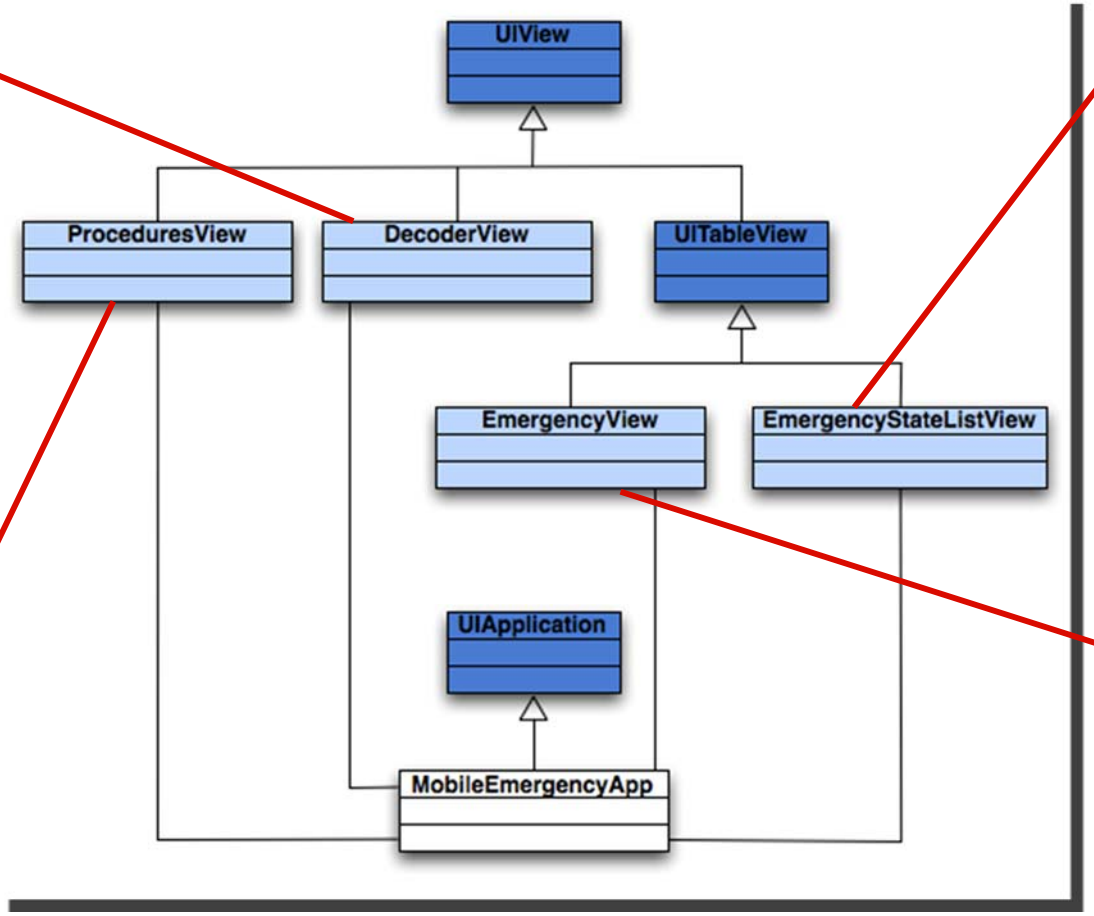
Content Indexer, Semantic Ingestion and processing

Information and history database

Content database

Diagramma delle classi





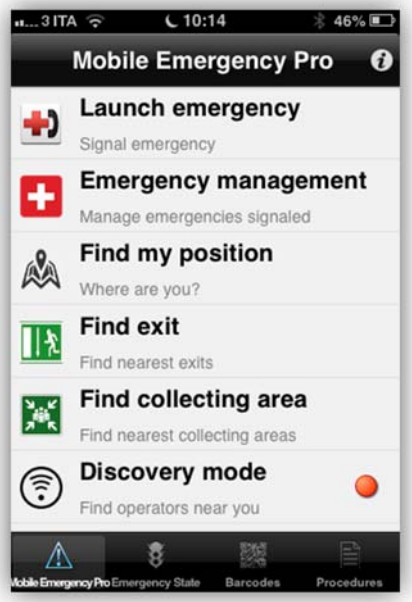
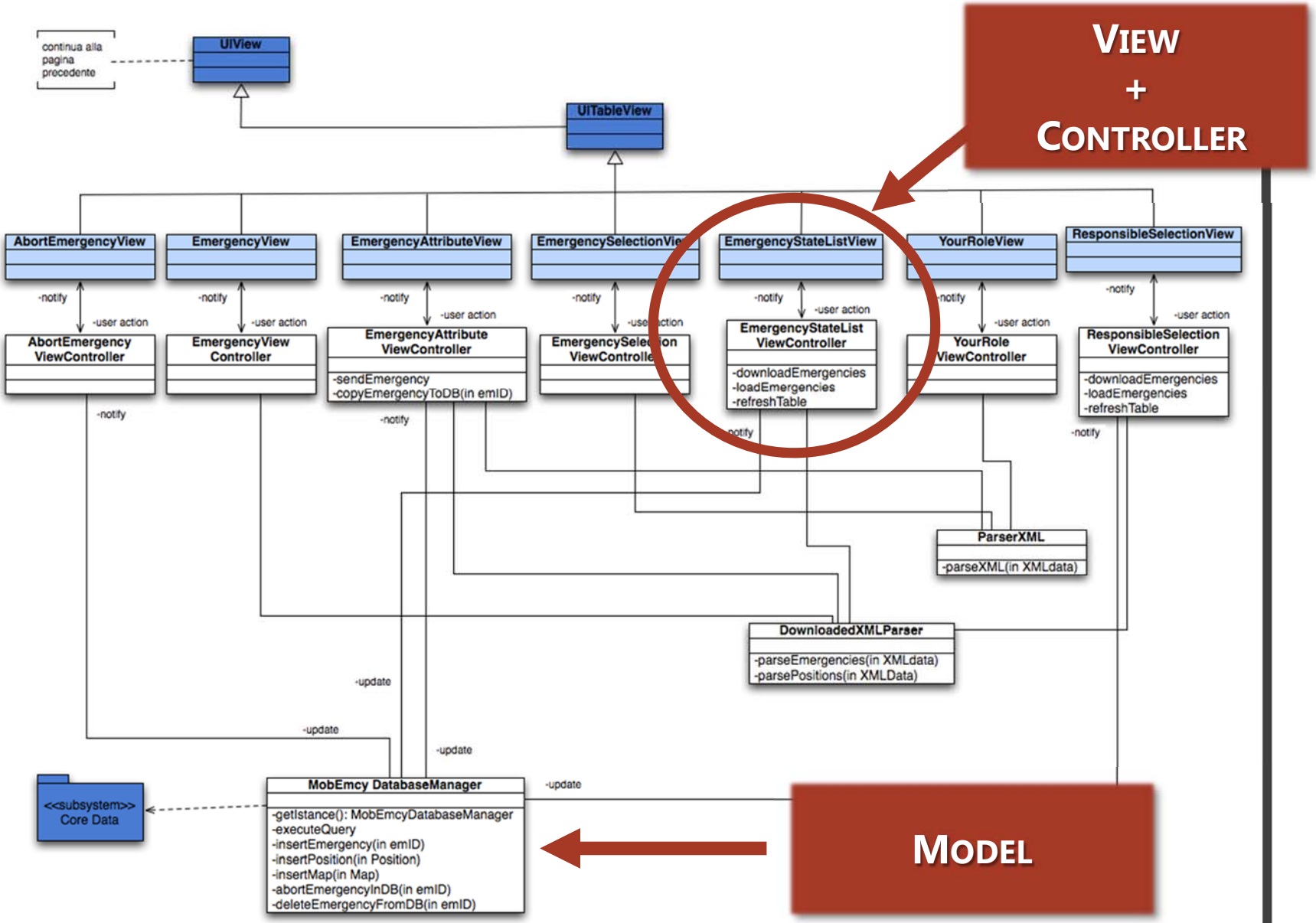
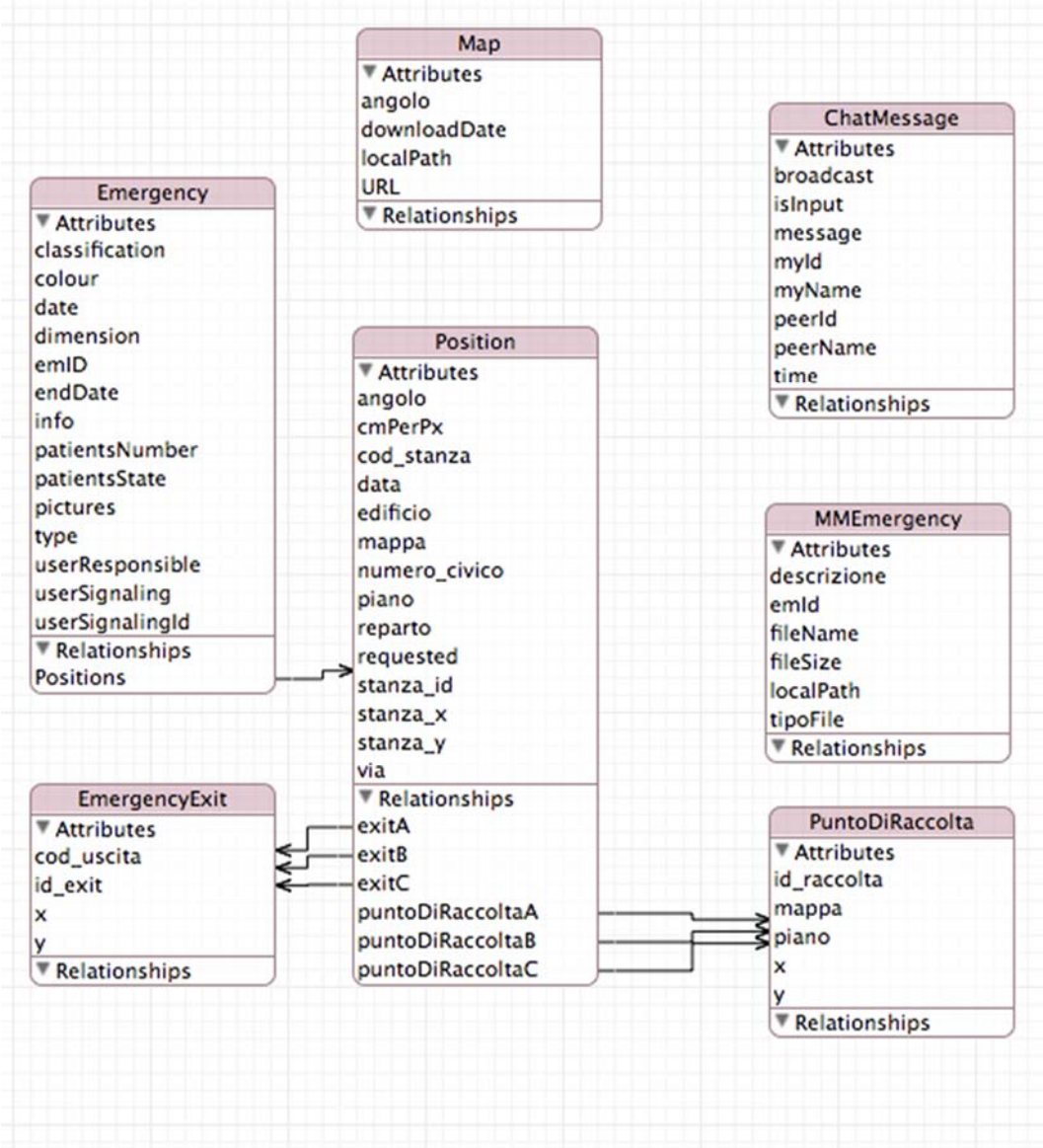


Diagramma delle classi



La base di dati



- **PERSISTENZA DEI DATI**
- **MODALITÀ OFFLINE**
- **RIDURRE IL TRAFFICO DI RETE**

Comunicazione Client / Server

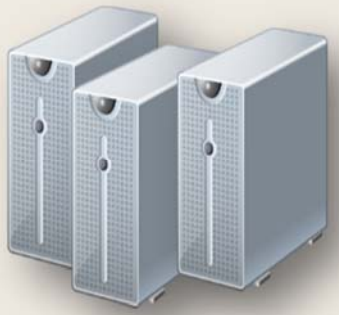
Comunicazioni tra dispositivi e Centrale Operativa tramite connessioni HTTP con richieste POST e GET (NSURLConnection)

Informazioni inviate dalla Centrale Operativa ai dispositivi sotto forma di contenuti XML/JSON

- Invio di **informazione formattata** per ridurre il traffico di rete
- Il dispositivo è responsabile della **visualizzazione** dei dati

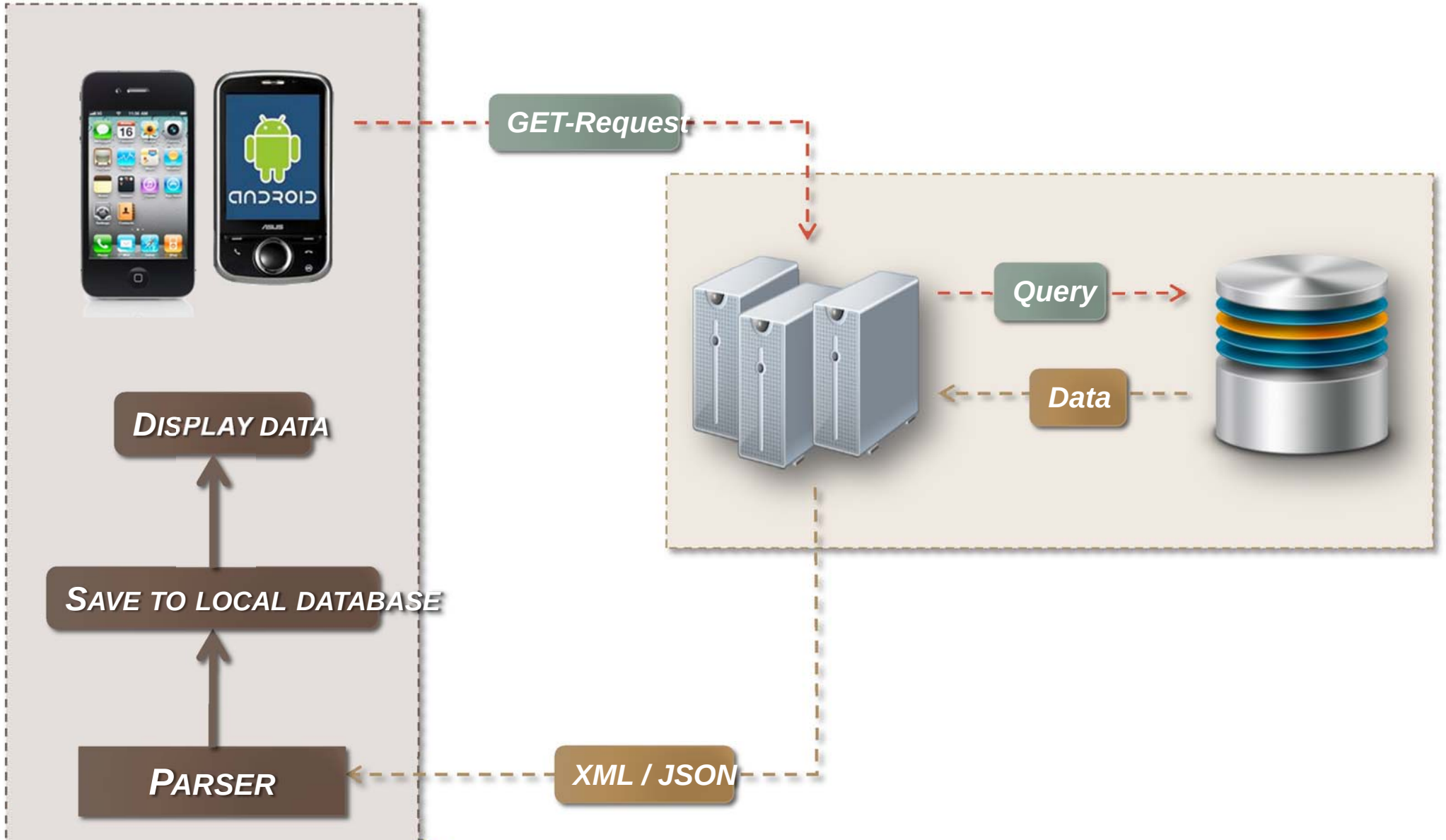
Comunicazione Client / Server

CLIENT



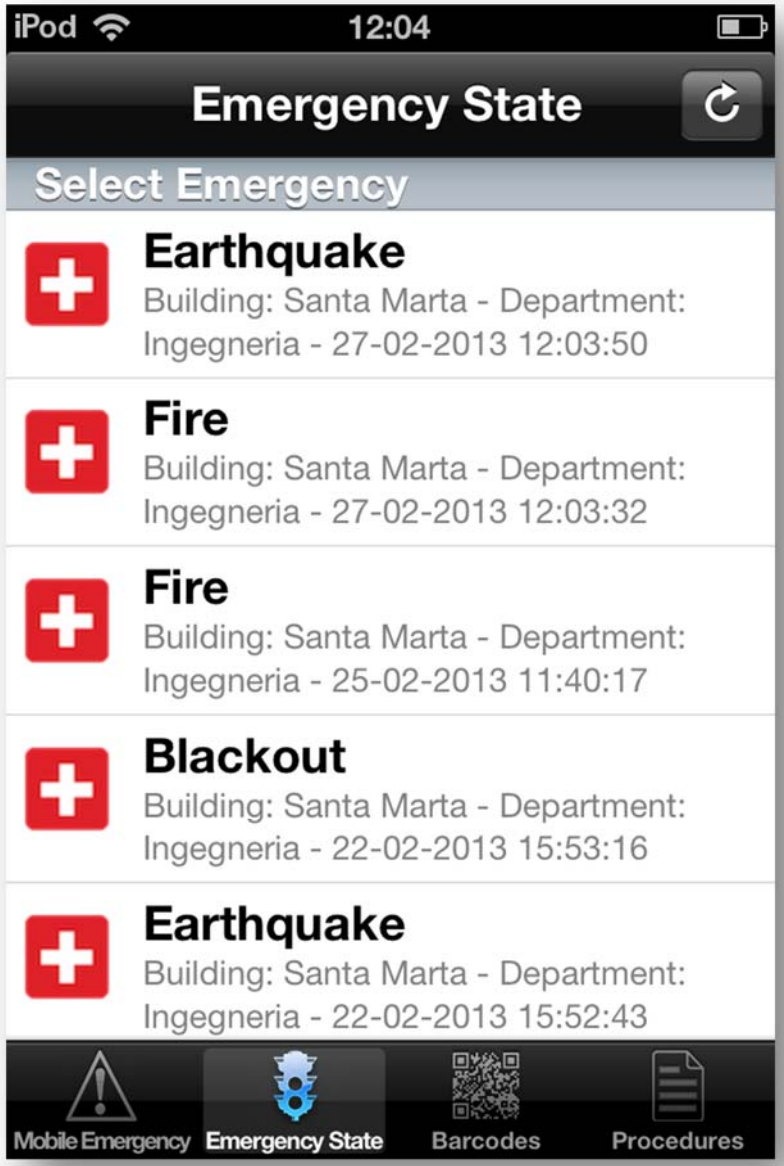
SERVER

Comunicazione Client / Server

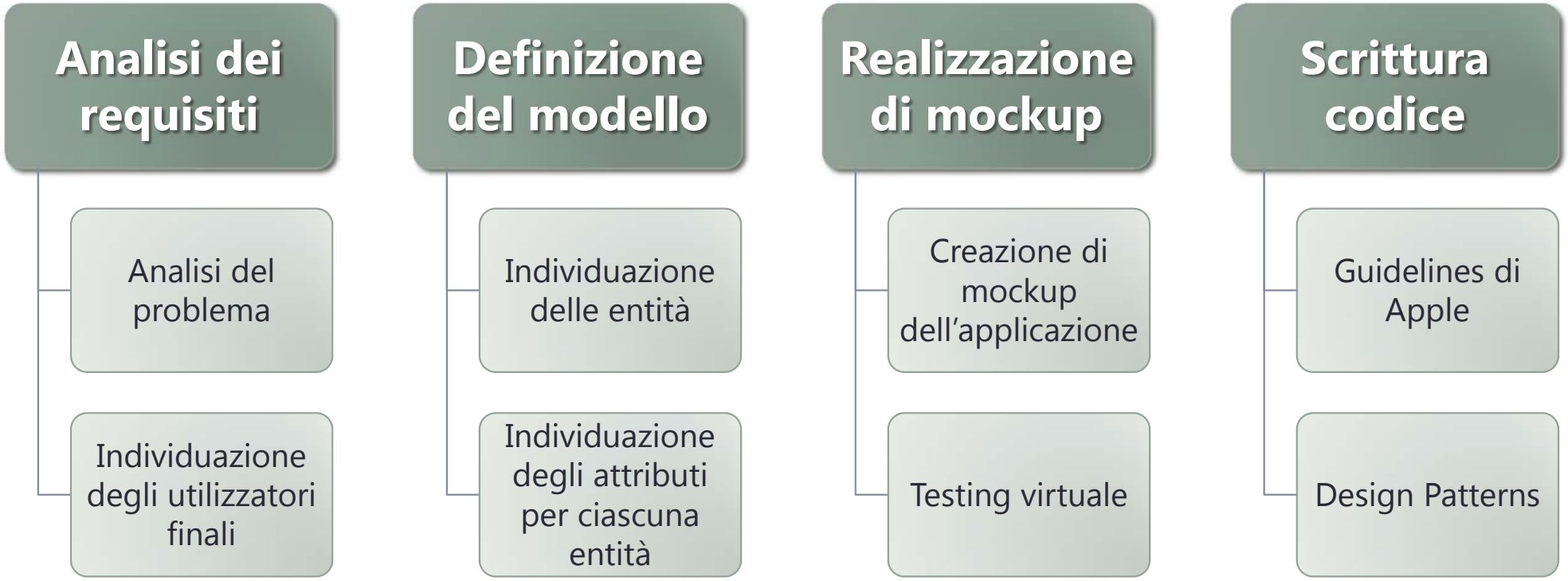


Comunicazione Client / Server

```
<?xml version="1.0" encoding="UTF-8"?>
<emergencies>
 <emergency>
  <id>15</id>
 <date>2013-05-10 09:50:55</date>
 <type>Fire</type>
 <dimension>Estesa</dimension>
 <patientsNumber>Nessuno</patientsNumber>
 <description></description>
 <patientsState></patientsState>
 <street>via di Santa Marta</street>
 <streetNumber>28</streetNumber>
 <building>Santa Marta</building>
<roomId>1</roomId>
 <x>1199</x>
 <y>699</y>
 <edificio_id>1</edificio_id>
 <gpsLat>43.798503619421275</gpsLat>
 <gpsLng>11.25300407409668</gpsLng>
 <floor>2</floor>
 <department>Ingegneria</department>
  <room>466</room>
  <userSignalingId>42</userSignalingId>
  <resposibleId>42</resposibleId>
  </emergency>
<emergency>
(…….)
```



Conclusioni



Credit

- Le slide relative alla programmazione sono state derivate dalle slide pubblicate su SlideShare da: Giuseppe Arici
  - giuseppe.arici@gmail.com
  - thepreacher@pragmamark.org
- Le slide relative Mobile Emergency sono derivate da un lavoro svolto al DISIT:
  - P. Bellini, S. Boncinelli, F. Grossi, M. Mangini, P. Nesi, L. Sequi, "Mobile Emergency: supporting emergency in hospital with mobile devices", Theme Issue Media Tablets & Apps (Guest editors: Pinciroli & Pagliari), JMIR RESEARCH PROTOCOLS, http://dx.doi.org/10.2196/resprot.2293, 2013.