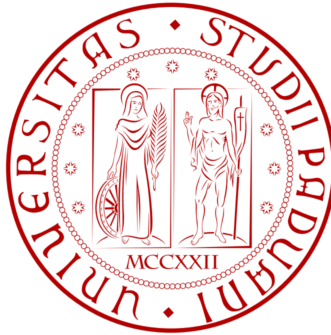


***Università degli Studi di Padova***

**Facoltà di Ingegneria**

# 2lvlROS

## a Second level Ruby Operating System

*Elaborato di progetto*

*Laureando: D. Brazzolotto*

*Matricola: 578761*

*Relatore: Prof. G. Clemente*

*Dipartimento di Ingegneria dell'Informazione*

*Anno Accademico 2009-2010*

18 luglio 2010

# Indice


**1 Digital literacy: stato dell'arte e prospettive future** **5**

1.1 Il problema . . . 5
1.2 Le proposte dei grandi produttori di software . . . 10
1.3 L'applicativo ideale . . . 13
1.4 Second Level Operating System . . . 14
1.5 Gadget . . . 17
1.6 Conclusioni . . . 21

**I Sviluppo di 2lvlROS** **23**

**2 Architettura** **25**

2.1 Architettura di base . . . 25
2.2 Organizzazione delle classi . . . 27

**3 Gadget** **29**

3.1 Nomenclatura . . . 29
3.2 Tipi di gadget . . . 29
3.3 Gadget: reti di Petri . . . 30
3.4 Interfaccia Grafica . . . 33
3.5 Gadget e funzionalità . . . 35
3.6 Gerarchia di Gadget . . . 41
3.7 Implementazione . . . 41

**4 2lvlOS: Kernel** **45**

4.1 Struttura del Kernel . . . 45
4.2 MicroKernel . . . 47
4.3 Moduli di sistema: I System Gadget . . . 56

**5 2lvlOS: framework** **63**

5.1 System.Configuration . . . 63
5.2 System.Data: Data Flow . . . 65
5.3 System.Security . . . 70
5.4 System.UI: Interfaccia . . . 70
5.5 System.IO . . . 71

5.6 System.IO.Net: Network e sicurezza . . . . . . . . . . . . . . . . . . . . . . . 74
5.7 System.Gadget . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 78

**6 Basi dati interne 81**
6.1 Basi di dati . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 81

**II Progettazione dei gadget 83**

**7 Farmacie 85**
7.1 Obiettivi . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 85
7.2 RdP . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 85
7.3 Sorgenti dati . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 86
7.4 Implementazione . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 86
7.5 ScreenShots . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 89

**8 Orologio 91**
8.1 Obiettivi . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 91
8.2 RdP . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 91
8.3 Sorgenti Dati . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 92
8.4 Implementazione . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 92
8.5 ScreenShots . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 95

**9 Adaptive Engine 97**
9.1 Obiettivi . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 97
9.2 RdP . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 98
9.3 Sorgenti Dati . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 98
9.4 Implementazione . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 98

**10 Conclusioni 101**

**III Appendici 103**

**A Processo di sviluppo 105**
A.1 Definizione processo di sviluppo . . . . . . . . . . . . . . . . . . . . . . . . . 105
A.2 Planning . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 106
A.3 Analisi dei rischi . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 108

**B Prototipi 111**
B.1 Ruby: Accesso dinamico ai WS . . . . . . . . . . . . . . . . . . . . . . . . . 111
B.2 Rails: Esporre WS . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 112
B.3 Shoes: convivenza widget . . . . . . . . . . . . . . . . . . . . . . . . . . . . 113
B.4 Ruby: eventi . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 113

B.5 Ruby: ereditarietà multipla . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 115

# Introduzione

Sono ancora in molti a credere che i computer siano l'apice della tecnologia, intesi come l'esperienza quotidiana ci suggerisce: scatoloni appoggiati a qualche scrivania, che consumano, scaldano, fanno rumore ed il cui quasi unico scopo sia complicare la vita ai propri utenti.

Il computer, viene insegnato ai corsi di informatica, è uno strumento general purpose che è possibile utilizzare per gli scopi più disparati, tuttavia l'idea di una macchina adatta a fare qualunque cosa è quanto mai superata, in favore di una miriade di dispositivi di ridotte dimensioni, a ridotti consumi, dalle specifiche funzioni, altamente connessi gli uni agli altri e di più facile utilizzo (almeno idealmente).

Si tratta di ipad, riproduttori musicali, smartfone, palmari ecc.. che anche se certamente godono di molte delle caratteristiche citate, richiedono ancora una certa abilità nel loro uso, un'abilità che continua ad essere un ostacolo per molti potenziali utenti e che va ad alimentare un problema di digital literacy già esistente da parecchio tempo (il livello di comfort percepito dalle persone nell'uso di software applicativo).

Per fare solo un esempio si pensi ad un utente che possiede determinate esigenze e che desidera utilizzare un applicativo per soddisfarle: egli non ha alcun supporto nella ricerca dell'applicazione a lui più corretta, nella eventuale fase di installazione e di apprendimento all'uso della stessa; deve avanzare per tentativi, a colpi di motore di ricerca, installando e rimuovendo diversi software (con tutte le relative conseguenze rispetto al corretto funzionamento del sistema operativo del dispositivo) prima di individuare il programma che risolve i suoi problemi. Molte delle applicazioni che tenterà di utilizzare non saranno nemmeno compatibili con il sistema utilizzato o richiederebbero determinate librerie per funzionare correttamente, ma ancora una volta l'utente è lasciato in balia degli eventi e la sua user experience sarà "non funziona, proviamone un altro".

Eppure con le tecnologie attuali, con la moderna potenza di calcolo esiste una concreta possibilità di rendere gli strumenti tecnologici realmente a misura d'uomo, di progettarli specificatamente per essere utilizzati da un essere umano sia esso un bambino od un novantenne, di dotarli di una user experience tanto impeccabile da dubitare di ogni software affiancato da un servizio d'assistenza poiché già prevederne l'uso dovrebbe essere sintomo di una mal progettazione (almeno per quanto riguarda gli aspetti legati al digital literacy).

Nell'ambito dei dispositivi di piccole dimensioni, come pda e smartfone, su questo punto si stanno facendo grossi passi avanti, mentre rimane una notevole carenza d'innovazione negli strumenti dotati di schermo più ampio, dove il classico PC è sicuramente l'ogget-

to più comune. È in questo contesto che il progetto 2lvlROS si inserisce, attraverso la progettazione di una piattaforma di sviluppo mediante la quale sia possibile creare applicazioni estremamente facili da utilizzare, fortemente connesse, capaci di recuperare, gestire e manipolare conoscenza idealmente alla pari di un essere umano, fornendo così una user experience completamente nuova, sia esteticamente che da punto di vista del behaviour, intuitivo e human-like.

Per supportare tutto questo non bisogna però dimenticare chi queste applicazioni idilliache deve crearle: anche gli sviluppatori hanno delle esigenze da analizzare e soddisfare al pari delle esigenze degli utenti, perché la disponibilità di applicazioni economiche ma di qualità è unicamente nelle loro mani.

Questa piattaforma è stata progettata come un sistema operativo di secondo livello, uno strato software da appoggiare sull'OS reale della macchina (per il quale è una semplice applicazione) che standardizzi l'accesso alle risorse disponibili esponendo interfacce ad elevatissima astrazione, così che le proprie applicazioni (chiamate gadget) possano usufruire di tutta una serie di servizi inimmaginabili per gli odierni software applicativi:

- installazione e rimozioni automatizzate
- accesso controllato a file system, rete, database privati e condivisi
- meccanismo per adattarsi nel tempo al comportamento dell'utente che le sta usando (in modo completamente trasparente ed automatico rispetto all'applicativo stesso)
- supporto multi lingua e multi cultura
- interfaccia grafica personalizzata per utente (e non più per applicativo)
- ecc..

Con questo approccio è possibile quindi installare il sistema operativo di secondo livello in macchine Windows, Linux o Mac, pur garantendo il corretto funzionamento dei gadget, che vivendo all'interno del 2lvlROS non necessitano di alcuna modifica o ricompilazione qualsiasi sia il sistema sottostante (strategico in questo senso è stata la scelta di Ruby come linguaggio di programmazione); l'utente invece sarà immerso in un mondo nuovo, costruito e calibrato a sua immagine e somiglianza: il sistema gli suggerirà i gadget che più si avvicinano alle sue esigenze, l'interfaccia grafica uniforme ma personalizzata gli permetterà di lavorare in un ambiente piacevole ma efficace, in cui tutti i comandi sono esattamente dove si aspetterebbe di trovarli (indipendentemente dalla specifica applicazione in uso), mentre il behaviour interno dei gadget in breve tempo si adatterà all'uso che ne verrà fatto, velocizzando le operazioni più frequenti.

Scopo di questo documento quindi è raccogliere e presentare la progettazione del 2lvlROS, prima attraverso un'analisi sistematica delle esigenze che utenti e sviluppatori manifestano nell'uso e nella creazione di un'ampia gamma di applicazioni di uso quotidiano, per proseguire attraverso l'esplorazione delle soluzioni proposte dai grandi produttori internazionali di software e solo successivamente verranno espresse le caratteristiche tecniche della piattaforma in oggetto attraverso i tipici strumenti UML. In appendice, dopo qualche gadget

di esempio e brevi conclusioni su quanto è stato fatto, sarà presente la documentazione del processo di sviluppo e una raccolta dei prototipi utilizzati.

# 1 Digital literacy: stato dell'arte e prospettive future

*Webster's Online Dictionary definisce il* Digital Literacy *come "le conoscenze e le abilità necessarie per usare la tecnologia in modo efficiente o, equivalentemente, il livello di comfort percepito dalle persone nell'uso di software applicativo". Si tratta di un concetto che ha acquisito una notevole importanza nella moderna società, che sempre più spesso non solo si rende conto delle potenzialità offerte dalla diffusione massiva di strumenti hi-tech per l'elaborazione delle informazioni, ma si rende altrettanto conto che molte delle limitazioni che si incontrano nell'applicabilità di tanta tecnologia sono imposte dalle elevate conoscenze che tali strumenti richiedono.*

*Per potersi liberare di questi vincoli è necessario semplificare l'accesso agli strumenti tecnologici, così da abbattere tutte quelle difficoltà percepite dagli utenti quando sono alle prese con un applicativo: se oggi è l'uomo che deve farsi* computer friedly *per interfacciarsi con una macchina, in futuro è necessario che sia lo strumento ad essere completamente ed incondizionatamente* human friedly.

*La chiave del successo sta quindi nel saper individuare una interfaccia uomo-macchina, ovvero un modello di applicativo, che sia comodo non solo agli utilizzatori finali, ma anche ai suoi sviluppatori che sebbene siano spesso dimenticati quando si parla di digital literacy, sono proprio coloro che con le loro azioni rendono realmente un applicativo più o meno accessibile alle masse.*

## 1.1 Il problema

Molte delle applicazioni che quotidianamente vengono sviluppate, si pongono l'obiettivo di semplificare l'accesso a quell'immensa mole di dati che oggi ci circonda ma che, per molteplici ragioni, non siamo in grado di sfruttare pienamente.

Questi applicativi, incentrati sui dati e dedicati ad un utente non esperto, pur essendo un'ampia parte della produzione mondiale di software non sempre raggiungono l'obiettivo che si erano prefissati e spesso finiscono per peggiorare la vita degli utenti invece di migliorarla. Nella situazione più comune ad esempio, è l'utente stesso (non esperto) che prima di iniziare ad utilizzare un applicativo deve:

1. Trovare l'applicazione giusta per le sue necessità
2. Installare l'applicazione sulla propria macchina

3. Imparare ad usare l'applicativo senza addestramento (le istruzioni non sono disponibili o l'utente non ha il tempo di leggerle)

Si tratta di tre operazioni concettualmente più vicine allo sviluppo che all'uso dell'applicazione, per nulla semplici o intuitive per l'utilizzatore che deve affrontarle completamente solo.

D'altra parte gli stessi sviluppatori si trovano spesso imbarazzati nella creazione di questi software in quanto:

- la connettività elevata
- la manipolazione dei dati anche complessa
- la necessità di rendere disponibile lo stesso applicativo in diversi sistemi operativi
- la necessità di fornire procedure di setup semplici in ogni condizione
- la necessità di proporre una user experience accattivante ma efficace

in assenza di adeguati framework, complicano ma soprattutto allungano notevolmente lo sviluppo, obbligandoli a spendere molto più tempo su questi tecnicismi che sulla vera logica dell'applicazione e rischiando, in questi tempi di vacche magre, di distribuire applicazioni grezze pur di rispettare i tempi di consegna.

È chiaro, a questo punto, che gli strumenti che oggi si hanno a disposizione per sviluppare software, in particolare questo tipo di applicazioni, non sono adeguati alle necessità di entrambe le parti in gioco: utenti e sviluppatori hanno necessità diverse ma complementari, che una più attenta analisi potrà aiutare a capire.

### 1.1.1 Il problema degli utenti

L'utilizzatore di queste applicazioni, come accennato, è un utente non esperto che desidera, mediante l'applicativo stesso, ottenere velocemente una particolare informazione.

Per fare questo è essenziale che l'utente:

1. Ottenga l'accesso all'applicativo
2. Capisca il funzionamento dell'applicativo
3. Effettui le operazioni desiderate

Si noti che dal suo punto di vista è significativo solo il punto 3, i precedenti sono solo un fastidio necessario. Perché l'applicativo possa effettivamente migliorare la vita ai propri utilizzatori è fondamentale che i punti 1 e 2 siano quanto più semplici e veloci possibile.

Allo stato attuale, nella maggior parte dei casi:

1. Per cercare l'applicativo che soddisfa la sua particolare esigenza, l'utente deve affidarsi ai motori di ricerca, avanzando per tentativi, in quanto non vi è alcuna relazione tra l'esigenza informativa e l'applicativo che la soddisfa (salvo rare eccezioni).
2. Ogni applicativo ha una diversa procedura di inizializzazione / installazione. Anche in questa fase l'utente è solo (salvo rare eccezioni).

3. Se il software non dovesse funzionare (perché non è disponibile una libreria nel sistema, o un plug-in del browser ad esempio) l'utente tipicamente lo scarta piuttosto di tentare di affrontare l'analisi e la risoluzione del problema.
4. L'interfaccia grafica è diversa per ogni applicazione (o quasi), creando disorientamento nell'utente che deve associare tipicamente le stesse operazioni a elementi grafici sempre diversi.
5. Le applicazioni non hanno un metodo di scambio dati standard, così se l'utente vede i risultati che cerca spesso non è in grado di estrarli dall'applicazione che li ha forniti per ulteriori elaborazioni[1].

### 1.1.2 Il problema degli sviluppatori

Gli sviluppatori hanno un unico grande problema: *soddisfare i bisogni (informativi e tecnici) degli utenti in tempi rapidi.*

Gli utenti però possono essere Windows, Mac o Linux user, possono avere o non avere installate sulla loro macchina questa o quella libreria, questo o quel browser ecc..

In questa situazione chi si accinge a sviluppare un applicativo, può scegliere essenzialmente tra due alternative: sviluppare una desktop applicati on o una web application.

#### 1.1.2.1 Desktop application

Con il termine desktop application si intendono i classici software applicativi, tipicamente compilati, che interagiscono direttamente con il sistema operativo della macchina che li ospita. Si caratterizzano per le elevate performance, per interagire con l'utente attraverso una finestra messa a disposizione dal sistema operativo stesso e per necessitare di procedure di installazione e rimozione.

Non vi sono limitazioni a quello che può essere fatto: una desktop application ha libero accesso a tutte le risorse della macchina attraverso le primitive del software d'ambiente, esponendo l'utente a un notevole problema di sicurezza. Il codice compilato rende praticamente impossibile un controllo runtime, ma d'altra parte anche nei pochi casi in cui si tratta di codice interpretato l'assenza di standard non permette comunque un'analisi dei sorgenti che vada oltre al "consenti / non consenti tutto".

Chi sviluppa desktop application, inoltre, nella quasi totalità dei casi si appoggia a un framework capace di fornire un accesso alle risorse (in particolare quelle grafiche) di più alto livello rispetto alle primitive esposte dai sistemi operativi, framework che talvolta è anche capace di rendere cross-platform il codice sorgente del software in questione, ma che non è esente da complicazioni:

[1] Questo è un aspetto che nel tempo si è perso: quando i software applicativi erano unicamente a linea di comando, passare i dati da un'applicazione all'altra era un'operazione molto più comune di oggi, che faceva dei file di testo, in particolare CSV, il suo vettore principale. Ai nostri giorni, dove tutto è grafico, gli standard di scambio dati si sono moltiplicati ma l'effettiva capacità degli applicativi di importare ed esportare informazioni (soprattutto in modo automatizzabile) è estremamente limitata.

| | **Desktop application** | **Web application** |
|---|---|---|
| *Ambiente di hosting* | Sistema operativo | Browser |
| *Codice* | Compilato / Interpretato | Interpretato |
| *Sicurezza* | Codice difficilmente controllabile | Codice più facilmente controllabile |
| *Prestaz in elaboraz* | Elevata | Ridotta, soprattutto in client-side |
| *Connettività* | Non vi è supporto, ma con qualche sforzo è possibile implementare qualsiasi protocollo di comunicazione | Orientata al client-server, altre architetture sono più difficili da implementare se non impossibili |
| *User experience* | In genere fluida. Grafica standard non accattivante, grafica personalizzata spesso scomoda e di cattivo gusto | Non vi sono standard per l'interfaccia grafica, tuttavia questa risulta spesso accattivante se statica ma ha grosse limitazioni negli effetti dinamici |

Tabella 1.1: Desktop e Web application a confronto

1. I framework sono molto legati ai sistemi operativi, sono pochi quelli che permettono di passare in un ambiente diverso mediante una semplice ricompilazione, nella maggior parte dei casi sono necessari adattamenti nel codice o di particolari librerie pre-installate nella macchina target (l'alternativa, in assenza di framework, è riscrivere l'applicazione per ogni sistema operativo).
2. Le librerie che sono necessarie sulla macchina che ospiterà l'applicativo possono pesare anche svariate decine di MB, allegarle al setup dello stesso può risultare troppo oneroso per la distribuzione, ma d'altra parte è anche spiacevole per l'utente doverselé procurare manualmente.
3. Anche con l'uso di un framework, la *user interface* che questi applicativi propongono può essere:
    a) quella nativa del SO: standard ma spesso insoddisfacente
    b) personalizzata: spesso non intuitiva e visivamente pesante
4. Solo qualche framework fornisce un accesso alle risorse e strumenti di manipolazione dati al reale livello di astrazione richiesto da chi sviluppa applicazioni, i più lavorano ancora ad un livello troppo basso.

#### 1.1.2.2 Web Application

Le Web Application sono profondamente diverse dalle Desktop Application prima di tutto per l'ambente di hosting: se le Desktop Application lavorano all'interno di una finestra

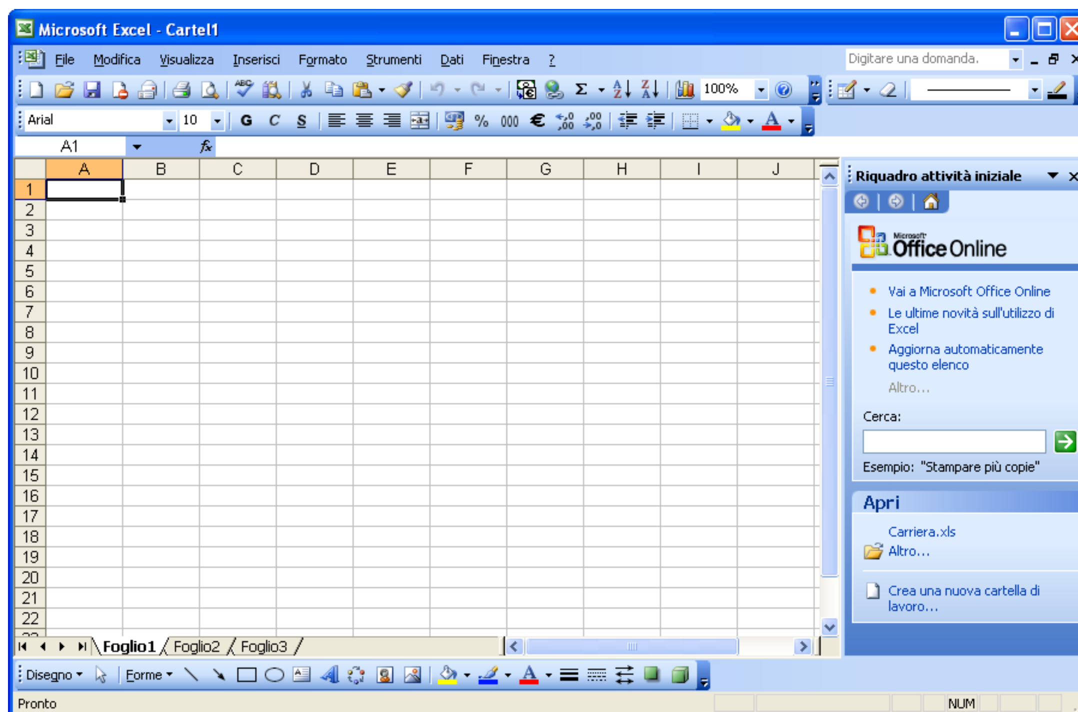


Figura 1.1: Tipica Desktop Application: Microsoft Excel

messa a disposizione dal SO, le Web Application vivono all'interno di un browser. Da questo punto di vista le risorse che una Web Application è in grado di usare sono molto limitate, sia in termini di varietà, per le limitazioni imposte dai browser, sia in termini di performance, a causa del codice interpretato.

La presenza di un browser però, rende molto più facile lo sviluppo di interfacce grafiche (statiche) accattivanti, anche se completamente esenti da standard, ed inoltre essendo un ambiente confinato è relativamente semplice implementare meccanismi di verifica del codice.

Un'altra caratteristica interessante è l'architettura client-server che una web application è naturalmente portata ad implementare, facilitando ad esempio la suddivisione tra *business logic* e interfaccia utente (cosa non semplice nelle Desktop Application). Allo stesso tempo però, questa stessa architettura limita quelle applicazioni che non si adattano ad un modello di questo tipo, rendendone quasi impossibile lo sviluppo (se non si utilizza qualche workaround).

Un grosso problema che si presenta in queste applicazioni, inoltre, è la varietà della tecnologia utilizzata:

- sia intesa come browser, in quanto ogni browser ha un comportamento leggermente diverso a cui l'applicativo deve adattarsi
- sia intesa come tecnologie di sviluppo, in quanto una moderna web application è costituita da un insieme di moduli integrati, sviluppati in linguaggi diversi, con logiche di funzionamento diverse, il cui sviluppo coordinato non è certo semplice.

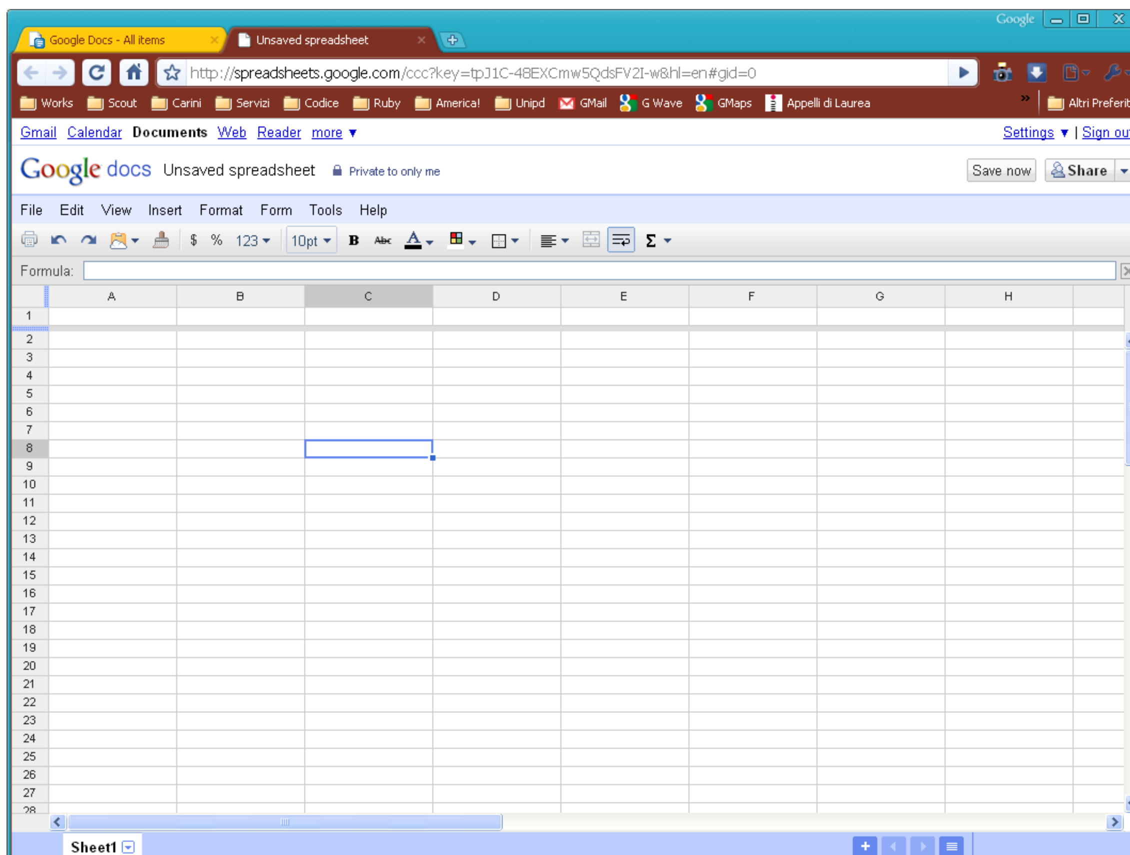


Figura 1.2: Tipica Web Application: il foglio elettronico di Google Docs

## 1.2 Le proposte dei grandi produttori di software

Che questi ultimi anni siano stati tempi di intensi cambiamenti è fuori discussione, almeno per quanto riguarda il software: i più grandi colossi mondiali sfornano tool e IDE di sviluppo a velocità incredibile, tanto da superare in frequenza i rilasci delle suite da ufficio, storicamente ricordate per le release annuali.

C'è tuttavia da chiedersi quanto questi continui cambiamenti siano il segno, oltre che di uno sforzo di ricerca, anche di effettiva e concreta innovazione.

### 1.2.1 IDE e tecnologie di sviluppo

Quando viene rilasciato un nuovo IDE o una nuova versione di un linguaggio, i primi a notare cambiamenti nel proprio modo di lavorare sono gli sviluppatori, che devono velocemente cogliere l'essenza della novità e adattare la propria metodologia di lavoro e i propri pattern, per poterla sfruttare al meglio. Queste novità si orientano verso:

**Maggior espressività dei linguaggi di programmazione** anche se non sempre si traduce in una riduzione significativa delle righe di codice impiegate, direttamente collegata al tempo di coding.

**Maggiore astrazione dei linguaggi di programmazione:** sempre più spesso il linguaggio usato dal programmatore non è lo stesso che viene compilato o interpretato durante la fase di compilazione o di esecuzione, è il caso di Rails in cui si scrive ruby e si ottiene javascript.

**Approccio più funzionale allo sviluppo:** lo sviluppatore specifica sempre meno come devono essere fatte le cose, affidandosi sempre più spesso al framework per la scelta migliore in ogni circostanza.

**Scrittura automatica del codice:** Intellisense e code-completation (ormai di serie in tutti gli IDE degni di nome) non bastano più; shippet, template e code generator permettono agli ambienti di sviluppo di scrivere in modo automatico anche interi moduli. Più difficile invece è addestrare gli IDE per modificare, in modo altrettanto automatico, questi moduli: sebbene comincino a diffondersi funzioni di refactoring, il più delle volte è il programmatore che deve operare le modifiche una ad una.

**Debugging e testing semplificato:** vi sono applicativi particolarmente ostici da testare, a causa dell'assenza di output facilmente controllabili o perché composti da diverse tecnologie integrate tra loro. Anche in quest'ambito è stato compiuto un notevole sforzo di semplificazione: gli IDE mettono a disposizione tool omogenei ed integrati capaci di tenere traccia dei diversi flussi esecutivi sotto diverse tecnologie sia che forniscano o meno output visibili all'utente.

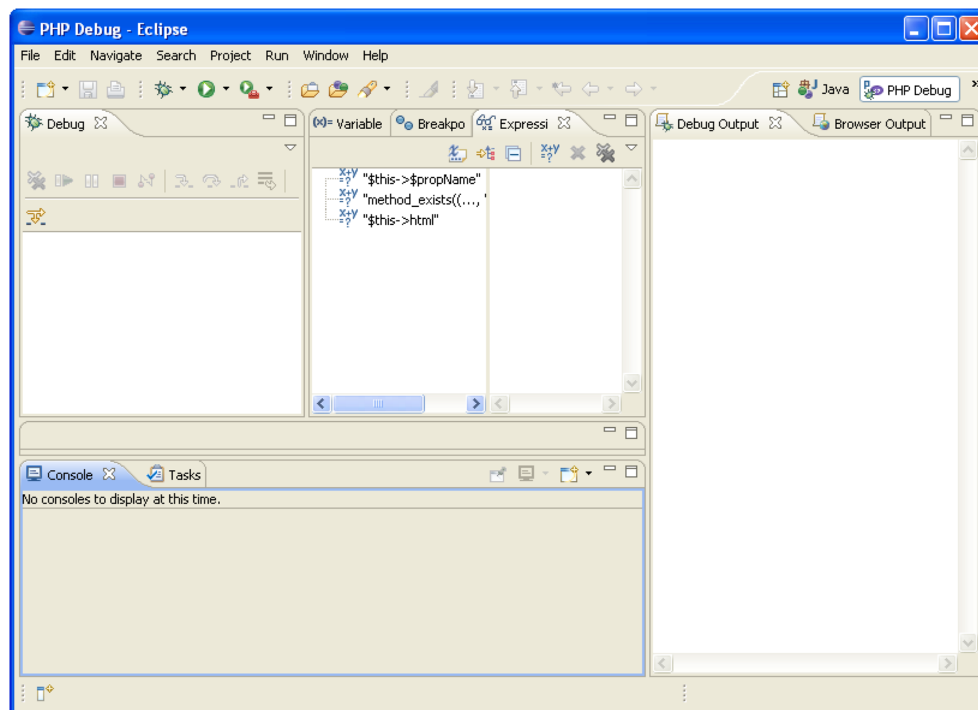


Figura 1.3: Uno dei più completi IDE disponibili in rete: Eclipse

### 1.2.2 Desktop e Web application

Per quanto le novità introdotte negli ambienti di sviluppo e nei linguaggi di programmazione migliorino la vita dei programmatori, nessuna di queste è in grado di aggirare i limiti delle Dektop o delle Web Application.

In questo senso la strada che si è cercato di percorrere è sostanzialmente quella di trovare un punto di incontro tra questi due approcci, capace di unire i vantaggi di entrambe le soluzioni:

1. Sviluppare Desktop Application con interfaccia html
2. Integrare le Web Application con moduli che più vicini al sistema operativo del client

Il primo caso è poco usato in quanto non elimina i punti deboli di uno sviluppo *desktop based* ma eredita, qualche beneficio e molti problemi dallo sviluppo *web based* (un esempio lo si può trovare nei moduli di gestione delle licenze di qualche grosso software).

Il secondo caso è già più comune: il grandissimo numero di applicazioni web che sono state sviluppate negli ultimi anni, appartenenti a domini applicativi sempre nuovi, hanno costituito una notevole spinta ad integrare nei browser (nati per leggere documenti ipertestuali) moduli eseguibili capaci di tutto (almeno in teoria) dal video streaming, alla grafica 3D.

I più comuni portano il nome di:

- Sun Java Applets
- Macromedia / Adobe Flash
- Adobe Flex
- Microsoft Silverlight

Si noti tuttavia che sono tutti workaround ai limiti dei browser, delle pezze proprietarie applicate senza una pianificazione strategica a medio / lungo termine che tenga conto della naturale evoluzione delle esigenze e degli stessi strumenti. Sintomi di questa poca attenzione sono l'assenza di standard (anche teorici), le pessime prestazioni e i numerosi problemi di comunicazione tra i contenuti all'interno e all'esterno di questi moduli eseguibili.

L'utilizzo di questi strumenti, inoltre, aumenta la varietà tecnologica cui poggiano gli applicativi, con tutti i problemi che questo comporta:

- *per gli sviluppatori* che devono pensare, sviluppare e testare su diversi approcci, diverse filosofie e talvolta anche su diversi strumenti
- *per gli utenti* in quanto più numerose sono le tecnologie con cui è sviluppato un applicativo, più sono le probabilità che una libreria manchi, rendendo inutilizzabile l'intero software. Questa probabilità di non-funzionamento aumenta ancora a causa della totale assenza di standard che può causare veri e propri comportamenti aleatori dell'applicativo in caso di esecuzione in condizioni non previste (per esempio una particolare versione del browser).

### 1.2.3 Proposte: successi o fallimenti?

Le proposte dei grandi del software sono necessarie, anzi indispensabili, per milioni di applicazioni in tutto il mondo, e non vanno certamente considerate come fallimenti: nel momento in cui sono state pensate e sviluppate, le esigenze erano tali da rendere inevitabile un salto tecnologico veloce e retrocompatibile. Va considerato tuttavia che questo non basta, oggi è necessaria una pianificazione di più ampio respiro, capace di condurre ad un nuovo concetto di applicazione, coerente e compatto, in grado di soddisfare in modo naturale le esigenze di tutte le parti in gioco, anche se può costare una netta rottura dal presente.

## 1.3 L'applicativo ideale

Per rendere più human friedly gli strumenti operativi degli utenti, ma al tempo stesso semplificare la vita a chi questi strumenti deve crearli, è necessario ripensare al concetto, al ruolo e alle caratteristiche degli applicativi, che dovranno conciliare sia le necessità degli utenti che degli sviluppatori.

### 1.3.1 Punto di vista dell'utente

Dal punto di vista degli utenti, l'applicativo ideale dovrebbe essere caratterizzato da:

1. Semplicità:
   a) L'utente non deve aver bisogno di libretti di istruzioni, tutto deve essere banalmente ovvio
   b) L'utente deve fare solo ed unicamente ciò che gli interessa, il sistema deve provvedere da solo a tutti i dettagli tecnici necessari
   c) Informazioni personali e modi d'uso automatizzati: il sistema deve autonomamente autenticare l'utente quando necessario e deve proporgli ciò che più probabilmente sta cercando.
2. Man-centric:
   a) L'utente deve essere messo nelle migliori condizioni di lavoro, l'interfaccia grafica deve essere personalizzata per utente, non per applicazione
   b) L'utente deve essere in grado di lavorare con tutte le applicazioni che gli servono sempre allo stesso modo: ricercare nuove applicazioni, installarle, utilizzarle sono operazioni che devono avvenire in modo trasparente dalla specifica applicazione.
   c) I dati devono essere completamente separati dalla loro rappresentazione e si devono assicurare le comunicazioni tra componenti dello stesso applicativo e tra applicativi diversi, così che l'utente sia in grado di modificare il flusso delle informazioni come ritiene più opportuno.

3. Multi user:

    a) Il sistema deve prevedere l'uso della stessa applicazione, nella stessa macchina ospite, da parte di più utenti, anche di cultura diversa (lingua, sistema di numerazione, sistema di valute, ecc)

### 1.3.2 Punto di vista dello sviluppatore

Dal punto di vista degli sviluppatori, invece, l'applicativo ideale si dovrebbe caratterizzare per:

1. Velocità:

    a) Il tempo dello sviluppatore è una risorsa preziosa, da questa dipende numero e qualità delle applicazioni disponibili. Avere una metodologia di sviluppo veloce ed espressiva permette di produrre applicazioni più complete, più efficienti e meno costose.

2. Struttura standardizzata:

    a) Il sistema deve prevedere un modello standard di sviluppo, a cui tutti gli applicativi devono attenersi, in modo da facilitare la manutenzione degli stessi.

    b) La stessa struttura standardizzata deve inoltre essere flessibile e scalabile, garantendo l'ordine nel codice anche nelle fasi evolutive più critiche dell'applicativo

    c) Il codice deve permettere un controllo automatizzato di sicurezza, capace di filtrare in modo accurato l'accesso alle informazioni e alle risorse sensibili.

3. Essenzialità:

    a) Ogni applicativo deve svolgere tutte e sole le sue funzioni.

    b) Lo sviluppatore deve definire gli applicativi con logica funzionale, specificando il cosa, non il come, al più alto livello di astrazione. Dovrà essere il sistema a prendersi carico dei dettagli tecnici.

4. Cross-platform:

    a) La possibilità di far girare un applicativo in diversi ambienti deve essere perseguita e massimizzata

    b) Deve essere chiaro in quali ambienti gli applicativi possono essere eseguiti e in quali no: comportamenti anomali ed errori dovuti a componenti mancanti non sono tollerati.

## 1.4 Second Level Operating System

La soluzione che si vuole proporre, interpone un nuovo livello di astrazione tra la macchina fisica e applicativo utente, chiamato sistema operativo di secondo livello (d'ora in poi

abbreviato **2lvlOS**) con lo scopo di fornire una piattaforma solida ma flessibile, capace di garantire ai suoi applicativi tutta una serie di servizi troppo di alto livello per il reale sistema operativo della macchina ospite, ma così comuni e trasversali che sarebbe assurdo integrarli (quindi svilupparli) in ogni applicativo. Tra questi servizi si troveranno un meccanismo capace di adattare automaticamente un applicativo ai propri utenti, interfacce verso le risorse del sistema maggiormente controllate, installazione e rimozioni automatizzate ecc.. in generale tutto ciò che rende felice l'utente finale ma che costituisce uno spreco di tempo per gli sviluppatori, più interessati alla logica funzionale che a questi "dettagli".

Questo modello nasce da una semplice considerazione: idealmente un programmatore di applicativi non è interessato a specificare tutti i dettagli di basso livello delle operazioni che programma (si pensi all'ordinamento di un grosso array) ma solo al risultato finale (l'array ordinato) che deve essere raggiunto con il minor consumo possibile di risorse. La scelta dell'algoritmo e la sua implementazione sono solo una seccatura per questo sviluppatore, che potrebbe limitarsi a dare delle indicazioni di alto livello (ordina l'array) e demandare al sistema operativo lo svolgimento dell'effettiva operazione (algoritmo usato).

I sistemi operativi odierti, tuttavia, sono troppo legati alla macchina fisica per permettere l'applicazione di questo principio a 360 gradi, le interfacce che espongono sono primitive di basso livello, flessibili ma ostiche da usare per chi è interessato solo al comportamento macroscopico e ai suoi risultati; per questo è necessario un ulteriore strato intermedio: il sistema operativo di secondo livello ha proprio lo scopo di tradurre le indicazioni di alto livello che i suoi applicativi (battezzati *gadget*) forniscono in efficiente implementazione fisica. Inoltre questo layer ha anche il compito di uniformare l'accesso al reale sistema operativo, rendendo i gadget quanto più possibile cross-platform (un altro punto dolente per i sistemi operativi tradizionali) senza incorrere nelle tipiche perdite di prestazioni delle macchine virtuali.

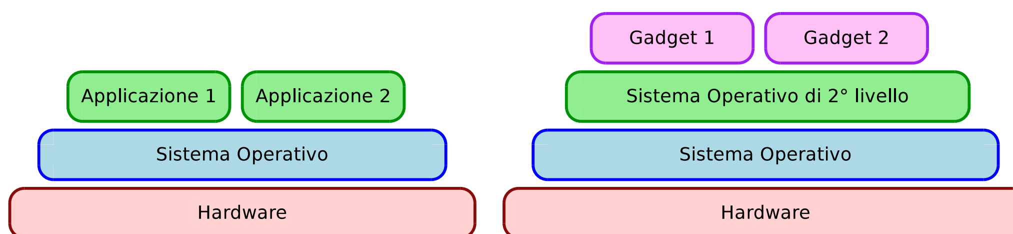


Figura 1.4: Architettura classica e con 2lvlOS a confronto

### 1.4.1 Ruby

Sebbene l'intero progetto possa essere sviluppato in qualsiasi linguaggio di programmazione (con qualche adattamento) si è scelto di utilizzare Ruby; dove sarà necessario prendere decisioni ad un livello più implementativo, pertanto, saranno le caratteristiche di questo linguaggio a guidarle.

"Ruby è un linguaggio di scripting completamente ad oggetti nato nel 1993 come progetto

personale del giapponese Yukihiro Matsumoto (spesso chiamato semplicemente Matz). Il linguaggio che ha maggiormente ispirato l'autore è lo Smalltalk, da cui Ruby ha tratto la maggior parte delle sue caratteristiche. A seguire ci sono il Lisp (ed in generale i linguaggi funzionali), da cui provengono le chiusure (blocchi o proc, in Ruby), e il Perl, per la sintassi e l'espressività.

Ruby, pur essendo principalmente un linguaggio ad oggetti, presenta alcune caratteristiche tipiche dei paradigmi imperativo e funzionale: il paradigma ad oggetti di Ruby è puro (tutto è un oggetto), come quello di Smalltalk, ma a differenza dei linguaggi come C++ e derivati, gli oggetti in Ruby sono qualcosa di molto più dinamico, in quanto è possibile aggiungere o modificare metodi a run-time. Il tipo di un oggetto, perciò, non è definito tanto dalla classe che lo ha istanziato, quanto dall'insieme dei metodi che possiede: è il cosiddetto *duck typing*, dall'inglese *if it looks like a duck, and quacks like a duck, it must be a duck.*

Nell'implementazione corrente, Ruby è un linguaggio interpretato. L'interprete, scritto in C, è rilasciato con una doppia licenza, GPL oppure "Licenza Ruby", e si trova attualmente alla versione 1.9.1. Negli ultimi anni la popolarità di Ruby ha subito una forte impennata, dovuta alla comparsa di framework di successo per lo sviluppo di applicazioni web, come Nitro e Ruby On Rails, nonché del Metasploit Framework, ambiente per la creazione e l'esecuzione facilitata di exploit." [Wikipedia]

Ciò che rende questo linguaggio particolarmente adatto al progetto è proprio la sua anima: i programmi Ruby sono semplici file di testo, facili da scrivere e da controllare (una volta standardizzati), ma l'interprete che li esegue non teme confronti in termini di prestazioni, proprio perché scritto in c. In questo modo è possibile sviluppare un'applicazione ad un elevatissimo livello di astrazione, con un linguaggio semplice, comodo ed espressivo, nella sicurezza di godere di prestazioni impeccabili in ogni situazione, tanto che qualora si manifestasse un collo di bottiglia sarebbe sempre possibile riscrivere quelle poche istruzioni in c e passare solo in quel particolare punto al codice compilato.

Si noti inoltre che l'interprete di Ruby è attualmente disponibile per numerosi sistemi operativi tra cui:

- La maggior parte di sistemi basati su Unix, incluso Linux
- DOS
- Microsoft Windows 95/98/2000/2003/NT/XP
- Mac OS X
- BeOS
- Amiga OS
- Acorn RISC OS
- OS/2
- Syllable

- Playstation Portable

il che rende le applicazioni ruby estremamente cross-platform.

**2lvlOS vs 2lvlROS**
Con 2lvlOS ci si riferisce genericamente a tutti i sistemi operativi di secondo livello, a tutti quei software cioè il cui scopo è fornire interfacce di più alto livello rispetto a quelle esposte dal reale OS della macchina ospite, per una determinata famiglia di applicativi; mentre con 2lvlROS si intende il "Ruby Second Level Operating System", quel particolare 2lvlOS qui oggetto di progettazione.

## 1.5 Gadget

I gadget sono le applicazioni del 2lvlOS ed in quanto tali sapranno godere di tutti i servizi messi a disposizione dal sistema operativo di secondo livello, saranno veloci e semplici da sviluppare ed ancor più semplici da utilizzare. In questo capitolo se ne darà una prima analisi evidenziandone caratteristiche e requisiti principali, mentre una più approfondita trattazione sarà fatta nel capitolo 3 dedicato alla loro progettazione.

Figura 1.5: Esempio di come potrebbe apparire il 2lvlROS ed alcuni suoi Gadget

**Definizione.** (di Gadget) Un gadget è un'applicazione del 2lvlOS che ha lo scopo di

esporre all'utente, o ad altri gadget del sistema, una specifica funzionalità (vedi immagine 1.5).

Il sistema nel suo complesso quindi, è composto da un numero arbitrario (anche grande) di gadget, ognuno caratterizzato da funzioni specifiche e complete. I gadget, in quanto applicazioni del 2lvlOS, risiederanno fisicamente nella stessa macchina che ospita il 2lvlOS stesso, il quale fornirà l'unica via di accesso, controllata e al giusto livello di astrazione, alle risorse del sistema.

### 1.5.1 Scenari d'uso

Un gadget viene utilizzato sostanzialmente da due attori:

- Il Provider, ovvero chi lo crea
- L'utente, chi lo utilizza

Le operazioni che concettualmente riguardano un gadget sono quelle in figura 1.6:

- Provider:
  - La pubblicazione del gadget creato, per renderlo accessibile dagli utenti.
- Utente:
  - La ricerca (eventualmente semantica) del gadget che più si avvicina alle sue esigenze
  - L'installazione di un nuovo gadget
  - L'esecuzione delle operazioni proprie del gadget usato

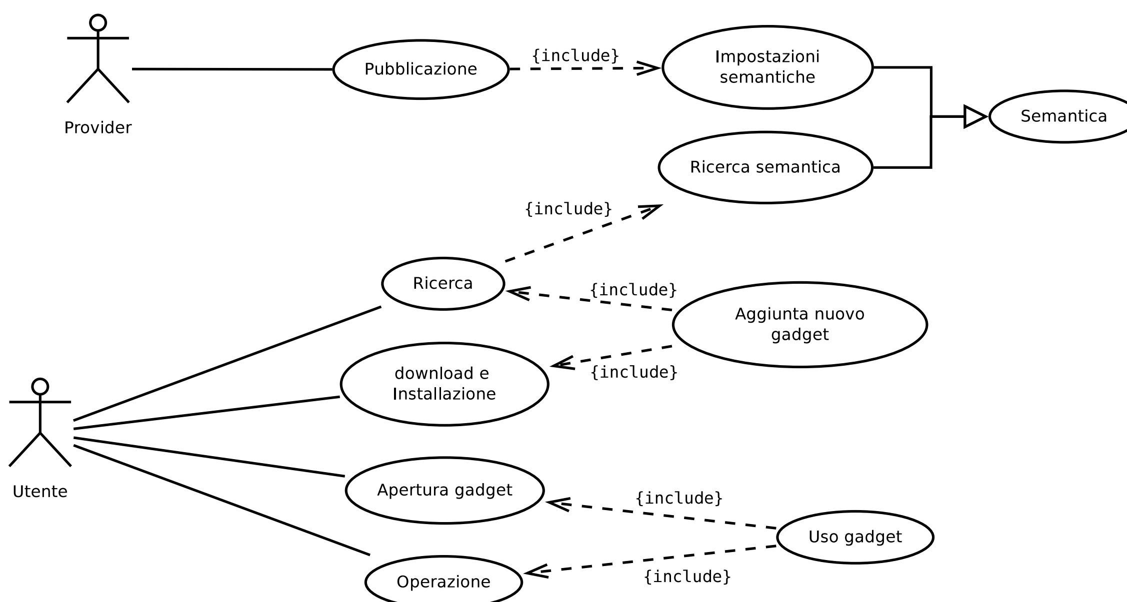


Figura 1.6: Gadget - Use Case Diagram: Utente - Livello 0

#### 1.5.1.1 Pubblicazione nuovo gadget

| Nome | Pubblicazione |
|---|---|
| Attori coinvolti | Provider |
| Scopo | Rendere disponibile un nuovo gadget per gli utenti |
| Descrizione sintetica | Il Provider inserisce all'interno del sistema di pubblicazione il nuovo gadget sviluppato |
| Pre-condizioni | Gadget non presente nel parco gadget |
| Flusso eventi | 1. Il Provider sviluppa il gadget<br>2. Il provider effettua l'upload del gadget attraverso il sistema messo a disposizione<br>3. Il provider struttura le informazioni semantiche per permettere la ricerca da parte degli utenti |
| Post-condizioni | Gadget disponibile per gli utenti |

#### 1.5.1.2 Aggiunta nuovo gadget

| Nome | Aggiunta nuovo gadget |
|---|---|
| Attori coinvolti | Utente |
| Scopo | Trovare ed aggiungere di un nuovo gadget all'interno dell'applicazione dell'utente |
| Descrizione sintetica | L'utente cerca e installa un nuovo gadget all'interno della sua applicazione |
| Pre-condizioni | Gadget non presente nell'applicazione utente |
| Flusso eventi | 1. L'utente cerca il gadget<br>2. L'utente sceglie il gadget che lo soddisfa<br>3. L'utente installa il gadget |
| Post-condizioni | Gadget installato nel sistema dell'utente |

#### 1.5.1.3 Ricerca

| Nome | Ricerca |
|---|---|
| Attori coinvolti | Utente |
| Scopo | Ricerca di un nuovo gadget da installare |
| Descrizione sintetica | L'utente cerca un nuovo gadget all'interno della sua applicazione |
| Pre-condizioni | Gadget non presente nell'applicazione utente |
| Flusso eventi | 1. L'utente si collega al sistema di distribuzione gadget<br>2. L'utente immette voci di filtro<br>3. Il sistema esegue una ricerca basata sulla semantica<br>4. Il sistema restituisce a video i gadget selezionati<br>5. L'utente può:<br>a) Tornare al punto 2<br>b) Selezionare uno dei gadget proposti |
| Post-condizioni | - |

#### 1.5.1.4 Download ed installazione

| Nome | Download ed installazione |
|---|---|
| Attori coinvolti | Utente |
| Scopo | Installazione di un gadget scelto |
| Descrizione sintetica | L'utente installa il nuovo gadget nel proprio sistema |
| Pre-condizioni | Gadget non presente nell'applicazione utente. L'utente ha già davanti a se il gadget da installare nel sistema di distribuzione |
| Flusso eventi | 1. L'utente avvia il download dal relativo pulsante<br>2. Il sistema si preoccupa di tutto il resto (download, installazione, linking..) |
| Post-condizioni | - |

### 1.5.2 Requisiti

Il 2lvlOS deve provvedere a fornire ad ogni gadget:

1. Distribuzione e installazione assistite ed automatizzate
    a) Meccanismo di riconoscimento delle funzioni esposte dal gadget
2. Accesso controllato alle risorse del sistema, in particolare a:

   a) FileSystem
   b) Rete
   c) Database
   d) Chiavi di configurazione

3. Supporto *adaptive engine* su tutte le operazioni di scelta (dove con *adaptive engine* si intende il meccanismo di adattamento del sistema all'utente, la previsione delle sue necessità e delle scelte che effettuerà).

4. Supporto multilingua

   a) per stringhe statiche
   b) per dati dinamici

5. Supporto alla comunicazione tra gadget

   a) Comunicazione locale
   b) Comunicazione remota

6. Supporto alla UI concettuale (sarà il 2lvlOS a tradurla in GUI fisica)

## 1.6 Conclusioni

Come già brevemente citato le esigenze di utenti e sviluppatori evidenziano le forti limitazioni che affliggono gli odierni pattern di sviluppo, tanto che neppure le soluzioni proposte dai grandi produttori internazionali di software sono in grado di soddisfare pienamente.

Queste soluzioni, ha evidenziato l'analisi condotta, sono state certamente indispensabili non solo perché permettono a milioni di applicazioni di funzionare correttamente, ma anche e soprattutto perché rappresentano la naturale evoluzioni delle tecniche e degli strumenti già presenti da anni; tuttavia i tempi sono maturi per qualcosa di nuovo, qualcosa che non faccia uso di trucchetti e workaround per soddisfare le esigenze espresse, ma che sia naturalmente adeguato a farlo, grazie ad una solida progettazione che guardi ai prossimi 5 anni almeno.

Il "qualcosa di nuovo" è stato individuato come un sistema operativo di secondo livello: un applicazione del sistema applicativo reale della macchina che si comporti come un secondo OS per le proprie applicazioni, chiamate gadget. Se il sistema operativo di primo livello espone interfacce di medio-basso livello, questo nuovo OS dovrà esporre interfacce di alto-altissimo livello, astraendo completamente i gadget dalle caratteristiche proprie della macchina e fornendogli supporto a 360° per tutte le operazioni più comuni, qualora anche complesse: dall'accesso alla network, al supporto multilingua, al supporto per l'adattamento automatico dell'applicativo alle caratteristiche dell'utente che lo sta usando.

In questo modo i gadget saranno veloci e facili da sviluppare, come ogni buon programmatore si augura, ma allo stesso tempo completi e facili da cercare, installare ed utilizzare, come ogni utente sogna.

# Parte I

# Sviluppo di 2lvlROS

# 2 Architettura

*Prima di addentrarsi in una progettazione più specifica, che occuperà diversi capitoli, è opportuno avere una conoscenza generale dei flussi d'informazione che interesseranno i vari componenti del sistema, almeno a livello concettuale. Ciò permetterà non solo di identificare il ruolo che i diversi componenti giocano nel sistema, differenziando quindi quelle che sono le responsabilità dei Gadget rispetto alle responsabilità del 2lvlOS, ma faciliterà anche la comprensione dell'organizzazione delle classi in package.*

## 2.1 Architettura di base

### 2.1.1 Flusso Dati

L'architettura di base del sistema (intesa come flusso dati) è quella mostrata nel DFD in figura 2.1: l'utente interagisce solo con l'interfaccia grafica, creata dal 2lvlOS sulla base delle richieste del gadget, il quale comunicherà con la sorgente dati d'interesse. In questo modo l'utente lavora in un ambiente uniforme e standardizzato, indipendente dalla natura della sorgente dati di origine.

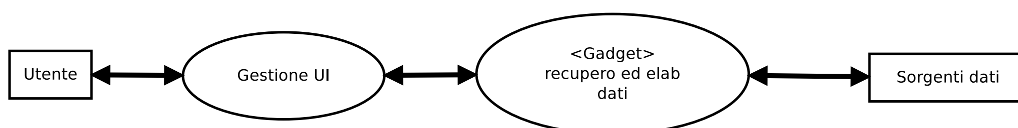


Figura 2.1: Architettura - DFD - Livello 0

Analizzando più in dettaglio si evidenziano:

- Una base dati condivisa dell'utente che fornisce il supporto per i dati condivisi tra le diverse entità di elaborazione
- Una base dati locale dedicata per ogni entità applicativa
- Sorgenti dati di diversa natura remote.
- Una entità (opzionale) di elaborazione remota, per ogni entità di elaborazione locale.

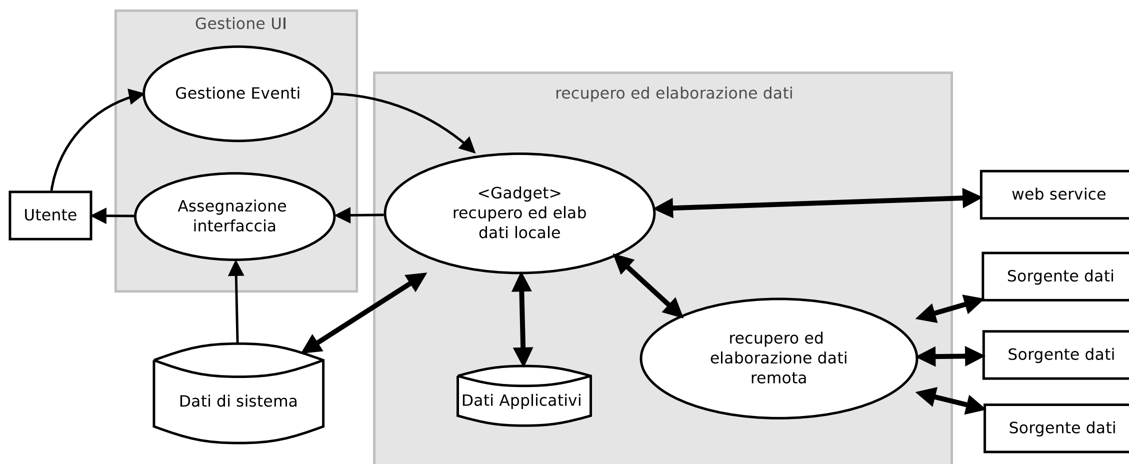


Figura 2.2: Architettura - DFD - Livello 1

### 2.1.2 Responsabilità

Ciò che il flusso dati di figura 2.2 non mostra, ma che è necessario per definire una adeguata organizzazione delle classi, è chi sono le parti attive, cioè chi effettivamente sposta ed elabora i dati.

Un'analisi in questo senso (vedi figura 2.3) rileva che il gadget è completamente abbracciato dal sistema operativo di secondo livello: nulla può essere fatto dal gadget se non attraverso il 2lvlOS (così come specificato dai requisiti), tanto che lo stesso utente interagisce con il gadget solo attraverso opportuni moduli di sistema che fungono da mediatori.

Questa particolare struttura permette tra le altre cose:

- di astrarre il gadget da tutti i compiti di basso livello legati all'input / output dei dati

- di uniformare l'interfaccia utente rendendola indipendente dall'applicativo in uso

- di implementare meccanismi di sicurezza

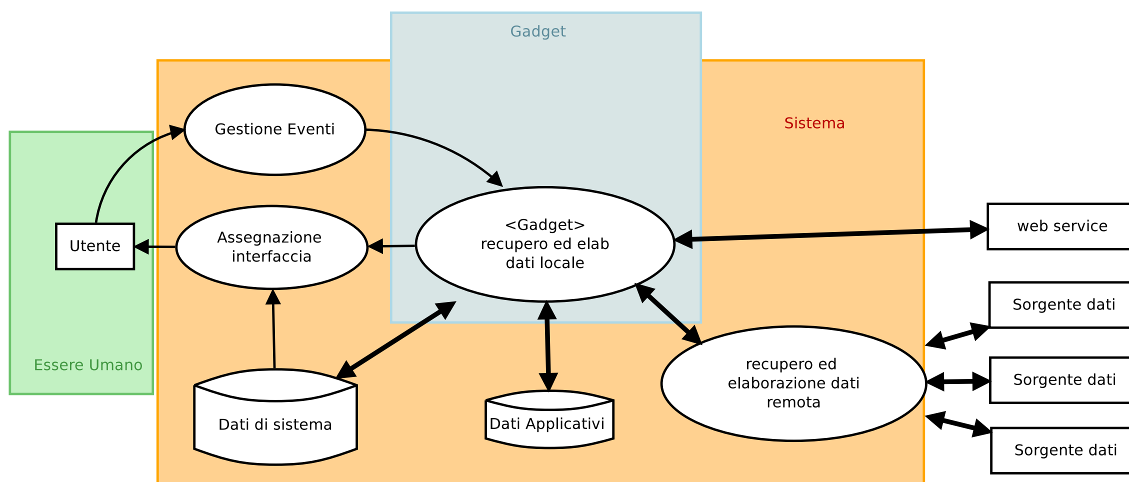


Figura 2.3: Architettura - DFD - Livello 1 - Aree di responsabilità

## 2.2 Organizzazione delle classi

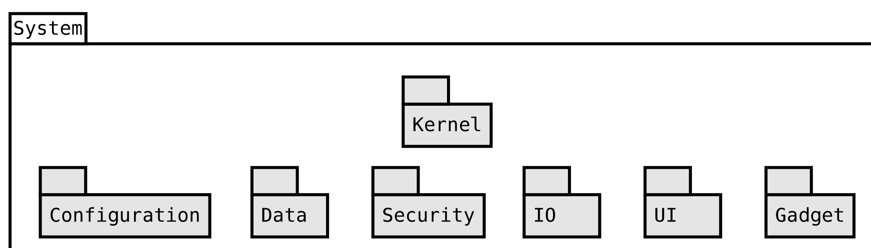


Figura 2.4: Diagramma delle classi - Livello 0

Le classi che comporranno il sistema saranno organizzate in package (vedi figura 2.4), in modo da riflettere il ruolo che ogniuna gioca all'interno del sistema stesso. In particolare si possono identificare 3 grandi categorie:

**Classi Gadget:** ogni gadget sarà definito da una o più classi.

**Classi Framework:** classi il cui scopo è fornire supporto ad altre entità, prime tra tutte i gadget, relativamente alla propria area di competenza. A questa categoria appartengono:

- System.Configuration
- System.Data
- System.Security
- System.IO
- System.UI
- System.Gadget
- System.Multitasking

**Classi di sistema:** classi che compongono il kernel del 2lvlOS. A differenza delle classi di framework, istanziate all'occorrenza, tipicamente le classi di sistema hanno una sola istanza attiva, generata al boot. Tali classi sono raccolte in:

- System.Kernel

# 3 Gadget

*L'intero sistema può essere visto come la combinazione di 3 elementi: i Gadget, il Kernel, e il Framework. In questa sede si analizzerà lo sviluppo e l'implementazione del primo di questi.*

*Chiaramente nessun gadget potrà mai funzionare senza che alle spalle vi sia un sistema che lo supporti, ma analizzarli per primi permette di definirli in modo che soddisfino le necessità di utenti e sviluppatori senza il rischio di essere distorti da problemi implementativi legati al sistema; in altre parole in questo capitolo ci si concentra su come dovrà essere il gadget ideale per essere veloce da sviluppare e semplice da usare, come questo possa essere implementato e quali strutture si debbano mettere in piedi perché funzioni è un problema che sarà affrontato successivamente.*

## 3.1 Nomenclatura

Per assegnare un nome univoco ai gadget si utilizzerà un approccio simile a quello usato in ambiente Java: ogni gadget sarà identificato da una terna di valori:

- Un namespace che ne identifica l'autore, nella forma puntata inversa (es: it.unipd.dei)
- Un nome, definito dal nome del modulo di codice principale, che identifica l'oggetto
- Un numero di versione nella forma Major Version, Minor Version, Build: $xx.xx.xxxx$

**Esempio:** it.unipd.dei.mail v.0.8.23 per un gadget che implementa un client di posta giunto alla versione 0.8.23 e prodotto dal Dipartimento di Ingegneria dell'Informazione dell'Università di Padova.

## 3.2 Tipi di gadget

La definizione di gadget è molto generica e permette di sviluppare oggetti, strutturalmente identici, ma concettualmente e visivamente molto diversi tra loro. Raggruppandoli per le funzionalità che espongono è possibile identificare 4 famiglie:

***Gadget applicativi*:** sono i "normali" gadget fin qui intesi, quelli cioè che l'utente seleziona per svolgere funzioni accessorie (orologio, sveglia, meteo ecc ecc). Normalmente sono pressoché indipendenti dal sistema e non necessitano di particolari permessi per l'esecuzione.

***Gadget di servizio***: sono gadget privi di interfaccia grafica che svolgono funzioni in background di sistema. Sono prevalentemente preinstallati, verificati e protetti (l'utente non può manipolarne il comportamento) e hanno elevati permessi per l'esecuzione (possono accedere ai dati condivisi, ai dati di altri gadget ecc)

***Gadget amministrativi***: sono gadget che possiedono interfaccia grafica ma espongono funzioni di configurazione e amministrazione del sistema; per questo motivo godono di elevati privilegi.

***Gadget desktop***: con questo termine si intende il gadget che effettivamente implementa il concetto di desktop nel sistema, che gestisce lo spazio grafico dello schermo assegnandolo ai diversi gadget applicativi e amministrativi.

## 3.3 Gadget: reti di Petri

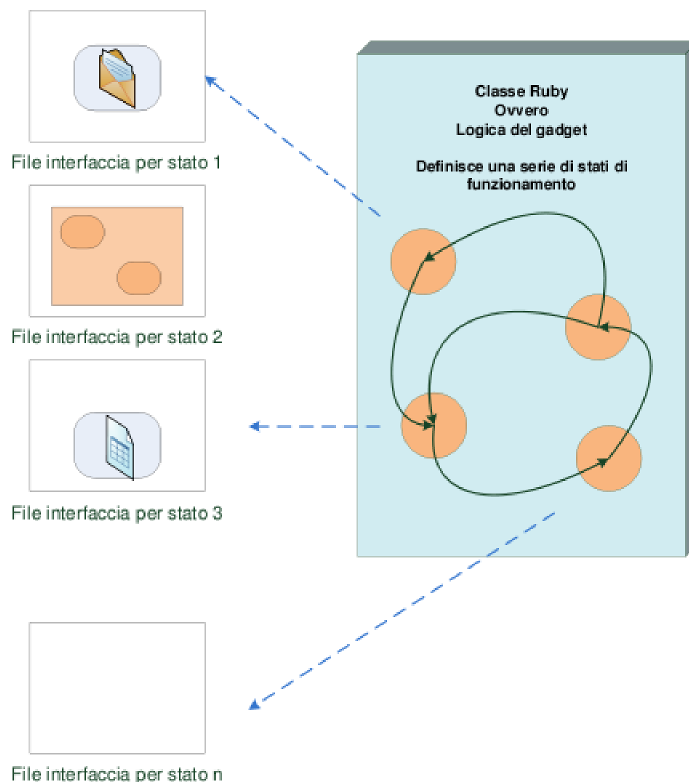


Figura 3.1: Gadget: Struttura concettuale

Per rendere lo sviluppo dei Gadget semplice ma scalabile è necessario adottare un modello concettuale che sia al tempo stesso preciso e flessibile, così da rendere la creazione dei gadget veloce ed omogenea. Nello specifico è stata scelto un adattamento delle reti di Petri che verrà qui presentato:

**State**: ogni posto della rete di Petri rappresenta uno stato di funzionamento del gadget,

concettualmente un'operazione che l'utente deve eseguire. Ad ogni stato è possibile (non obbligatorio) associare:

1 template grafico per la visualizzazione in *main view*[1]

1 template grafico per la visualizzazione in *dock view*

Quando il gadget si troverà in quel particolare stato potrà essere posto in *main* o in *dock view* solo se queste visualizzazioni sono state implementate per quel particolare stato. Ovviamente non tutti i gadget definiranno forme grafiche per i loro stati:

- *Gadget applicativi e amministrativi*: definiranno almeno un template grafico (spesso entrambi) per la maggior parte dei loro stati.
- *Gadget di servizio*: per definizione lavorano senza mostrare alcunché, pertanto non definiranno mai template grafici.
- *Gadget desktop*: lavorando unicamente in *main state* (anche se il significato di main state viene un po' distorto) definirà solo questi template grafici.

**Transition:** esattamente come nelle reti di Petri classiche, una transizione scatta se e solo se tutti gli stati di ingresso la abilitano, portando il sistema in tutti gli stati di uscita contemporaneamente. Per ogni transizione, quindi, devono essere specificati il vettore di stati di ingresso (la transizione scatterà quando l'ultimo dei suoi stati effettuerà l'operazione di *enableTransition*), il vettore degli stati di uscita (se sono più d'uno vengono istanziati ulteriori thread per permettere l'abilitazione contemporanea di tutti gli stati) e l'eventuale funzione da eseguire nello scatto.

**Posti esterni:** nel modello è previsto anche l'uso di posti esterni, concettualmente analoghi agli omonimi elementi delle RdP classiche:

**Stato "InitState":** è un particolare posto esterno in ingresso che rappresenta lo stato iniziale del gadget e che pertanto sarà sempre presente. Quando il sistema avvia il gadget automaticamente lo porterà in questo stato che avrà il compito di generare i primi *enableTransition.*

**Stato "ExitState":** è un particolare posto esterno in uscita che permette la morte dei flussi esecutivi. Quando una transizione scattando abilita questo stato, quel particolare thread viene terminato (una sorta di buco nero per le marche).

**TimerState:** è un tipo di posto esterno in ingresso che "genera marche a tempo". Implementa 3 primitive di controllo: *enableOn(delay)* per abilitare tutte le transizioni collegate dopo *delay* tempo dalla chiamata (una sola volta), *enableEvery(interval)* per abilitare tutte le transizione collegate ogni *interval* tempo, *disable()* per fermare la produzione di marche dopo la primitiva *enableEvery* o per annullare una chiamata *enableOn* non ancora giunta allo scatto.

**SoftwareState:** è un tipo di posto esterno in ingresso utilizzato nella comunicazione tra diversi gadget. Tutte le funzioni definite in stati di questo tipo sono visibili

[1] *Vedi approfondimento in sezione 3.4*

dall'esterno del gadget e obbligatoriamente devono portare la firma *"HiperCube nomeFunz(HiperCube)"*[2] (ovviamente queste funzioni possono richiamarne altre o abilitare transazioni, quindi uscire dal posto esterno). In ogni stato di questo tipo, inoltre, è possibile applicare le primitive *enable()* e *disable()* per rendere o meno effettivamente richiamabili dall'esterno tutte le funzioni contenute. Si possono inoltre applicare le primitive enableRemote() e disableRemote() per definire l'accessibilità delle medesime funzioni da gadget remoti.

Può essere utile a questo punto fare una piccola precisazione sulla terminologia utilizzata: quando si parla di funzione si intende un puntatore a un blocco di codice eseguibile o a una transizione: in questo modello, infatti, eseguire un blocco di codice o abilitare una transizione sono operazioni concettualmente così vicine da giustificare l'interscambiabilità (entrambe rappresentano l'operazione "fai qualcosa"). Per i puristi della programmazione si noti che associare una transizione ad una funzione può essere visto come una rappresentazione compatta di un tradizionale blocco di codice contenente un'unica istruzione *enableTransition*.

### 3.3.1 Stati ed eventi

Si possono a questo punto definire degli eventi, delle particolari funzioni richiamate direttamente dal 2lvlOS in determinate situazioni:

**State.OnEnter** evento scatenato all'ingresso di uno stato, indispensabile per sviluppare il comportamento di stati particolari come *InitState* e *TimerState* ma utile anche in situazioni più normali.

**Transition.OnTrigger** evento scatenato allo scattare di una transizione.

Esempi di applicazioni di questi eventi possono essere:

**InitState.OnEnter()** per eseguire del codice all'avvio del gadget.

### 3.3.2 Sviluppo scalabile

Per facilitare la scalabilità dello sviluppo, ovvero per non rischiare che il programmatore soffra di attacchi di panico nella revisione di grossi applicativi, è possibile suddividere la complessità di un gadget in più componenti, chiamati *component* appunto, completando così il modello delle reti di Petri:

- Ogni *component* può essere sviluppato come un piccolo gadget a sé stante: risiederà in un modulo di codice separato e sarà modellano anch'esso attraverso una rete di Petri con un posto *InitState* e un posto *ExitState* (che rappresentano i punti di attivazione e chiusura analogamente a quanto avviene per il gadget principale).

[2] *HiperCube* è una struttura dati "standard", che verrà presentata nella sezione 5.2. Per ora non è necessario conoscerne i dettagli, basti sapere che è in grado di memorizzare dati complessi e strutturati.

- Il gadget principale tratterà ogni *component* come un normale oggetto: potranno essere definiti all'interno di uno specifico stato, o a livello di gadget, verranno avviati (*InitState*) alla creazione e verranno distrutti alla chiusura (*ExitState*). L'interfaccia grafica del gadget può dedicare dello spazio ad un *component* semplicemente inserendolo all'interno di un *Panel*.

- è possibile pensare alla Rete di Petri di un *component* come all'esplosione di uno stato del gadget principale (caso di oggetto definito nello stato stesso) o come flusso parallelo (caso di oggetto definito a livello di gadget).

Si noti che in questo modo i component non differiscono molto dai gadget, almeno per quanto riguarda la loro modellazione, e questo permette ai gadget desktop di inserire al proprio interno (anche come parte della propria UI) altri gadget, esattamente come se si trattasse di normali component, senza alcuna modifica o deroga al modello presentato.

## 3.4 Interfaccia Grafica

Il gadget si dovrà presentare in due diverse forme:

- Main view

- Dock view

Il programmatore, in fase di design del gadget, sarà libero di definire quando l'utente può scegliere in quale visualizzazione tenere il gadget, quando obbligare l'uso della sola main view o della sola dock view.

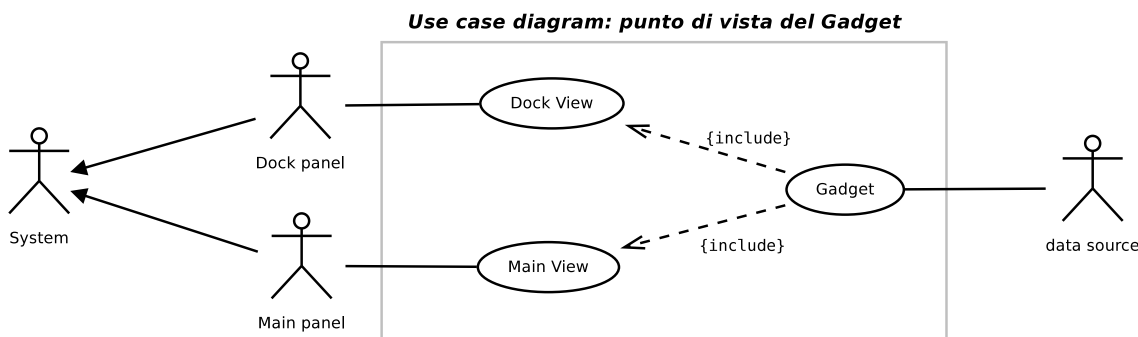


Figura 3.2: Gadget - UI - Use Case Diagram: Gadget - Livello 0

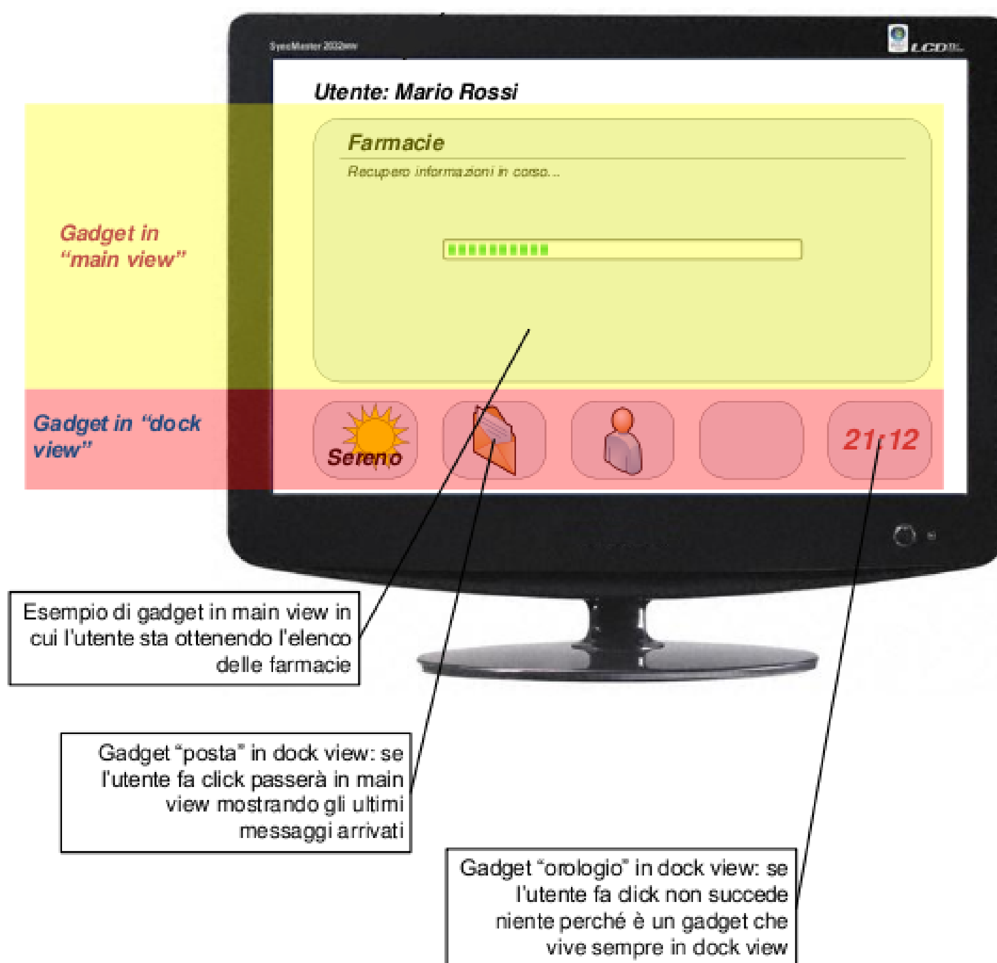


Figura 3.3: Gadget - UI: Esempio di interfacce Main view e Dock view

### 3.4.1 Main view

Si tratta della modalità "normale" di utilizzo: il gadget occupa la maggior parte dello schermo dell'utente e detiene il focus operazionale (concettualmente l'utente è su quel gadget che sta lavorando).

Dal punto di vista concettuale è ammesso un solo gadget in main view ad un certo istante di tempo $t$ e tale gadget rappresenta l'operazione che in quel momento l'utente sta eseguendo. Per praticità, tuttavia, si impone l'estensione di questo concetto per accettare più gadget in main view sovrapposti, con il vincolo che l'utente sia in grado di interagire solo con il gadget che si trova in cima alla pila.

### 3.4.2 Dock view

Il *dock view* è una modalità di utilizzo del gadget compatta, in cui espone in uno spazio limitato chiamato *slot* un'icona (anche dinamica) e, al focus, un tooltip descrittivo.

Gli *slot* sono organizzati sullo schermo in *dock panels*, ma ogni gadget ha a disposizione al più uno *slot*.

### 3.4.3 Implementazione dell'interfaccia grafica

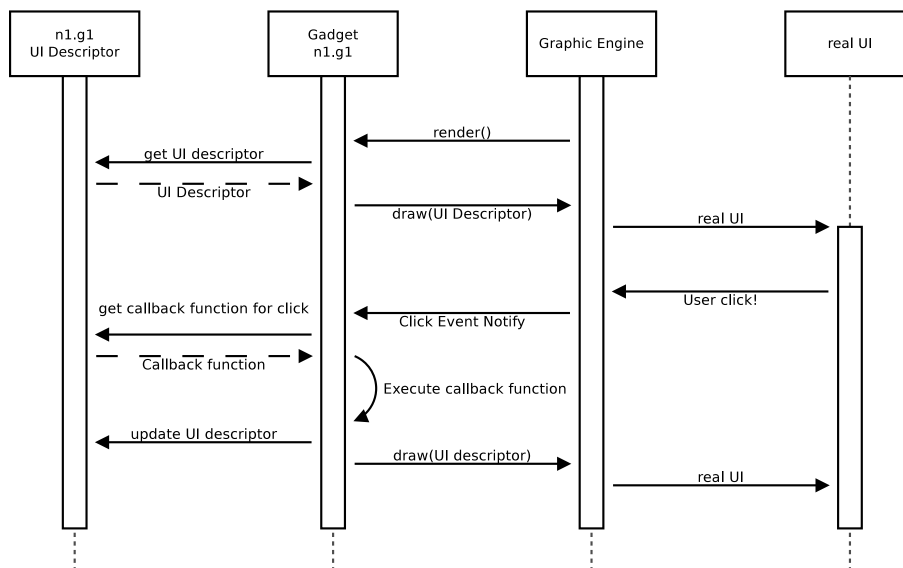


Figura 3.4: Gadget - UI - Diagramma di sequenza: Processo di rendering

Come si vede dal diagramma in figura 3.4 ogni gadget avrà un proprio "UI Descriptor" in cui descriverà a livello concettuale la propria interfaccia grafica, in modo indipendente dal sistema. Sarà quindi compito del motore di rendering tradurre questa descrizione astratta nella UI reale del sistema, sia essa nativa o personalizzata.

In questo modo è possibile non solo rendere i gadget indipendenti dal sistema operativo in cui lavoreranno, ma anche sostituire in qualunque momento il motore di rendering del 2lvlOS, qualora la scelta fatta non sia soddisfacente (per esempio se vi è la necessità di usare una grafica più accattivante rispetto alla grafica nativa del SO).

## 3.5 Gadget e funzionalità

Per permettere agli utenti di individuare facilmente i gadget di cui hanno bisogno è indispensabile creare un'associazione tra le loro necessità e le funzioni che i gadget espongono, in modo da rendere possibile il recupero dei soli applicativi che le soddisfano (la situazione ideale è che il sistema proponga all'utente tutti e soli i gadget che soddisfano tutte e sole le necessità espresse).

Questo meccanismo, potrà forse sembrare banale, ma in realtà sottintende una serie di prerequisiti tutt'altro che di semplice progettazione:

1. Ad ogni gadget deve essere associato il proprio set di funzionalità, anche nei casi in cui:

a) un gadget estenda le funzionalità di un'altro

b) un gadget utilizzi una o più funzionalità di un'altro

c) le funzionalità che due gadget espongono siano parzialmente o completamente sovrapporsi

d) una funzionalità sia un caso particolare di un'altra funzionalità

2. Deve esistere il concetto di set di funzionalità e di distanza tra set di funzionalità

Dalla struttura di un gadget non è semplice identificare le funzionalità che questo espone: il concetto umano di funzionalità è tipicamente molto distante dal codice che la implementa e anche in questo caso, nonostante la relativa complessità del modello utilizzato, un gadget ben progettato può fornire indicazioni sulle operazioni che supporta (sono i suoi stati), ma nulla o quasi sulle funzionalità che espone.

Si prenda ad esempio un gadget con i seguenti stati:

- Search
- Details

Senza analizzarne semanticamente il nome, *Farmacie*, è praticamente impossibile capire che si sta parlando di un applicativo che recupera le informazioni relative alle farmacie presenti nel territorio.

La soluzione più semplice ed efficace al problema è quindi quella di lasciare allo sviluppatore, al momento della pubblicazione, il compito di assegnare il set di funzionalità al gadget.

### 3.5.1 Sistema di pubblicazione

Il sistema di pubblicazione dei gadget sarà quindi composto da:

- Un reposity centrale contenente tutti i gadget disponibili e le relative informazioni ausiliarie
- Una procedura guidata di pubblicazione che effettui l'upload del nuovo gadget e chieda di specificarne le funzionalità
- Un gadget amministrativo pre-installato in tutte le macchine che, dato un set di funzionalità richieste, cerchi nel reposity i gadget che meglio le soddisfano.

Il reposity centrale è costituito da uno spazio virtuale n-dimensionale in cui ogni gadget è chiamato ad occuparne determinate coordinate in base alle funzioni che espone, in questo modo:

| | | |
|---|---|---|
| procedura di pubblicazione | ► | attribuzione delle corrette coordinate al gadget in base alle funzionalità che espone |
| ricerca dei gadget che meglio rispondono al set di necessità date | ► | proiezione delle necessità espresse nello spazio virtuale e recupero dei gadget che si trovano a minima distanza |

#### 3.5.1.1 Il modello

Il modello utilizzato per tradurre le funzionalità in coordinate dello spazio n-dimensionale (chiamato *universo*) è simile simile agli ipercubi trattati nella sezione dedicata al data flow (5.2):

**Definizione.** (di Visione) Si definisce visione un albero omogeneo di funzionalità, che cataloghi l'universo secondo un determinato criterio, chiamato appunto *criterio della visione*. Ogni nodo di una visione individua tutti e soli i gadget dell'universo che espongono quella particolare funzionalità o una qualsiasi delle funzionalità dei propri discendenti.

- Più i nodi sono vicini alla radice più rappresentano funzionalità generali, più sono vicini alle foglie e più rappresentano funzionalità specifiche
- La root di ogni visione (come ci si aspetta) individua tutti i gadget dell'universo che possono essere catalogati attraverso il criterio della visione (cioè su cui ha senso applicare il criterio, vedi esempio in figura 3.5).

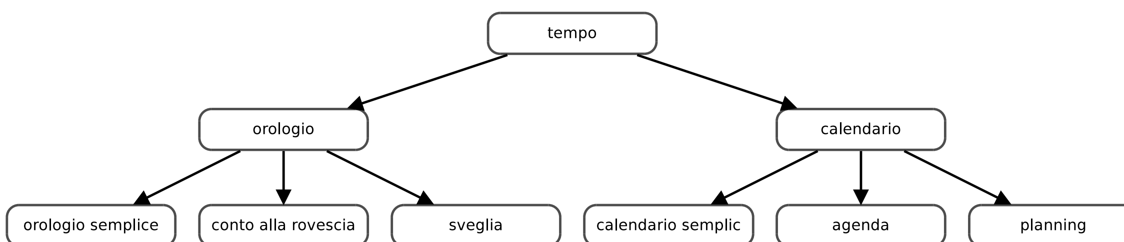


Figura 3.5: Pubblicazione dei gadget: esempio di visione

- Ogni dimensione dello spazio n-dimensionale viene associata ad una diversa visione.
- Ogni gadget deve essere associato ad almeno una visione, ovvero deve avere posizione nota in almeno una dimensione dello spazio, ovvero deve esporre almeno una funzionalità.
- Ogni gadget può essere associato anche a più nodi in una stessa visione, ovvero può essere presente in più coordinate in una stessa dimensione, ovvero può esporre più funzionalità omogenee.

- Ogni gadget deve essere associato al nodo che meglio rappresenta la funzionalità che realmente espone, ovvero al nodo più vicino alle foglie dell'albero che la rappresenta.

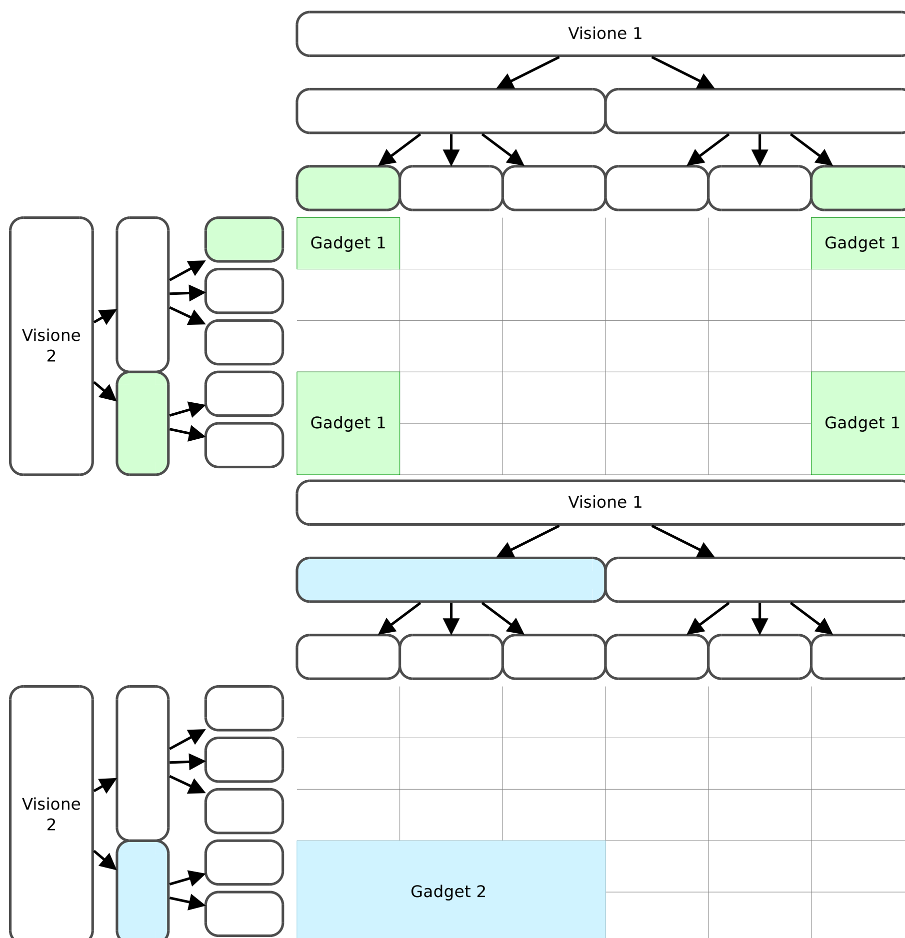


Figura 3.6: Pubblicazione dei gadget: esempi di assegnazione dei gadget alle visioni

La procedura di pubblicazione di nuovi gadget, quindi, si limita a chiedere al provider del gadget stesso la sua collocazione nell'universo mediate la scelta dei nodi a cui associarlo per ogni criterio.

Per non rendere alcuni gadget più raggiungibili di altri, però, vi è un importante vincolo: ad ogni associazione gadget-coordinata (o equivalentemente gadget-funzionalità) vi si attribuisce un peso, tale per cui la somma di tutti i pesi di un gadget deve essere 1 (si può parlare quindi di distribuzione di funzionalità). In questo modo si può attribuire un peso anche ad ogni collocazione nello spazio di un gadget (si ricordi che può averne più di una), semplicemente moltiplicando i pesi delle coordinate che la compongono.

Più interessante è la ricerca dei gadget che meglio soddisfano un set di necessità (di funzionalità) date:

1. Si individua il punto (o i punti) di ricerca, generando dinamicamente le coordinate che rappresentano tale set di funzionalità

2. Si restituiscono i primi n gadget (in ordine) che ottengono il massimo punteggio in base alla formula:

$$(puntGadget) = \sum_{collocazioniDelGadget} (distDaPuntoRicerca)(pesoCollocazione)$$

In questo modo è possibile attribuire la giusta rilevanza ai gadget che espongono tutte e sole le funzionalità richieste rispetto ai gadget che ne espongono molte di più.

Chiaramente la struttura dell'universo (l'insieme delle visioni) deve essere unica a livello mondiale per garantire la consistenza delle collocazioni dei gadget e non può dipendere dalla lingua con cui le funzionalità sono definite, ma questo non è un grande problema se si ha l'accortezza di definire le funzionalità per mezzo di identificativi e presentarle di volta in volta tradotte nella lingua opportuna.

### 3.5.2 Software State: funzionalità verso altri gadget

#### 3.5.2.1 Il Problema

Si immagini ora la seguente situazione: un determinato gadget applicativo abbia necessità di ottenere un accesso gestito al file system e si appoggi, come da consuetudine, ad un gadget di sistema che ne fornisce (mediante softwareState) le primitive di lettura e scrittura. Non solo gli autori del gadget applicativo e del gadget di sistema sono diversi ma, anzi sono disponibili più gadget di sistema che forniscono un'interfaccia controllata al disco, sviluppati autonomamente da enti diversi.

In questa situazione tutti i gadget di sistema interessati espongono più o meno le stesse funzionalità (le prestazioni potranno variare, ma le funzionalità esposte grossomodo coinciderebbero), tuttavia sebbene la firma di tutte le funzioni esposte tramite i softwareState sia unica e standardizzata non vi è alcun vincolo sul nome della funzione, ne sulle modalità di interpretazione dei parametri di input e di output:

- Il gadget *gestioneDisco1* potrebbe esportare la primitiva *hq_out=read(hq)*, aspettandosi il nome del file in *hq[filePath]* e ritornandone il contenuto in *hq_out[file]*
- Il gadget *gestioneDisco2* potrebbe esportare la primitiva *hq_out=readFile(hq)*, aspettandosi il nome del file in *hq[path]* e ritornandone il contenuto in *hq_out[fileContent]*

Gli autori di entrambi i gadget, completato lo sviluppo, provvederanno alla loro pubblicazioni, indicandone le funzionalità (che supporremo coincidenti). In ogni macchina sarà quindi presente solo uno dei due gadget, in quanto l'uguaglianza delle funzionalità esposte rende di fatto inutile la presenza del secondo.

A questo punto però, sorge il problema: gli sviluppatori del gadget applicativo a quale chiamata dovranno interfacciarsi? Se utilizzassero *read()* creerebbero un gadget non

funzionante in tutti quei sistemi in cui la gestione del disco avviene mediante *gestioneDisco2*, ma simmetricamente se si allineassero alle specifiche di *readFile()*, il gadget non funzionerebbe in presenza del solo *gestioneDisco1.*

Si noti che, nonostante le difficoltà evidenziate, le funzionalità che i system gadget espongono sono ben definite, a tal punto da essere riconosciute coincidenti dal sistema stesso, e che un essere umano non avrebbe alcuna difficoltà a individuare la corretta chiamata da effettuare in presenza di uno o dell'altro gadget, che a ben vedere non possono essere così diverse.

Da queste riflessioni nasce un naturale completamento del modello utilizzato per la pubblicazione dei gadget, che tenga conto anche di queste nuove esigenze.

#### 3.5.2.2 La soluzione

Se si pensa ai gadget che fanno uso di *softwareState* come applicativi dedicati ad un utente software (il gadget che con loro colloquierà) invece che ad un essere umano, diviene naturale aggiungere ad ogni nodo di ogni visione tutte quelle informazioni che una persona è in grado di dedurre dal significato semantico delle funzionalità in oggetto, ma che un elemento software non è (ancora) in grado di fare.

Assumendo che gadget che espongono le stesse funzionalità, esporranno concettualmente anche le stesse funzioni (cioè che qualsiasi gadget di gestione del disco ad esempio avrà le primitive per leggere e per scrivere un file, qualsiasi sia il loro nome), le informazioni ausiliarie da includere nei nodi si limiteranno alla *firma interna* delle chiamate ritenute necessarie e sufficienti per l'implementazione della funzionalità che il nodo stesso sottintende:

**Definizione.** (di firma interna) Si definisce firma interna di una funzione *HQout = nomeFunz(HQin)* il nome e il significato della funzione (*nomeFunz*) , il nome e il significato delle dimensioni utilizzate nei due ipercubi *HQin* e *HQout*.

Per chiarire il concetto si pensi ai soliti gadget di gestione del disco: in una delle visioni presenti nell'universo vi sarà un nodo dedicato alla "gestione del disco" allegato al quale saranno definite le primitive (complete di documentazione)

**hqout=read(hqin)** che legge il file dal percorso specificato in *hqin[path]* e restituisce il contenuto del file in *hqout[content]*

**hqout=write(hqin)** che scrive il contenuto specificato in *hqin[content]* nel file identificato da *hqin[path]* e restituisce in *hqout[return]* se l'operazione ha avuto successo

Al momento della pubblicazione di un gadget (di gestione del disco) che fa uso di software state, la procedura di pubblicazione:

1. chiederà al provider che sta pubblicando il gadget se le operazioni di gestione del disco sono accessibili mediante softwareState
2. in caso di risposta affermativa, verrà effettuato un matching tra le funzioni read e write "standard", con quelle realmente definite nel gadget. Tale matching potrà essere

automatico, semiautomatico o completamente manuale, a seconda della potenza del sistema di pubblicazione, del tipo di gadget ecc..

In questo modo i gadget applicativi che necessiteranno di un accesso al file system potranno referenziare le firme standard dicchiarate dal sistema stesso, sicuri che tutti i gadget che implementano ed espongono quelle funzionalità sono conformi a tali definizioni.

## 3.6 Gerarchia di Gadget

Come si è visto, il meccanismo utilizzato per individuare le funzionalità esposte dai gadget non dipende dalla superclasse da cui i questi ereditano, lasciando lo sviluppatore libero di utilizzare l'ereditarietà per questione più tecniche, come i sani principi di programmazione ad oggetti prevedono.

È tuttavia necessario prendere in esame due casi:

- Il gadget eredita da GadgetBase
- Il gadget eredita da un altro gadget

Nel primo caso sarà necessario sviluppare tutte le funzionalità del gadget da zero, mentre nel secondo si erediteranno tutti gli stati, i metodi e le variabili non private dal supergadget[3], così come prevede la programmazione ad oggetti.

In entrambi i casi non sussistono vincoli che leghino le funzionalità del gadget figlio con le funzionalità del gadget padre.

## 3.7 Implementazione

Dal punto di vista del programmatore un gadget sarà costituito da un insieme di file e cartelle:

**<dir> Code** : contiene tutti i file di codice del gadget

**<dir> UI** : contiene tutti i file che descrivono l'interfaccia grafica del gadget

**nomeGadget.yaml** : è un file contenente tutte le informazioni ausiliarie che descrivono il gadget (nome, autore, versione, funzionalità..)

**setup.rb** : un file contenente le procedure di installazione e rimozione del gadget

### 3.7.1 Code

La cartella code contiene il file principale di codice, *_nomeGadget.rb*, e il codice relativo ai diversi components, *nomeComponent.rb*.

La tipica struttura del file di codice principale di un gadget è:

[3] Supergadget: scrittura compatta per indicare *il gadget rappresentato dalla superclasse del gadget in oggetto*

```
1  class gagdetX < gadgetBase
2     # DEFINIZIONE DEL VETTORE DELLE TRANSAZIONI
3     # Specificando per ogni transition gli input e gli output
        state
4     # ed eventualmente anche il codice da eseguire durante lo
        scatto
5     transitions = {
6        Transition.new :inStates => {:mioStato1}, :outStates => {:
           mioStato2},
7        Transition.new :inStates => {:mioStato1, :mioStato2}, :
           outStates => {:mioStato1} do
8           # codice da eseguire durante lo scatto
9        end
10    }
11
12    # DEFINIZIONE DEGLI STATI DEL GADGET
13    class mioStato1 < State
14       def OnEnter
15       end
16
17       def OnLeave
18       end
19    end
20
21    class mioStato2 < State
22       def initialize
23       end
24       #variabili e funzioni di stato
25    end
26
27    class mioStato3 < SoftwareState
28       #funzioni esposte ad altri gadget
29    end
30
31    # DEFINIZIONE DI VARIABILI E METODI APPLICATIVI
32    def initialize
33    end
34
35    def func
36    end
37 end
```

All'interno del file di codice di un gadget troviamo (la struttura dei component è sostanzialmente analoga):

1. Gli stati che il gadget può assumere: ogni stato è definito come una classe interna, figlia della classe che ne rappresenta il tipo.
2. Il vettore delle transazioni, in cui vengono specificati per ogniuna delle componenti:
   a) Gli stati di ingresso
   b) Gli stati di uscita
   c) Il codice da eseguire allo scatto
3. Eventuali altri metodi / variabili globali

### 3.7.2 UI

La cartella UI contiene tutti gli *UI Descriptor* del gadget: file che definiscono l'interfaccia grafica astratta di un particolare stato/modalità del gadget, in modo completamente indipendente dalla reale UI che verrà implementata dal 2lvlOS.

Sono ammessi fino a 2 di questi file per ogni stato, uno per la visualizzazione in *main view* e uno per la visualizzazione in *dock view* (se uno dei due descrittori manca viene negato il passaggio alla relativa modalità), ogniuno dei quali è uno script Ruby che, con un approccio *Shoes like*, utilizza le classi definite in System.UI per rappresentarne l'aspetto grafico.

```
1 #Esempio di UI Descriptor
2 :resultState_main = Panel.new do
3     @grdResults.render
4     Button.new :id=>"filter", :text=>"Nuova Ricerca", :transition
          =>3
5 end
```

Il nome di questi file è standardizzato come:

$$UI/nomeStato.modalità.rui$$

Ovvero *UI/nomeStato.main.rui* oppure *UI/nomeStato.dock.rui*

### 3.7.3 nomeGadget.yaml: Gadget Descriptor

Per poterlo individuare e classificare, ogni gadget deve essere corredato da un *gadget descriptor*: un file (scritto in yaml, un metalinguaggio leggero e veloce da intepretare sia dall'uomo che dalle macchine) che contenga tutta una serie di informazioni quali chi lo ha sviluppato (namespace), il nome del gadget, la versione, il tipo, la data di rilascio, dove cercare le nuove versioni, politiche di distribuzione ecc..

Si tratta, quindi, di memorizzare tutte quelle informazioni che gli utenti e il sistema devono conoscere per capire di che gadget si tratta, di cui non è possibile fornire una lista esaustiva.

### 3.7.4 Setup.rb

Quando un nuovo gadget viene scaricato in un sistema, è necessario preparare l'ambiente per metterlo nelle condizioni di funzionare correttamente. All'interno di questo script (ruby ancora una volta) quindi si troveranno:

- Creazione e primo setting delle chiavi di configurazione
- Creazione e popolazione iniziale dell'eventuale base dati associata al gadget
- Richiesta e download di eventuali altri gadget necessari per il funzionamento

La progettazione più dettagliata del contenuto di questo file è lasciata ad un momento futuro.

# 4 2lvlOS: Kernel

*Il Kernel (del 2lvlOS) è quell'insieme di classi che rende "vivo" il sistema, fungendo da collante tra i diversi componenti del sistema stesso. Compito di questo elemento è permettere ai gadget di funzionare correttamente, vegliando sul loro operato e facendosi mediatore tra i gadget e le risorse del sistema operativo ospite.*

*Il 2lvlOS abbraccerà l'architettura a microkernel, in cui ogni funzione non-kernel sarà implementata e gestita da particolari gadget (chiamati system gadget) strutturalmente e tecnicamente equivalenti ai comuni gadget applicativi e da questo punto di vista giocheranno il ruolo di moduli di sistema. All'interno del sottile strato software che comporrà il microkernel invece, si troveranno unicamente le funzioni indispensabili per il corretto funzionamento dei gadget: l'emulatore delle reti di Petri, la procedura di boot, un meccanismo che permetta la comunicazione tra diversi gadget, il codice dedicato agli aspetti di sicurezza e al supporto multi-user.*

## 4.1 Struttura del Kernel

### 4.1.1 Architetture classiche

La teoria dei sistemi operativi (classici, di primo livello) sostiene che vi sono essenzialmente due architetture di riferimento per i Kernel: a strati e microkernel[1].

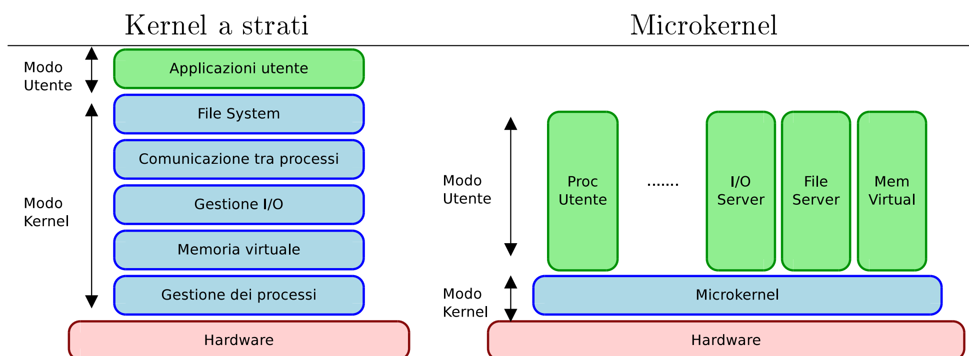


Figura 4.1: Kernel: architetture classiche di riferimento

[1]Per ulteriori informazioni sul kernel e le sue tipiche implementazioni si rimanda al corso di Sistemi Operativi e al materiale in quella sede consigliato.

#### 4.1.1.1 Kernel a strati

Nel primo modello il Kernel è implementato logicamente come una pila di strati software in cui ogni strato utilizza i servizi messi a disposizione di chi gli sta sotto e fornisce nuovi servizi a chi gli sta sopra. Si tratta di un modello che soffre di qualche limitazione legata alle dimensioni e per questo utilizzato spesso solo per piccoli sistemi operativi, in quanto la modifica di uno degli strati in generale può comportare la modifica ti tutti gli strati superiori.

#### 4.1.1.2 Microkernel

Nel modello a microkernel invece, vi è un sottilissimo strato di base che espone solo le funzioni realmente essenziali, chiamato microkernel appunto, e tutte le altre funzioni sono fornite da moduli di sistema che si appoggiano direttamente su questo strato base e grazie a questo comunicano tra loro. Si tratta di un modello molto più flessibile del precedente, anche se spesso leggermente più complicato da implementare, in cui la distinzione tra applicazione e modulo di sistema è molto sottile.

### 4.1.2 Kernel nel 2lvlOS

Anche nel 2lvlOS si può individuare un vero e proprio kernel, costituito dall'emulatore delle reti di Petri e in generale da tutto quell'insieme di strutture che rendono i gadget realmente eseguibili. Entrando più nello specifico, è abbastanza facile ricondursi ad un'architettura di tipo microkernel, sfruttando la potenza messa a disposizione proprio dalle Reti di Petri che consentono di modellare i *system gadget*, che in quest'ottica giocano il ruolo di moduli di sistema, esattamente come se fossero dei comuni gadget applicativi.

Il microkernel del 2lvlOS quindi, è l'unico codice del sistema non strutturato in gadget e contiene:

1. Un emulatore delle reti di Petri
2. La procedura di boot del sistema
3. Il supporto al multi utente
4. I meccanismi di sicurezza
5. Il supporto alla comunicazione cross-gadget locali
6. Il supporto alla comunicazione cross-gadget remoti

Per ogni altra funzione del sistema, dal gestore degli eventi al motore di rendering, ai moduli che sovrintendono l'accesso alle risorse della macchina (file system, rete, database..) vi sarà un gadget che la implementa, che ne ingloba la complessità, e che ne espone delle interfacce controllate.

Questi gadget vengono chiamati appunto *system gadget* e differiscono dai comuni gadget

applicativi per gli elevati permessi di esecuzione che richiedono, necessari per accedere a basso livello alle risorse della macchina[2].

**Framework o System Gadget?**
Ci si potrebbe chiedere perché prevedere l'uso dei *system gadget*, dato che già il framework ha il compito di controllare l'accesso alle risorse disponibili attraverso interfacce al giusto livello di astrazione.
La risposta è da ricercare nella natura dei due elementi, più che nelle scelte progettuali: le istanze delle classi del framework sono completamente *stateless*, indipendenti una all'altra e senza memoria di ciò che è avvenuto nel passato (si pensi a due diversi gadget che istanziano ciascuno la stessa classe del framework), mentre i *system gadget* sono costituiti da un'unica istanza pienamente capace di mantenere nel tempo il proprio stato. Il compito del framework, quindi è quello di fornire interfacce al giusto livello di astrazione, mentre i system gadget possono essere usati per operazioni di sincronizzazione e comportamenti attivi.

## 4.2 MicroKernel

### 4.2.1 Reti di Petri

L'emulatore delle reti di Petri è il primo e più importante componente del microkernel, quello da cui dipende il funzionamento e le prestazioni di tutti i gadget.

Dal punto di vista tecnico la costruzione di questo modulo è assolutamente banale: definiti gli oggetti "stato" e "transizione" di fatto l'emulatore si riduce ad un ciclo infinito che ad ogni iterazione ricerca le transizione abilitate (*t.enabled=true*) e attiva tutti gli stati d'uscita di una qualsiasi di queste (per esempio la prima che trova). Gli eventi applicati agli stati o alle transizioni, la possibilità di sviluppare sottoreti (components) e i meccanismi di load-balancing dovuti alla conpresenza di più gadget si limitano ad essere comportamenti di contorno che non cambiano quella che è l'essenza dell'emulatore.

Unico punto delicato sono le prestazioni: dato che ogni gadget esistente necessariamente si appoggerà a questo modulo, è necessario valutare attentamente se implementarne una soluzione compilata piuttosto che interpretata (in ogni caso in Ruby questo aspetto non costituisce un problema).

### 4.2.2 Bootloading

Vista la relativa semplicità del microkernel e la grande omogeneità delle entità in gioco (in pratica solo gadget), la procedura d'avvio del sistema prevede pochi semplici passi:

[2]In realtà è raro che un System Gadget esegua direttamente l'accesso alle risorse messe a disposizione dal sistema operativo della macchina, in quanto nella situazione tipica sarà il framework ad innalzare il livello di astrazione (lavorando a basso livello), mentre il modulo di sistema si limiterà ad esserne il gestore, l'arbitro degli accessi (ma è un'operazione che avviene già ad alto livello).

1. Caricamento delle definizioni (codice delle classi) dei componenti che andranno in esecuzione
    a) Definizioni del framework
    b) Definizioni del microkernel
2. Caricamento e inizializzazione dei servizi primari
    a) RdP
    b) Sistema Utenti
    c) Sicurezza
    d) Comunicazioni cross-gadget
3. Avvio dei gadget di sistema
4. Avvio desktop-attivo
    a) Recupero desktop attivo tra i desktop installati
    b) Avvio del desktop

Il comportamento del sistema in questa fase di avvio è gestito da uno specifico componente software chiamato BootLoader. Sarà inoltre presente un secondo componente di caricamento, denominato DynamicLoader (parte dell'emulatore delle RdP) il cui scopo sarà caricare / scaricare le parti di sistema o i gadget necessari / superflui al fine di mantenere un uso ottimale delle risorse.

### 4.2.3 Sistema Utenti

Windows e Unix insegnano: se la capacità di gestire più utenti è parte integrante del sistema, è prevista in fase di progettazione e inserita a livello di kernel, allora sarà più facile da gestire, più coerente e più sicura della stessa funzionalità inserita ai livelli superiori.

Con questa idea è stato inserito un sistema di gestione degli utenti direttamente nel microkernel, costituito da:

- Anagrafica utenti:
- Meccanismo di associazione utente - gadget

Quindi un'ipotetico funzionamento può essere il seguente:

- Il microkernel è esterno al sistema utenti, pertanto è virtualmente associato all'utente *root*
- Ogni gadget *deve* essere associato ad un utente per poter essere eseguito. Tale associazione è fatta a livello di gadget, non di thread o component.
- Vi sono due diversi tipi di gadget (il tipo di gadget è un attributo del gadget stesso specificato nel Gadget Descriptor, vedi sezione 3.7.3):

  - *SIGadget*[3], Gadget a singola istanza: in ogni macchina vi potrà essere massimo una istanza di questo gadget, indipendentemente da quanti utenti sono contemporaneamente collegati
  - *MIGadget*[4], Gadget a istanze multiple: in ogni macchina vi potranno essere tante istanze quanti utenti

  Nel primo caso il gadget verrà avviato dall'utente *System*, nel secondo dall'utente a cui quella istanza farà riferimento. Ogni utente, compreso *System* quindi, deve tenere traccia dei propri gadget avviati.

- Nelle comunicazioni cross-gadget, se il gadget ammette istanze multiple è necessario specificare a quale utente appartiene l'istanza cui si vuole fare riferimento (vedi sezione 4.2.5).

#### 4.2.3.1 Processo di boot

Il processo di avvio, dal punto di vista degli utenti, si può esprimere in questo modo:

1. Root: Caricamento delle definizioni
2. Root: Caricamento e inizializzazione dei servizi primari (RdP, Sistema Utenti, Sicurezza, Comunicazioni cross-gadget)
3. Root: Avvio dei gadget di sistema come System
4. System: Attesa autenticazione[5]
5. System: Avvio desktop-attivo come UserX

### 4.2.4 Security

L'alta connettività che caratterizza il sistema giustifica la presenza di un controllo sul codice eseguito cosicché un gadget mal scritto (involontariamente o volontariamente) non possa danneggiare la macchina. Il modello utilizzato è simile a quello adottato in molti sistemi operativi e pervede la suddivisione del codice in 4 classi:

**Kernel code:** rappresenta l'esecuzione di codice del sistema (2lvlOS), questa modalità ha accesso a tutte le risorse (intese anche come informazioni) senza restrizioni.

**Privileged code:** rappresenta l'esecuzione di codice dei gadget che fa uso di risorse protette. Per accedere a questa modalità è necessario la conferma dell'utente o una equivalente conferma da parte del sistema stesso (evitanto un elevato numero di pop-up all'utilizzatore finale).

**Normal code:** normale codice dei gadget che non accede a risorse protette.

[3]SIGadget: Single Instance Gadget
[4]MIGadget: Multiple Instance Gadget
[5]Si assume che l'autenticazione degli utenti (grafica) sia sviluppata come gadget di sistema. Chiaramente le primitive di autenticazione faranno parte del microkernel.

**Unsafe code:** codice di gadget non verificato, di dubbia attendibilità o proveniente da fonti non ufficiali. Massima restrizione nell'uso delle risorse.

Il sistema deve quindi prevedere:

1. Un meccanismo per individuare le risorse protette
2. Un meccanismo di di classificazione del codice in classi sulla base dell'uso delle risorse, prima o contestualmente alla sua esecuzione
3. Un portachiavi (Keyrings) in grado di tenere traccia delle abilitazioni in possesso ai gadget (per l'utente corrente[6])
4. Un costrutto per richiedere l'esecuzione di una o più istruzioni in una specifica modalità

#### 4.2.4.1 Risorse protette

Il compito di questo modulo è controllare l'accesso alle risorse che il sistema operativo della macchina ospite mette a disposizione (2lvlROS è solo una via per accedere a tali risorse, non ne aggiunge di nuove). Tale controllo avviene su due diversi piani:

**A basso livello (LowLock)** imponendo un blocco sulle funzioni proprie del linguaggio di programmazione che ottengono l'accesso alle risorse del sistema

**Ad alto livello (HiLock)** imponendo un blocco all'accesso dei moduli del sistema che gestiscono quella particolare risorsa richiesta

In altre parole il blocco a basso livello ha il compito di fornire un controllo sull'uso di particolari funzionalità intrinseche del linguaggio di programmazione, tipicamente una funzione o un "include" (un buon esempio può essere la funzione File.open che per sua natura implica un accesso al file system), mentre il controllo denominato ad alto livello intercetta le chiamate verso il framework e quei particolari gadget che si occupano della gestione di una particolare risorsa. Per esempio si analizzi il caso di un *system gadget* che implementa un client e-mail:

- L'accesso al gadget sarà controllato da un *HiLock* che verifica se è consentito l'uso della posta elettronica
- Il *system gadget*, a sua volta, sarà costretto a passare attraverso un *LowLock* per ottenere l'accesso alla network

[6]Le autorizzazioni per accedere alle risorse è opportuno specificarle per utente (e non solo per gadget) in quanto sebbene ogni istanza dei gadget abbia un comportamento indipendente dall'utente che l'ha generata, tra le risorse cui accede vi sono le informazioni sull'utente stesso e, si sa, ogniuno di noi percepisce la privacy in modo diverso.

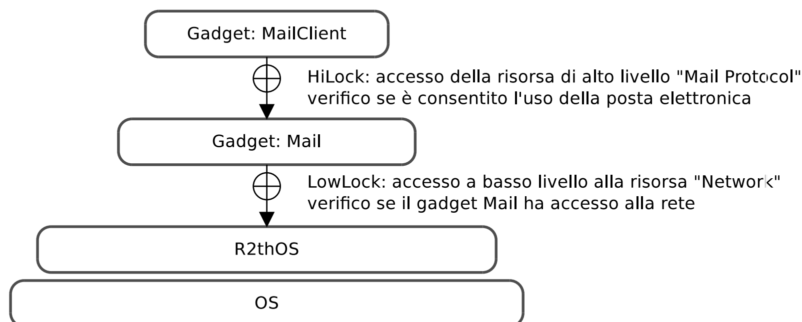


Figura 4.2: Microkernel, Sicurezza: esempio d'uso dei Lock nel caso di un client e-Mail

Già questo semplice esempio evidenzia un comportamento importante e ricorrente: le risorse visibili ad alto livello (quelle gestite dai gadget e dal framework) sono una specializzazione di quelle di basso livello, fornite dal sistema operativo.

La struttura che si crea, quindi, prevede una foresta di risorse (un insieme di alberi) in cui alle radici vi sono le risorse di basso livello, negli altri nodi le risorse di alto livello. Se le risorse di basso livello sono già mappate e definite a livello di sistema, ogni classe del framework e ogni posto esterno dei system gadget dovranno invece essere corredati da una descrizione della risorsa che intendono gestire e delle risorse padre, quelle di cui usufruiscono nell'erogazione dei propri servizi.

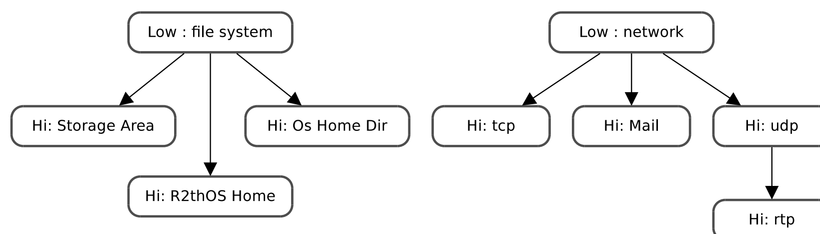


Figura 4.3: Microkernel, Sicurezza: esempio di foresta di risorse

#### 4.2.4.2 Classificazione del codice

Chiaramente, definita una risorsa, non tutte le operazioni che la utilizzano devono essere poste sullo stesso piano (ad esempio nel caso di accesso al file system, la lettura da un'area dati è decisamente meno pericolosa della scrittura in un'area di sistema) ed in questo senso è opportuno assegnare una classificazione al codice: le porzioni di codice che usufruiscono di risorse protette al fine di fornirne un accesso di più alto livello possono essere racchiuse in un blocco, al cui ingresso sono definite le autorizzazioni minime per proseguire (in termini di *Kernel*, *Privileged*, *Normal* e relativamente alla risorsa interessata).

In questo modo chi utilizzerà la risorsa (gestita), si limiterà ad effettuare una normale chiamata al framework o al system gadget (praticamente gli unici moduli che forniscono l'accesso a risorse protette) e solo quando il flusso entrerà nel blocco di sicurezza saranno

avviate (quindi assolutamente trasparentemente a chi effettua l'accesso) tutte le operazioni necessarie per ottenere l'autorizzazione a proseguire.

In questo modo, dato che i blocchi di sicurezza saranno presenti solo nelle classi del framework e nei system gadget, gli unici a fornire ad altri l'accesso a risorse protette, l'intero meccanismo di protezione è completamente trasparente ai gadget applicativi che si limiteranno a invocare il driver per la risorsa interessata (funzione del framework o system gadget che sia) e solo quando il flusso esecutivo entrerà in un blocco automaticamente il kernel provvederà ad effettuare tutte le operazioni necessarie.

Si noti inoltre che la gerarchia di risorse, o meglio del livello di astrazione della funzione di accesso alle risorse, può essere sfruttata per effettuare dei controlli di congruenza e stima della pericolosità del contenuto del blocco:

- Un gadget applicativo può accede ad una risorsa di alto livello (es. Mail), gestita dal framework o da un system gadget, il quale per adempiere alle sue funzioni utilizzerà una risorsa di basso livello antenata della risorsa richiesta (es. Network). E' la situazione più comune e di fatto non rappresenta particolari rischi in quanto l'uso della risorsa è regolato dal driver che si assume essere "ben scritto".
- Se un gadget applicativo chiede l'accesso ad una risorsa di basso livello (direttamente, senza intermediari), può invece essere considerata una operazione decisamente insolita e di dubbia affidabilità.
- Se un gadget applicativo chiede l'accesso ad una risorsa di alto livello ma il driver in gioco (classe del framework o system gadget) usa risorse di basso livello non antenati della risorsa richiesta, allora il driver è tipicamente mal scritto e l'operazione è considerata decisamente pericolosa (è il caso in cui un client Mail richiedesse l'accesso a basso livello sia della network che del file system: chiedere l'accesso alla network è normale per quell'oggetto ma perché anche un accesso a basso livello al disco? Sarebbe stato più comune invece un accesso a basso livello alla network e un accesso ad alto livello al file system).
- Se un driver richiede invece accesso ad altre risorse di alto livello, di fatto non rappresenta particolari minacce in quanto vi sarà un secondo driver a gestirne l'uso.

#### 4.2.4.3 Keyrings

Il salvataggio delle autorizzazioni che un gadget possiede avviene, nella forma più generale, memorizzando:

- L'identificativo del gadget[7] (in realtà la coppia utente-gadget)
- Il massimo livello cui il gadget è abilitato a procedere ($maxAuth$)
- Il minimo livello cui si è negata l'esecuzione ($minNonAuth$)
- La risorsa interessata.

[7] Il 2lvlOS (inteso come kernel) eseguirà sempre con i massimi privilegi per default

Si noti inoltre che:

- Abilitare l'accesso ad una risorsa ad un certo livello dell'albero delle risorse, automaticamente ne abilita l'accesso a tutti i livelli suoi discendenti, in quanto casi particolari dell'autorizzazione già concessa.
- Non ha senso salvare le autorizzazioni per stato, in quanto l'organizzazione in stati è una cosa che avviene internamente al gadget e può essere slegata dagli accessi alle risorse.
- Può essere pericoloso assegnare ad un gadget un certo privilegio a tempo o indipendentemente dalle risorse, in quanto si tradurrebbe in un "consenti tutto" nella maggior parte dei casi.

#### 4.2.4.4 Richiesta di autorizzazione

In linea generale all'ingresso di un blocco critico, vi è un particolare gadget, che per quella particolare risorsa richiede un accesso di livello $x$:

- Se $x < maxAuth$ posso proseguire senza alcuna notifica all'utente in quanto la richiesta era già stata precedentemente autorizzata
- Se $x > minNonAuth$ posso negare l'accesso alla risorsa senza alcuna notifica all'utente in quanto la richiesta era già stata precedentemente negata
- Se $maxAuth < x < minNonAuth$ devo chiedere l'autorizzazione a procedere all'utente in quanto non ho informazioni per prendere una decisione sicura. Questa scelta comporterà un aumento di $maxAuth$ o una diminuzione di $minNonAuth$

Operando in questo modo un aumento del numero di classi in cui il codice è suddiviso non comporta variazioni nella gestione delle autorizzazioni, tuttavia ci si espone al rischio di proporre all'utente un numero di richieste ancora eccessivo, dovuto alla maggiore frequenza di incappare nel terzo caso. Per risolvere almeno parzialmente il problema è utile:

- Assegnare delle priorità statiche ai gadget (per esempio in base alla provenienza)
- Dicchiarare in fase di setup del gadget quali saranno le risorse utilizzate, così da concentrare le richieste in quella fase.

### 4.2.5 Comunicazioni tra gadget

#### 4.2.5.1 Il modello

Come si è detto, la comunicazione tra gadget installati nella stessa macchina avviene concettualmente attraverso dei posti posti esterni nel modello delle RdP. All'interno di questi posti, denominati *softwareState* è possibile definire delle funzioni "esportate", visbili e invocabili da altri gadget locali e remoti (vedi 3.3).

Chi intende passare marche ad un softwareState, ovvero chi intende richiamare una delle funzioni esportate in questi stati, deve semplicemente:

- Identificare il gadget che espone la funzione (che chiameremo gadget destinazione)
- Invocare la funzione
- Attendere il valore di ritorno (concettualmente quanto la funzione termina, indipendente da che transizioni ha abilitato nel gadget destinazione, restituisce sempre la marca)

**Nota:** definite in questo modo tutte le chiamate sono sincrone.

#### 4.2.5.2 Comunicazione locale

Localmente, tra gadget della stessa macchina si possono verificare due condizioni:

- Il gadget destinazione è di tipo SIGadget
- Il gadget destinazione è di tipo MIGadget

Nel primo caso il nome del gadget è sufficiente per identificare univocamente l'istanza in cui invocare la funzione, viceversa nel secondo è necessario specificare anche l'utente cui appartiene l'istanza desiderata (per un approfondimento vedi sezione 4.2.3).

La situazione si complica nel momento in cui l'istanza chiamata (sia che il destinatario sia un SIGadget, che un MIGadget), non è disponibile, che fare?

Anche in questo caso è necessario distinguere due casi:

- Il gadget è di tipo "*autoWakeUp*": quando le comunicazioni richiedono Un'istanza inesistente, questa viene creata dinamicamente dal sistema (naturalmente il gadget deve essere progettato adeguatamente). Può essere una situazione abbastanza tipica per SIGadget di sistema, meno frequente per gli MIGadget.
- Il gadget non è di tipo "*autoWakeUp*": quando le comunicazioni richiedono Un'istanza inesistente, la chiamata restituisce un'eccezione da gestire opportunamente nel gadget chiamante.

In figura 4.4 è mostrata una tipica comunicazione tra gadget locali (il destinatario è un SIGadget). Si noti la sincronia della chiamata che blocca g1 fino all'arrivo del valore di ritorno e il ruolo del 2lvlOS, che interponendosi tra i due gadget rimappa la chiamata sull'istanza corretta.

In figura 4.5 è invece possibile apprezzare la diversità della chiamata, che ora specifica anche l'utente cui l'istanza deve far riferimento, e l'errore generato dalla sua inesistenza.

#### 4.2.5.3 Comunicazione remota

La comunicazione remota non è così diversa dalla comunicazione locale come ci si potrebbe aspettare:

- Nella chiamata è necessario specificare il nome della macchina di destinazione
- Il 2lvlOS deve prevedere un modulo per trasferire tra le macchine chiamate, parametri e valori di ritorno

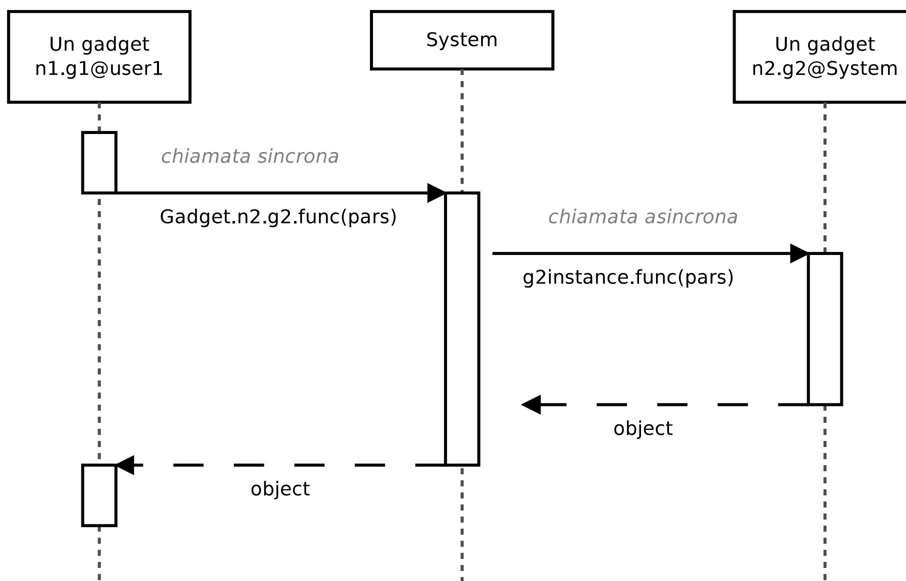


Figura 4.4: Tipica comunicazione tra gadget locali. Caso SIGadget

Ma dal punto di vista dei gadget la struttura della chiamata e i problemi che la affliggono rimangono sostanzialmente invariati: anche in questo caso si deve distinguere tra SIGadget ed MIGadget (è necessario specificare un utente della macchina di destinazione) e l'istanza desiderata può non esistere, sia per i motivi visti nella comunicazione locale sia per problemi di rete.

### 4.2.6 Classi

Le classi che compongono il microkernel sono riposte all'interno del package System.Kernel e sono quelle mostrate in figura 4.8. Per ogniuna di queste classi, in ogni momento vi sarà una e solo una istanza attiva.

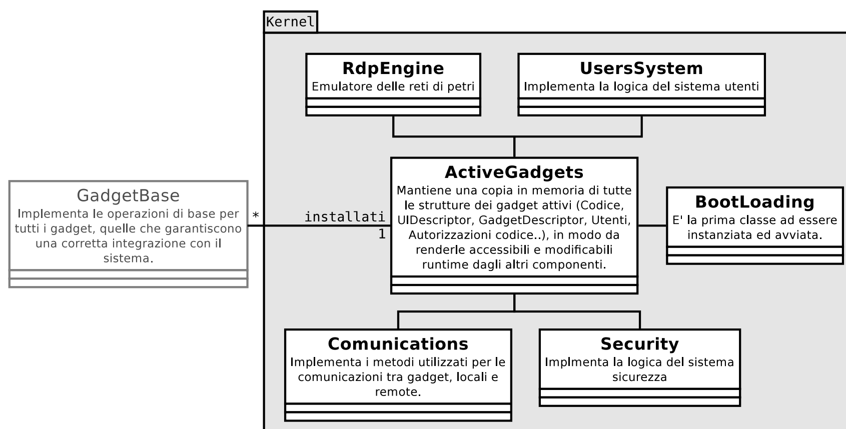


Figura 4.8: Diagramma delle classi - Livello 1 - Kernel

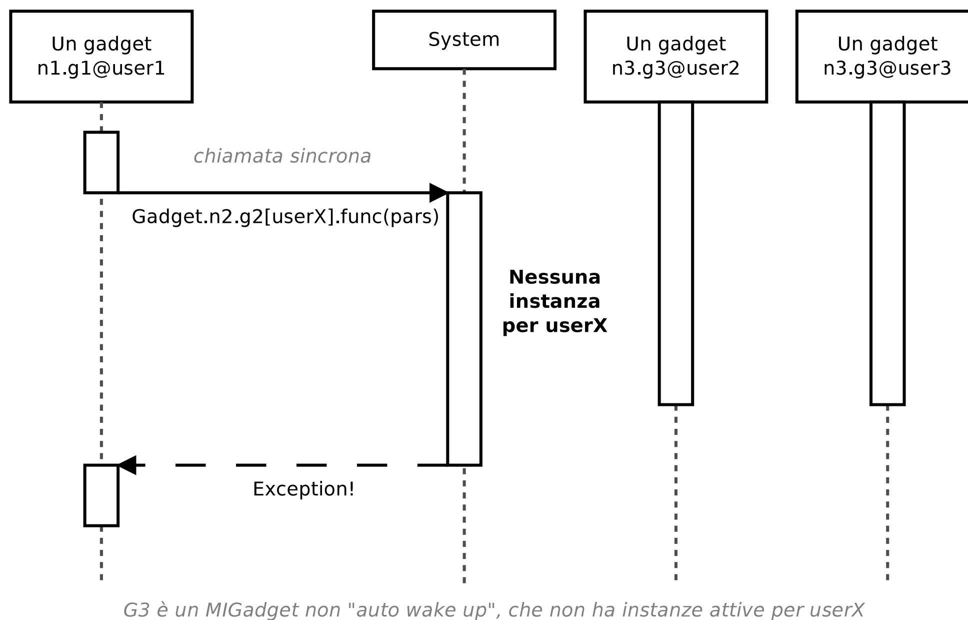


Figura 4.5: Tipica comunicazione tra gadget locali. Caso MIGadget con errore.

## 4.3 Moduli di sistema: I System Gadget

In questa sede si presenteranno solo le funzioni chiave di questi particolari gadget, posticipando in un momento futuro una progettazione più accurata, comunque necessaria per implementare velocemente e correttamente questi componenti.

Tutti i system gadget si caratterizzeranno per essere praticamente privi di UI, per possedere almeno un SoftwareState, per avviarsi automaticamente al boot e per essere *SIGadget* istanziati dall'utente *System*; tuttavia rimangono pur sempre gadget ed in quanto tali non sfrutteranno solo le funzioni messe a disposizione dal microkernel, ma anche l'intera infrastruttura fornita dal framework (che sarà presentato nel capitolo successivo). Può accadere perciò che durante la lettura di questa sezione si incontrino anticipazioni di concetti che saranno trattati più approfonditamente nel prossimo capitolo, se ciò dovesse accadere è sufficiente saltare al capitolo 5 per chiarire ogni dubbio[8].

### 4.3.1 Render Engine

*RenderEngine* rappresenta il motore di rendering del sistema, quel componente che traduce l'interfaccia grafica astratta in pixel colorati sullo schermo.

Vi sono essenzialmente due tecniche per sviluppare un RenderEngine:

1. sfruttare le primitive dell'OS (di primo livello), che permettono di disegnare diret-

[8]La scelta di inserire in questo punto la descrizione dei SystemGadget va vista come un completamento del concetto di kernel: posticiparli dopo la descrizione del framework avrebbe comportato la perdita di quello sguardo di insieme che permette di capire come sono state organizzate tutte le funzioni di base del sistema.

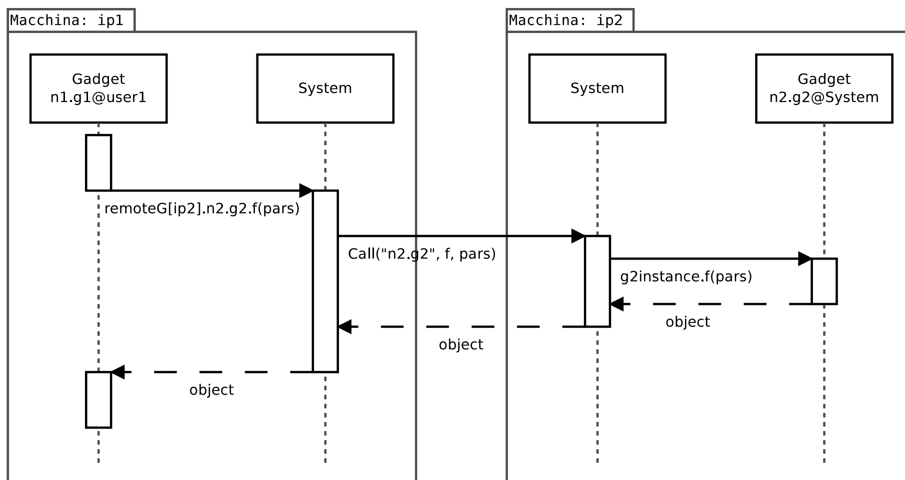


Figura 4.6: Tipica comunicazione tra gadget remoti. Caso SIGadget.

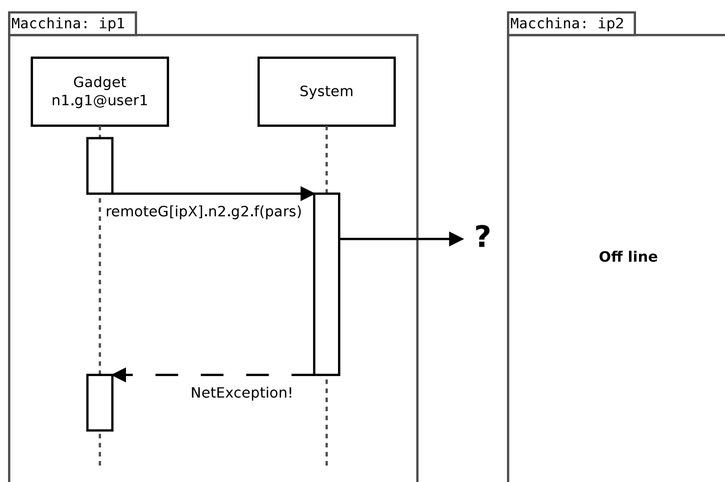


Figura 4.7: Tipica comunicazione tra gadget remoti. Caso SIGadget con errore di rete.

tamente oggetti grafici anche complessi, come bottoni, textbox, caselle combinate ecc..

2. chiedere all'OS (di primo livello) solo la window, disegnandone il contenuto pixel dopo pixel

Nel primo caso si otterrà l'interfaccia nativa dell'OS sottostante, mentre nel secondo una grafica molto più personalizzata, che può presentare anche controlli non previsti dal OS stesso (si pensi alle interfacce che vengono proposte nelle maschere di configurazione di molti videogame 3d).

Nessuna delle due tecniche presenta schiaccianti vantaggi sull'altra: la grafica nativa è più intuitiva (perché l'utente già la conosce) ma decisamente meno accattivante e si riduce a un set di controlli forse superato; mentre la grafica personalizzata indubbiamente richiede più lavoro per essere implementata, (anche se sfruttando degli ottimi motori di rendering per videogame, quale *Irrlicht*, è addirittura possibile averne un'implementazione cross-

platform) e la sua qualità dipende molto dalle capacità di chi lo programma.

Qualsiasi sia la tecnica utilizzata per implementare il *RenderEngine*, però, la sua struttura base non cambia: deve esporre un SoftwareState per ricevere le *UI Descriptor* di cui fare il rendering e deve effettuare una primissima gestione degli eventi che verrà poi affinata dall'*EventHandler*.

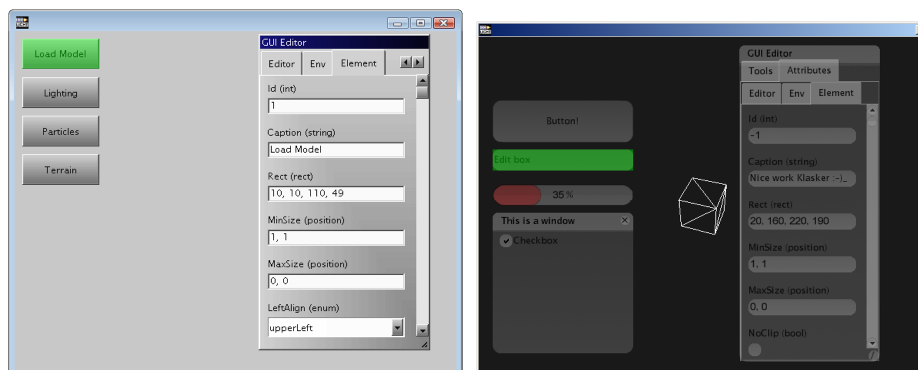


Figura 4.9: Irrlicht: diverse GUI create utilizzando lo stesso motore 3D. Si apprezzi la flessibilità della tecnica.

### 4.3.2 Event Handler

La gestione degli eventi avviene in due diversi step:

1. Il *RenderEngine* cattura gli eventi fisici generati dal sistema operativo sottostante e li standardizza, passandoli mediante un *SoftwareState* all'*EventHandler*

2. L'*EventHandler* effettua la vera gestione dell'evento, individuandone la sorgente e la funzione di call-back associata, che verrà poi processata.

È chiaro che si tratta di due componenti complementari, che dovranno lavorare a strettissimo contatto, ma che non possono essere fusi in un unico componente perché così facendo si perderebbe la possibilità di sostituire il *RenderEngine* (che come si è detto ha un diretto impatto sulla user experience) lasciando invariato l'*EventHandler*, che di fatto ha un comportamento che dipende solo dal modello utilizzato per sviluppare i gadget (quindi immutabile).

Scendendo più in dettaglio il gestore degli eventi deve:

1. Ricevere la notifica dell'evento dal motore di rendering

2. Identificare la funzione di call-back associata all'evento a partire dalla *UI-Descriptor*

3. In base al tipo di call-back function:

   a) Se è una funzione, avviarla

   b) Se è una transizione, abilitarla

### 4.3.3 Adaptive Engine

L'*AdaptiveEngine* è uno dei moduli di sistema potenzialmente più complessi da progettare e implementare, che consente ai gadget applicativi di adattarsi al comportamento di chi li usa.

Concettualmente il funzionamento è semplice: memorizzare le scelte fatte dall'utente in precedenza per prevedere le scelte che farà, attraverso due semplici primitive:

- $set(idScelta, valore)$ per mettere a storia che l'utente ha scelto *valore* nella scelta *idScelta*
- $get(idScelta, maxResult, filter)$ per ottenere i valori che più probabilmente l'utente sceglierà nella scelta *idScelta*.
  - *maxResult* è il numero di risultati desiderati: se è $n$ il metodo deve restituire se possibile le $n$ scelte più probabili; se non è possibile (per mancanza di informazioni per esempio) quante più scelte è in grado di fornire.
  - *filter* opera una pre-selezione sulle opzioni da considerare: in scelte che prevedono molte possibilità, o in caso di scelte libere (domande aperte), l'utente potrebbe aver già specificato delle condizioni per una prima pre-selezione delle opzioni (parte del testo descrittivo dell'opzione per esempio), in tal caso è possibile passare queste informazioni al metodo mediante questo parametro.

La previsione delle scelte più probabili tuttavia, può essere fatta in diversi modi (per semplicità si pensi al caso della scelta di un valore singolo):

- Riportando il valore scelto durante l'ultimo utilizzo
- Riportando il valore che appare con più frequenza nella sequenza delle ultime $N$ scelte (ma quanto grande prendere $N$?)
- Riportando il valore che appare con più frequenza nella sequenza delle ultime $N$ scelte effettuate nello stesso giorno della settimana e nella stessa ora del giorno
- Riportando il valore ottenuto da un modello basato sulle reti neurali (ma come progettarlo? come addestrarlo? come aggiornarlo per tenere traccia delle nuove scelte?)
- ...

Ma una scelta accurata dovrebbe considerare non solo la storia passata e il momento in cui avviene la scelta presente, ma anche l'utente che la esegue, la sua cultura, il comportamento medio degli utenti che gli assomigliano (e come definire i concetti di comportamento medio e somiglianza tra utenti? può nascere un problema di privacy?)

È qui che appare la vera complessità di questo modulo: esclusi gli algoritmi di previsione banali, che di fatto non riescono a prevedere veramente la scelta che farà l'utente, i problemi da risolvere in fase di progettazione di un *AdaptiveEngine* serio, sono tanti e di difficile soluzione (e si lasciano al progettista di *AdaptiveEngine*).

### 4.3.4 Internationalization

La presenza di utenti di diversa nazionalità è una situazione oggi comune a praticamente tutti i prodotti dell'informatica, tuttavia sono ben pochi quelli che si possono definire pienamente internazionali: per rendere un applicativo veramente multilingua infattti, non è sufficiente una mera traduzione delle stringhe statiche (etichette, descrizioni dei comandi ecc..) ma è necessario operare anche una traduzione dei dati dinamici, del formato di numeri, dei simboli di valuta ecc..

In questo senso il termine multilingua è decisamente limitativo e fuorviante, meglio il termine multi-culture, utilizzato in ambiente Microsoft assieme ai concetti di *cultura*, *globalization* e *localization*:

**Definizione.** (di *cultura*) In ambiente .NET con il termine *cultura* si intende quell'insieme di parametri che definiscono le preferenze internazionali di un utente: lingua, variazione linguistica, formato numerico, simbolo di valuta, ecc..

**Definizione.** (di globalization) In ambiente .NET con il termine *globalization* si intende la capacità del sistema di inserire una particolare cultura all'interno di un contesto più ampio: un utente americano ad esempio utilizzerà l'inglese americano come lingua preferenziale, ma qualora questa non fosse disponibile è opportuno che il sistema automaticamente si proponga in inglese generico; differentemente un utente messicano preferirà la variante locale dello spagnolo, seguita dallo spagnolo generico e solo come terza scelta l'inglese.

**Definizione.** (di localization) In ambiente .NET con il termine *localization* si intende la capacità del sistema di individuare automaticamente la cultura preferita di ogni utente, in base alle informazioni di contesto (tipicamente le preferenze del browser).

L'implementazione proposta da .NET può naturalmente essere oggetto di discussione, ma dal punto di vista concettuale gli ingegneri Microsoft hanno colto nel segno: il passaggio dall'idea che l'utente possa cambiare la lingua proposta dal sistema, all'idea che ogni utente sia naturalmente associato ad una determinata cultura ed è compito del sistema individuarla per comportarsi di conseguenza è un punto fondamentale per tutti i software moderni.

Chi ha usato la piattaforma Microsoft sa bene che questo meccanismo funziona egregiamente in presenza di stringhe statiche ma su dati dinamici mostra forti limitazioni, basti pensare che in assenza workaround la quantità 10 in presenza di un utente italiano verrebbe visualizzata come 10 €, mentre in presenza di un utente americano come 10 $.

Questo bizzarro comportamento si manifesta perché i tipi di dato primitivi non supportano nativamente l'internazionalizzazione, ovvero non separano il significato semantico (valore = 10 €) dalla rappresentazione imposta dalle regole di globalization e localization (€=10; $=14).

In 2lvlROS si vuole invece integrare un meccanismo di internazionalizzazione capace di supportare pienamente sia i dati statisti che dinamici, attraverso:

1. strutture dati capaci di separare il valore semantico dalla formattazione dei dati (per approfondimenti vedi sezione sul data flow, 5.2)
2. un gadget che si prenda carico del codice “attivo”

Il system gadget *Internationalization,* rappresenta proprio la parte attiva del sistema di supporto internazionale (il punto 2), i suoi compiti sono:

- mantenere una collezione di culture, standard e personalizzate, dove per ogni cultura sono memorizzate
  - Array ordinato per preferenza delle lingue utilizzabili dell’utente
  - Formato data
  - Formato numerico
  - Valuta utilizzata
- mantenere una collezione di valute, comprendente
  - Identificativo
  - Nome in lingua
  - Simbolo
  - Cambio odierno
- fornire metodi per gestire l’associazione utente-cultura

### 4.3.5 Network Manager

Il network manager è il modulo di sistema che ha il compito di gestire le connessioni di rete, in particolare le comunicazioni cross-gadget remote (per una descrizione dettagliata sul modello utilizzato in 2lvlROS per la gestione della rete, si rimanda alla sezione 5.6).

Se per raggiungere il livello di astrazione desiderato è sufficiente avere delle buone classi di framework che modellano correttamente l’accesso alla network, lo stesso non si può dire per la gestione del flusso: per direzionare sulle porte corrette il traffico in arrivo, per rispondere ai diversi messaggi di controllo (si pensi al semplice “quali SoftwareState sono in ascolto?”) le classi del framework istanziare all’occorrenza non sono più sufficienti, ma è necessario avere un’entità attiva capace di reagire in ogni momento agli stimoli della rete.

Il *NetworkManager* ha proprio questo compito: rimanere in ascolto sulla porta di controllo in attesa di comunicazioni remote per rispondere di conseguenza (ancora una volta per capire nel dettaglio le funzioni che questo componente deve esporre si rimanda alla relativa sezione del capitolo sul framework).

# 5 2lvlOS: framework

*Il framework è una libreria di classi che implementa quello strato software che permette ai gadget di lavorare ad un livello di astrazione più elevato: l'approccio consigliato infatti prevedere di affidare alle classi del framework, completamente stateless, il compito di fornire delle interfacce semplici ma fortemente controllate, lasciando ai gadget la logica applicativa, che di norma necessità il mantenimento di uno "stato". In questo modo la complessità legata all'adeguamento del livello di astrazione è completamente indipendente dalla business logic, nel caso di gadget applicativi, o dal codice attivo di controllo della risorsa in oggetto, nel caso di system gadget.*

*Come per i system gadget è possibile definirne di nuovi espandendo il parco moduli di sistema, così è possibile definire classi di framework personalizzate che ne ampliano le funzionalità. La tecnica per sviluppare e pubblicare queste estensioni è l'oggetto dell'ultima sezione di questo capitolo.*

## 5.1 System.Configuration

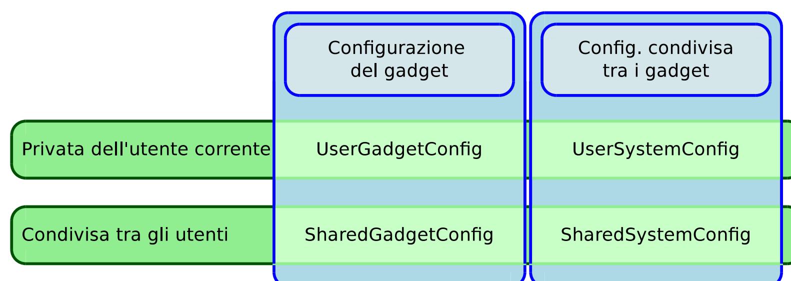


Figura 5.1: Schema dei diversi tipi di chiavi di configurazione

Package che raccoglie le classi che forniscono un'interfaccia controllata per gestire la configurazione dell'ambiente; l'implementazione prevede l'accesso a 4 risorse:

- UserGadgetConfig: chiavi di configurazione del Gadget. Ogni utente ha la sua copia.
- SharedGadgetConfig: chiavi di configurazione del Gadget condivise tra gli utenti del sistema.
- UserSystemConfig: chiavi di configurazione condivise tra tutti i gadget, ma ogni utente ha la sua copia.
- SharedSystemConfig: chiavi di configurazione condivise tra tutti i gadget e tutti gli utenti della macchina.

In particolare si utilizzano i metodi base *get* e *set*, rispettivamente per ottenere e impostare il valore delle chiavi di configurazione, applicati in modalità single e multi user in 2 diverse classi:

- GadgetConfiguration:
  - get / set: per UserGadgetConfig
  - getShared / setShared per SharedGadgetConfig
- SystemConfiguration:
  - get / set: per UserSystemConfig
  - getShared / setShared per SharedSystemConfig

### 5.1.1 Classi

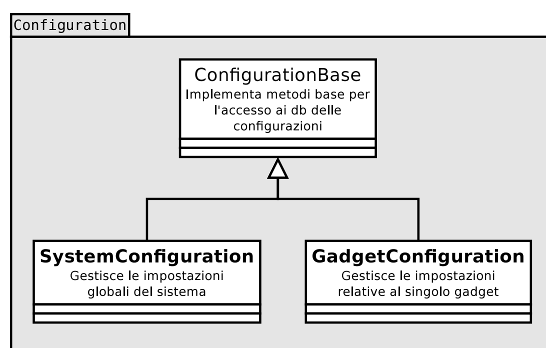


Figura 5.2: Diagramma delle classi - Livello 1 - Configuration

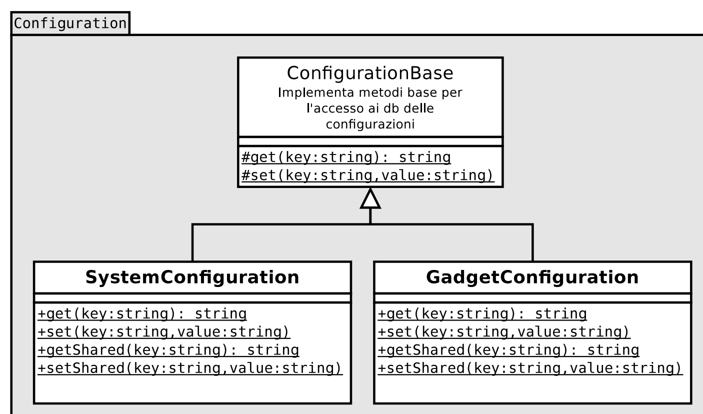


Figura 5.3: Diagramma delle classi - Livello 3 - Configuration

## 5.2 System.Data: Data Flow

### 5.2.1 Introduzione

Il sistema è composto da un'immenso insieme di componenti, molti del quali non solo lavorano con dati strutturati ma li scambiano tra loro e con il mondo esterno. Questo scambio dati, che è un aspetto critico per il successo del sistema stesso, può essere semplice e rapido solo attraverso l'adozione di un unico modello, un'unica struttura dati standardizzata utilizzata da tutte le parti in gioco; in questo modo è possibile:

- rendere i dati separati dalla loro rappresentazione
- facilitare lo scambio dati tra componenti
- facilitare la rimappatura dei dati tra una struttura e un'altra (per renderli conformi al nuovo ambiente ad esempio)

### 5.2.2 Verso il modello

Da quando l'informatica è entrata nella nostra vita quotidiana, siamo stati abituati ad effettuare una sequenza finita, fissa e predeterminata, di scelte per ottenere i risultati di interesse (è il classico wizard). Questo modello, però, è solo una semplificazione di quello che nativamente, quasi biologicamente, è più vicino al pensiero umano: quando nella vita quotidiana dobbiamo filtrare dei dati non ci atteniamo certamente ad una rigida sequenza di scelte, anzi modifichiamo tale sequenza, il numero e il tipo di scelte per poter trovare più velocemente i risultati d'interesse.

Riportare all'interno di un elaboratore questo concetto di scelta, più human-friedly, non è sempre facile perché se l'uomo è in grado di lavorare direttamente con l'informazione, indipendentemente dalla forma cui cui si presenta, un computer può lavorare solo con la sua rappresentazione e cambiarla è molto costoso, ma necessario per modificare dinamicamente i criteri di scelta.

A questo punto, prima di parlare di informatica è bene chiarire il significato di alcuni termini chiave:

**Definizione.** (filtro) Dato un insieme di dati di input $I$ e un insieme di scelte $S$, si definisce filtro il processo di applicazione di una sequenza ordinata di scelte volte ad estrarre dai dati di input $I$ i soli dati di interesse.

**Definizione.** (scelta) Dato un insieme $I$ di dati di input, qualificati da un attributo $A*$, e un valore v di tale attributo (non importa se presente, basta che sia un valore valido per $A*$), si definisce scelta l'operazione di selezione del sottoinsieme $I*$ di I contenente tutti e soli i dati di I che si caratterizzano per $A* = v$

È possibile quindi riflettere il concetto di wizard (sequenza predefinita, statica) organizzando i dati in un albero: partendo dalla radice, dove virtualmente si selezionano tutti i

dati, ad ogni scelta si scende di un livello, restringendo mano a mano la selezione, fino a giungere alle foglie dove si otterranno tutti e soli i dati d'interesse.

In questo modo di vedere le cose, è possibile rappresentare il concetto umano di scelta (in cui la sequenza cambia di volta in volta) attraverso un albero dinamico: ogni volta che si esegue la stessa operazione di filtro, l'albero delle scelte cambierà in funzione dei risultati ottenuti fin d'ora, del momento in cui è eseguito, dell'umore dell'utente che la esegue, ecc...

Per capire meglio quello che avviene è possibile passare ad una rappresentazione alternativa: se si organizzano i dati $I$ in uno spazio n-dimensionale (che chiameremo ipercubo), dove n è il numero di attributi dei dati (su cui ha senso pensare delle scelte) e dove ogni dimensione ha un numero finito di punti (validi) rappresentati da tutti i valori assunti dai dati $I$, in modo che l'elemento generico $i \in I$, che si caratterizza per $A_1 = v_1, A_2 = v_2 ... A_n = v_n$, possa essere collocato alle coordinate $(v_1, v_2 .. v_n)$, allora è possibile pensare all'operazione di scelta come una restrizione di una dimensione del cubo. In questo modo l'intera operazione di filtro è una trasformazione da un ipercubo a un altro ipercubo.

### 5.2.3 Il modello

Il modello che si va a tracciare è quindi così definito:

**Definizione.** (Ipercubo) Dato un insieme di dati $I$ caratterizzati da $A_1, A_2, .. A_n$ attributi rispettivamente nei domini $D_1, D_2, .. D_n$, si definisce ipercubo (associato ad I) uno spazio n-dimensionale discreto dove ogni dimensione $d_i$ è associata ad un attributo $A_i$ e possiede tanti punti validi quanti sono i valori ammessi dal rispettivo dominio $D_i$ dell'attributo cui è associata. Si crea quindi una corrispondenza biunivoca tra la coordinata $x_i^{d*}$ della dimensione $d*$ e il valore $x_i \in D*$

**Definizione.** (punti pieni e vuoti di un ipercubo) Dato un insieme di dati $I$ caratterizzati da $A_1, A_2, .. A_n$ attributi rispettivamente nei domini $D_1, D_2, .. D_n$ e l'ipercubo associato $Q$; un punto dello spazio $X*$ alle coordinate $x_1, x_2, .. x_n$ si dice punto pieno di $Q$ se esistono uno o più dati di $I$ caratterizzati da $A_1 = x_1, A_2 = x_2, .. A_n = x_n$ (si dice anche che tali dati si trovino alla coordinata $X*$). I rimanenti punti si dicono punti vuoti.

#### Note:

- NULL è un valore ammesso da tutti i domini e indica l'assenza di informazione.
- Il numero di dimensioni di un ipercubo è concettualmente pari al numero di dimensioni che hanno (o che si prevede abbiano) almeno un punto pieno in una coordinata diversa da NULL.
- Dire che una dimensione $d$ non è presente nell'ipercubo $Q$ è equivalente a dire che tale dimensione è presente ma ogni punto pieno di $Q$ si trova alla coordinata $d = NULL$

#### 5.2.3.1 Chiarimenti sulla notazione

Si chiarisce inoltre le principali notazioni per manipolare ipercubi:

$Q[d]$ punti pieni di $Q$ proiettati alla sola coordinata $d$

$d$ dimensione generica, identificata da un nome e costituita da un insieme di punti validi.

$d = x$ indica il punto valido $x$ della coordinata $d$

$Q[d = x]$ indica i punti pieni di $Q$ che si trovano alla coordinata $d = x$ (cioè i dati di $Q$ che si caratterizzano per $d = x$)

#### 5.2.3.2 Operazioni di scelta e filtro

Rappresentando i dati come ipercubi è possibile definire le operazioni di filtro e di scelta in questo modo:

**Definizione.** (scelta) Dato un ipercubo $Q$ e una sua dimensione $d*$ in cui è definito il punto valido $x^{d*}$, è possibile definire l'operazione di scelta di $x^{d*}$ in $d*$ come una funzione $Q' = Q[d* = x^{d*}]$ in cui $Q'$ è l'ipercubo $Q$ sezionato alla dimensione $d*$ in corrispondenza del punto $x^{d*}$. In altre parole in $Q'$ sono presenti tutti e soli i punti pieni di $Q$ che si trovano alla coordinata $d* = x^{d*}$

**Definizione.** (filtro) Dato un ipercubo $Q_{in}$ e una sequenza di scelte $S$, si definisce filtro la funzione $Q_{out} = f(Q_{in}, S)$, dove $Q_{out}$ è il risultato ottenuto applicando sequenzialmente (non importa l'ordine) tutte le scelte in $S$ (in modo che l'ipercubo di output di una scelta sia l'ipercubo di input della scelta successiva. L'ipercubo di output dell'ultima scelta è l'output della funzione $f$).

### 5.2.4 Considerazioni sull'approccio: il confronto con Java

Come accennato anche nel capitolo relativo ai gadget, queste strutture saranno il mezzo principale su cui si muoverà la conoscienza nel sistema, tanto che quando due gadget comunicheranno (attraverso un *softwareState*) il chiamante passerà al chiamato ciò che conosce del mondo proprio attraverso un *HiperCube*, contenente in generale molte più informazioni di quelle necessarie alla specifica funzione per espletare i propri compiti (sarà quindi il chiamato a recuperare dal parametro ciò che realmente gli è necessario).

I puristi del mondo Java potrebbero inorridire di fronte ad un simile approccio: in Java la conoscienza è quanto di più pericoloso esista nel software e ogni funzione dovrebbe avere a disposizione solo l'indispensabile per il suo corretto funzionamento.

Si tratta chiaramente di due punti di vista molto diversi, ma entrambi giustificati dal contesto in cui si inseriscono: Java nasce da un mondo C/C++ in cui veramente tutte le informazioni non indispensabili possono rappresentare un pericolo per la sicurezza dell'applicativo, basti pensare che il C è il regno dei *buffer overflow* e del *code injection*; mentre l'approccio qui presentato guarda più al *duck typing*, il principio per cui il programmatore

non è così sbadato da produrre qualcosa di incosistente. Si tratta perciò di un nuovo modo di pensare, basato sulla ricerca di una maggior flessibilità del linguaggio, anche a scapito di qualche controllo di sicurezza, motivato dalla presenza di framework stabili e completi che difficilmente lasciano spazio a bug realmente sfruttabili.

### 5.2.5 Data flow

Per supportare il modello descritto, il framework deve quindi prevedere:

- Una classe *HiperCube* che rappresenti i dati in memoria, cioè che associ un insieme di punti pieni a un *MetaHiperCube*.
- Una classe *MetaHiperCube* che descriva la struttura dimensionale di un *HiperCube* (dimensioni e relativi domini)
- Un set di metodi per modificare ipercubi, in-place e su copia, i più importanti:
  - *select*: operazione di scelta
  - *reduce*: riduzione di una dimensione dell'ipercubo
  - *merge*: fonde due ipercubi in base a una o più dimensioni di riferimento
- Un set di metodi per modificare ogni dimensione, i più importanti:
  - *name*: per cambiare il nome della dimensione
  - *type*: per cambiare il dominio della dimensione

#### 5.2.5.1 Struttura di HiperCube

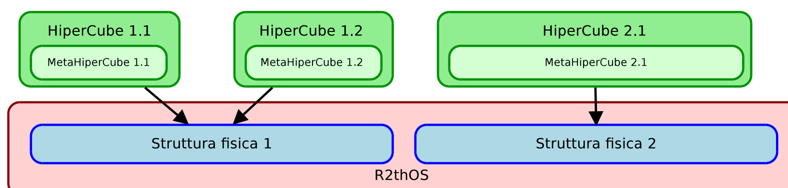


Figura 5.4: HiperCube: Struttura implementativa

*HiperCube* come si è detto rappresenta un ipercubo, in particolare associa punti pieni a un *MetaHiperCube*, non è stato detto tuttavia dove e come effettivamente vengono memorizzati i dati, e come questa classe gestisce l'archivio fisico.

HiperCube non dovrà, infatti, memorizzare nulla, ma si appoggerà a strutture dati di più basso livello a forma tabellare:

- Ogni ipercubo farà riferimento a una struttura dati fisica da cui reperisce i propri punti pieni
- Ogni ipercubo avrà un proprio MetaHiperCube, che definirà le dimensioni dell'ipercubo e attraverso queste quali dati della struttura fisica mostrare.

- Ogni colonna della tabella di basso livello può essere mappata su una diversa dimensione degli ipercubi. I valori che le tuple assumono in corrispondenza di ogni colonna corrispondono alle coordinate nella relativa dimensione dell'ipercubo.

#### 5.2.5.2 Operazioni di base

##### Select

$dstHQ = select(srcHQ, d, x)$

Seleziona tutti e soli i punti pieni di $srcHQ$ che si caratterizzano per $srcHQ[d] = x$ (ovvero che si trovano alla coordinata $x$ di $d$).

##### Reduce

$dstHQ = reduce(srcHQ, d)$

$dstHQ$è la copia di $srcHQ$ (struttura e dati) in cui è stata rimossa la dimensione $d$. Punti pieni coincidenti sono fatti collassare in un unico punto (cioè se vi sono punti che differiscono solo per la coordinata $d$ e che quindi una volta rimossa si troverebbero sovrapposti, questi verranno rimpiazzati da un solo punto pieno).

##### Merge

$dstHQ = merge(srcHQ1, srcHQ2)$

Merge restituisce un ipercubo che contiene i dati di $srcHQ1$ e di $srcHQ2$.

Le dimensioni di $dstHQ$ sono l'unione insiemistica delle dimensioni di $srcHQ1$ e $SRCHQ2$. Si individuano pertanto 3 casi:

- La dimensione generica $dx$ è dimensione sia di $srcHQ1$ che si $srcHQ2$: i dati provenienti da entrambi gli ipercubi sono posti in $dstHQ[dx]$ alla stessa coordinata in cui erano nei rispettivi ipercubi di origine senza spostamenti.

- La dimensione generica $dx$ è dimensione solo di $srcHQ1$: i dati provenienti da srcHQ1 sono posti in $dstHQ[dx]$ alla stessa coordinata in cui erano in $srcHQ1$, mentre i dati provenienti da $srcHQ2$ sono posti alla coordinata $dstHQ[dx] = NULL$

- La dimensione generica $dx$ è dimensione solo di $srcHQ2$: i dati provenienti da $srcHQ2$ sono posti in $dstHQ[dx]$ alla stessa coordinata in cui erano in $srcHQ2$, mentre i dati provenienti da $srcHQ1$ sono posti alla coordinata $dstHQ[dx] = NULL$

### 5.2.6 Classi

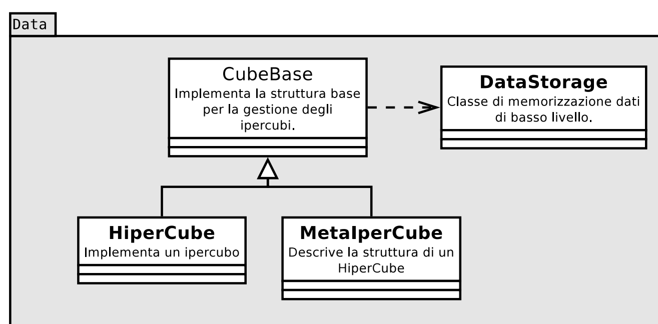


Figura 5.5: Diagramma delle classi - Livello 1 - Data

## 5.3 System.Security

Package che definisce le classi di supporto all'omonima sezione del microkernel (vedi 4.2.4)

### 5.3.1 Classi

Figura 5.6: Diagramma delle classi - Livello 1 - Security

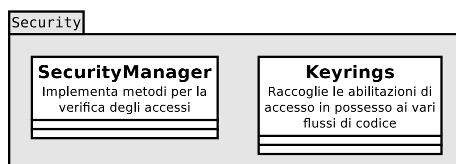


Figura 5.7: Diagramma delle classi - Livello 2 - Security

## 5.4 System.UI: Interfaccia

Come si è detto nella sezione 3.4 i gadget descriveranno solo concettualmente la propria UI, lasciando al motore di rendering il compito di tradurla nella reale interfaccia grafica del sistema. Questo package contiene proprio quell'insieme di controlli che saranno disponibili al progettista di UI per descrivere l'aspetto dei propri gadget.

### 5.4.1 Classi

Le classi che rappresentano l'interfaccia grafica concettuale sono:

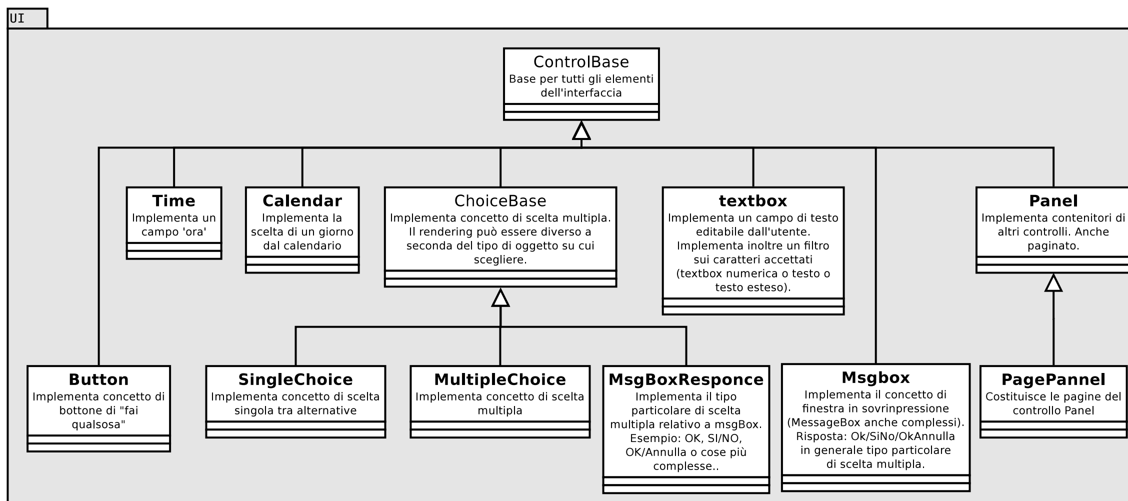


Figura 5.8: Diagramma delle classi - Livello 1 - UI

## 5.5 System.IO

Package che raccoglie le classi che forniscono supporto all'accesso dati, in particolare:

- Accesso a file system
- Accesso a database locali
- Accesso alla rete

### 5.5.1 File System

| | Gadget: none | Gadget: read only | Gadget: read write |
|---|---|---|---|
| User: none | Nessun accesso | Nessun accesso | Nessun accesso |
| User: read only | Nessun accesso | Sola Lettura | Sola Lettura |
| User: read write | Nessun accesso | Sola Lettura | Lettura e Scrittura |

Figura 5.9: Schema su come vengono interpretati i permessi di accesso

Per garantire un accesso coerente in ogni condizione, l'accesso al file system deve essere standardizzato, esattamente come avviene per le altre risorse, appianando le differenze che oggi sono presenti tra i diversi sistemi operativi di primo livello. Si tratta di operare una virtualizzazione composta da:

- una mappa tra path virtuale (dentro il 2lvlOS) e path fisico (OS)
- una definizione del livello di accesso per file / cartelle all'interno del 2lvlOS

Il File System visto dall'interno del 2lvlOS, quindi visto dai gadget, sarà caratterizzato da:

- / un punto di root unico per l'intera macchina (simile alla visione unix)
- **/cartella1/sottoCartella2/** un albero che dalla root si dirama (così come avviene in molti sistemi operativi di primo livello) che termina con i file che ne costituiscono le foglie
- **file.ext** ogni file è caratterizzato da un nome (eventualmente un'estensione)
- Ogni file e ogni cartella possiedono degli attributi che ne descrivono il livello di accesso:
  - *User*: permessi per l'utente proprietario
  - *otherUsers*: permessi per tutti gli utenti non proprietari
  - *Gadget*: permessi per il gadget proprietario
  - *otherGadgets*: permessi per tutti i gadget non proprietari

  Ove ogniuno di questi attributi può essere:
  - *None*: nessun accesso
  - *ReadOnly*: sola lettura
  - *ReadWrite*: lettura / scrittura[1]
- La combinazione tra i permessi Utente e Gadget va interpretata come mostrato in figura 5.9, dando priorità al vincolo più stringente.
- Il proprietario di un file / cartella è la coppia *utente* - *gadget* che lo ha creato. Quell'utente, attraverso quel gadget, è l'unico in grado di cambiare gli attributi del file o della cartella, compreso il proprietario dello stesso oggetto: attraverso un'opportuno comando egli può infatti definire il nuovo utente proprietario, perdendo immediatamente il possesso del file (in altre parole non può tornare indietro ridefinendosi proprietario).
  Per permettere una maggiore flessibilità però, è possibile *non* specificare una delle due componenti della coppia *utente-gadget*:
  - Se si specifica solo l'utente, si rilassa il vincolo sul gadget in modo che l'utente possa accedere come proprietario del file attraverso qualunque gadget.
  - Se si specifica solo il gadget, indipendentemente dall'utente, il gadget avrà pieno controllo su quel file (inteso come modifica degli attributi).

  In entrambi i casi la componente mancante perde di senso nella definizione dei permessi di accesso che saranno impostati per definizione a ReadWrite.
- Per garantire il corretto funzionamento del sistema, gli utenti *Root* e *System* (che rispettivamente rappresentano il kernel e i gadget di sistema) hanno pieno accesso a tutti i file / cartelle, come se ne fossero co-proprietari.
- Vi sono dei percorsi speciali, preconfigurati dal 2lvlOS:

[1] Applicata alle cartelle l'opzione ReadWrite permette di aggiungere e cancellare file all'interno

- /**Gadgets/GadgetX** Spazio riservato a GadgetX, di default con i seguenti permessi:
  * *User*: ReadWrite
  * *otherUsers*: ReadWrite
  * *Gadget*: ReadWrite
  * *otherGadgets*: None
- /**Users/UserX** Spazio riservato all'utente, di default con i seguenti permessi:
  * *User*: ReadWrite
  * *otherUsers*: None
  * *Gadget*: ReadWrite
  * *otherGadgets*: ReadWrite

### 5.5.2 Database

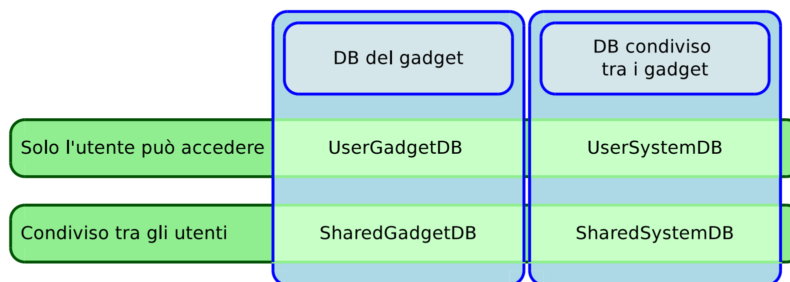


Figura 5.10: Schema dei diversi database supportati e loro ruoli

L'accesso ai database locali avviene in funzione della visibilità dei dati con cui si vuole lavorare (vedi figura 5.10). Sono previsti 4 tipi di risorse diverse:

- UserGadgetDB: database del Gadget. Ogni utente ha la sua copia.
- SharedGadgetDB: database del Gadget condiviso tra gli utenti del sistema.
- UserSystemDB: database condiviso tra tutti i gadget, ma ogni utente ha la sua copia.
- SharedSystemDB: database condiviso tra tutti i gadget e tutti gli utenti della macchina.

In realtà non si tratta di suddivisione fisica: che vi siano esattamente 1 database per ogni utente, 1 database per ogni gadget, 1 database per ogni coppia gadget-utente ecc.. o un unico database poco importa, in quanto l'unica cosa che i gadget vedranno sono le primitive di accesso, che dovranno avere un comportamento coerente con questa definizione.

In accordo con quanto definito alla sezione 5.2, che prevede un unico modello per tutti dati che fluiscono nel sistema, l'accesso ai database deve essere virtualizzato in modo da fornire ai gadget una visione per mezzo di ipercubi, mappando ogni tabella in un diverso *HiperCube*.

Esempio:

- **UserGadgetDB.Users** è un oggetto *HiperCube* che mappa la tabella *Users* del gadget corrente, riportando ogni colonna in una dimensione diversa.

### 5.5.3 Classi

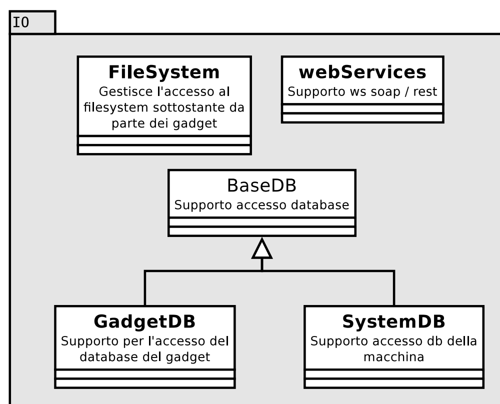


Figura 5.11: Diagramma delle classi - Livello 1 - IO

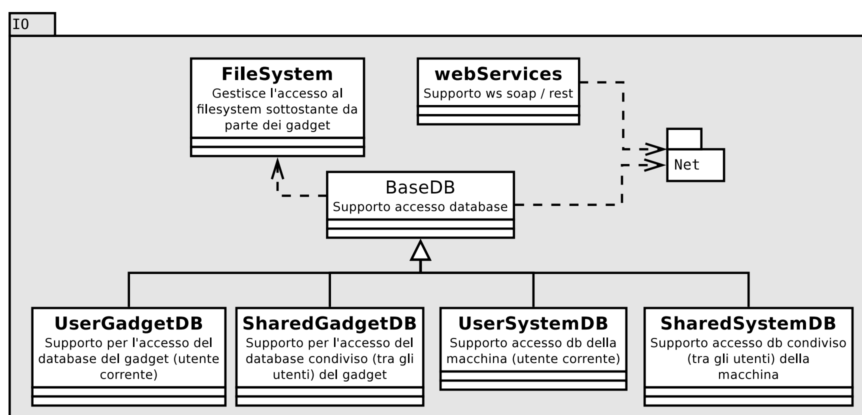


Figura 5.12: Diagramma delle classi - Livello 2 - IO

## 5.6 System.IO.Net: Network e sicurezza

Questo package fornisce un accesso uniforme alla rete ed implementa i meccanismi necessari per le comunicazioni tra gadget remoti (vedi sezione 4.2.5). Le sue classi modellano l'accesso alla rete attraverso un'infrastruttura a livelli:

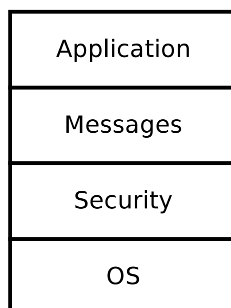


Figura 5.13: Network layers

**Application layer:** costituito dai metodi ad alto livello invocati dai gadget che intendono comunicare tra loro.

**Messages layer:** costruisce e interpreta i messaggi che vengono trasmessi sul canale. I messaggi rispetteranno lo standard web service rest.

**Security layer:** implementa un canale sicuro per la comunicazione.

**OS layer:** rappresenta il collegamento fornito dal sistema operativo mediante l'IP della macchina ospite.

Prima di addentrarsi in una descrizione più dettagliata di questi layer è necessaria una premessa: sebbene il package sia progettato anche per fornire un accesso base alla *inter-network*, si volgerà sempre lo sguardo alle comunicazioni tra gadget remoti, non solo perché è la situazione più complessa ma anche e soprattutto perché sarà poi possibile sfruttare i layer inferiori (in particolare OS Layer) per un accesso alla rete più *general porpuse*.

### 5.6.1 OS layer

Lo scopo di questo layer è creare una solida base comune, indipendente dal sistema operativo di primo livello, su cui costruire la pila di protocolli mostrata in figura 5.13, fornendo una connessione TCP tra mittente e destinazione.

A questo proposito si ricorda che TCP è un protocollo di trasporto (livello 4 del modello ISO/OSI) che fornisce una connessione affidabile punto-punto, in grado di trasferire un flusso di byte in modalità full-duplex.

OS Layer deve quindi mettere a disposizione primitive per:

- Instaurare la connessione specificando il destinatario
- Inserire un flusso di byte nel buffer di invio
- Ricevere un flusso di byte e passarlo al livello successivo
- Regolare il flusso

In modo ovviamente indipendente dal sistema operativo sottostante.

Sebbene saranno usati più raramente OS Layer implementerà ed esporrà ai livelli superiori anche il supporto per i protocolli:

**UDP** per lo scambio di messaggi senza particolai garanzie

**RTP** un protocollo basato su UDP senza garanzie di consegna e di ritardi, ideale per flussi multimediali in tempo reale

### 5.6.2 Security layer

Il Security layer protegge sia le comunicazioni tra gadget remoti, sia le comunicazioni gadget - server in modo da:

- Identificare i soggetti che stanno comunicando
- Evitare che soggetti interni od esterni al sistema possano accedere alle informazioni in transito
- Evitare che soggetti interni al sistema possano effettuare richiese illecite

Verranno quindi utilizzati meccanismi di cifratura e firma digitale (la cui specifica progettazione è ora tralasciata) per fornisce ai livelli superiori un canale di comunicazione sicuro sia che si tratti di flussi TCP che UDP.

### 5.6.3 Messages

Lo scopo di questo layer è definire la lingua comune da utilizzare nella comunicazione: il protocollo in cui vengono trasferiti comandi e dati.

Sulla rete possono viaggiare tre tipologie diverse di comunicazioni:

1. Messaggi di controllo generati dal *NetworkManager* (vedi sezione 4.3.5) per la regolazione del flusso
2. Flussi dati TCP, generati dalla comunicazione tra gadget remoti (quando uno dei due inserisce marche in un SoftwareState del remoto invocandone una funzione)

La comunicazione tra gadget remoti avviene passando un *HiperCube* come parametri di una funzione remota e recuperando un oggetto dello stesso tipo come valore di ritorno (vedi sezioni 3.3 e 4.2.5); in questa situazione è necessario:

1. prevedere un meccanismo di identificazione delle funzioni remotizzate
2. prevedere un meccanismo di serializzazione e deserializzazione di *HiperCube*, unici parametri delle funzioni remotizzabili
3. lasciare la possibilità che più coppie di gadget stiano comunicando tra gli stessi due host

#### 5.6.3.1 Identificazione delle chiamate

Ogni funzione remotizzata può essere univocamente identificata da:

- Gadget
- Nome del SoftwareState

- Nome della funzione

#### 5.6.3.2 Serializzazione di HiperCube

Ogni HiperCube si compone essenzialmente di due parti (vedi sezione 5.2):

- Il MetaHiperCube, che ne descrive la struttura
- La struttura fisica, contenitore dei dati, cui l'ipercubo è associato

Il trasferimento di un ipercubo quindi si realizza trasferendo sulla macchina remota la serializzazione del meta-ipercubo e dei soli dati della struttura fisica mappati nei punti pieni dell'HiperCube da trasferire.

#### 5.6.3.3 Comunicazioni multiple

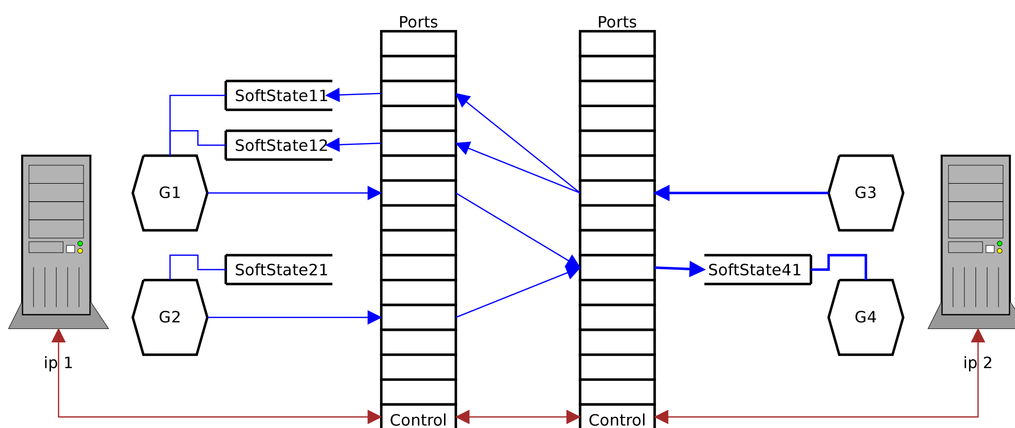


Figura 5.14: System.IO.Net: Connessioni tra macchine remote

Sia per motivi di robustezza che di ottimizzazione del flusso, è opportuno assegnare ogni comunicazione a connessioni di trasporto indipendenti:

- In ogni macchina viene riservata una porta standard (ed immutabile) alla ricezione dei messaggi di controllo
- Ad ogni *SoftwareState* (di ogni gadget attivo) viene riservata una diversa porta per la ricezione dei dati, qualsiasi sia il tipo di traffico in arrivo
- Ad ogni gadget attivo viene riservata una diversa porta per l'invio dei dati.

**Note:**

- Se due macchine stanno comunicando vi sarà una sola connessione TCP tra le porte di controllo di entrambe le macchine
- Se un gadget contatta contemporaneamente $n$ *SoftwareState* diversi, instaurerà $n$ diverse connessioni TCP dalla sua porta di uscita (sempre la stessa) alle rispettive porte di ingresso dei *SoftwareState* con cui interagisce.

### 5.6.4 Application

Questo layer implementa le tipiche funzioni di STUB: fornisce l'interfaccia per i gadget che intendono iniziare la comunicazione con il partner remoto, ed effettua le chiamate ai *softwareState* locali quando giunge una *call* dalla rete.

### 5.6.5 Classi

Figura 5.15: Diagramma delle classi - Livello 2 - IO: Network Layers

## 5.7 System.Gadget

Package che raccoglie le classi che forniscono la base per la creazione di nuovi gadget.

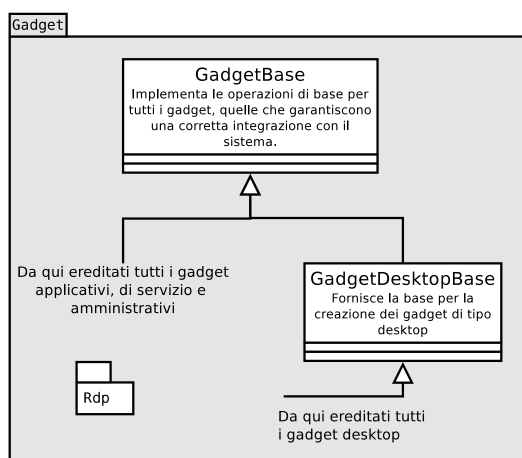


Figura 5.16: Diagramma delle classi - Livello 1 - Gadget

### 5.7.1 System.Gadget.RdP

Package che implementa le classi che rappresentano i componenti delle Reti di Petri con cui i gadget sono modellati.

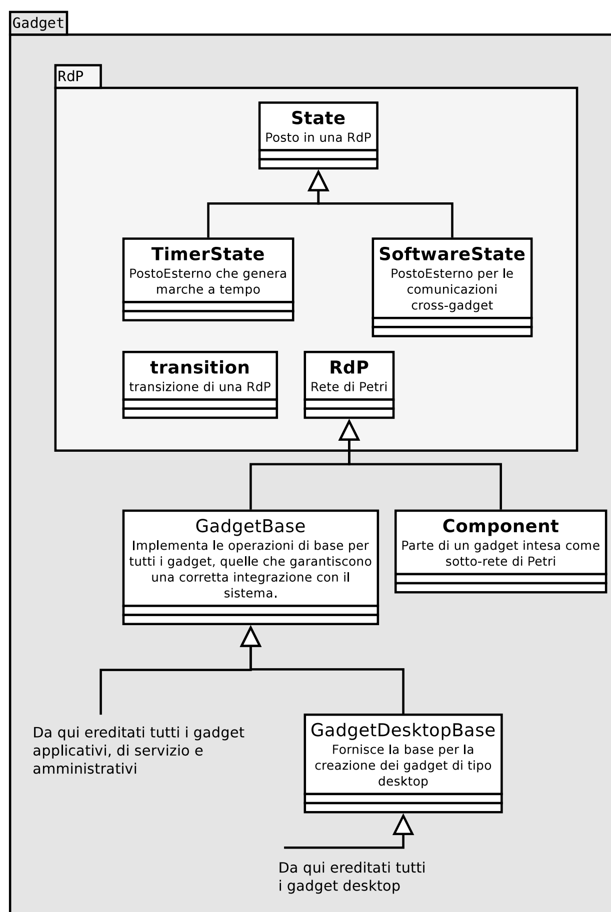


Figura 5.17: Diagramma delle classi - Livello 2 - Gadget

### 5.7.2 Estensioni al framework

E' prevista la possibilità di estendere il framework fornito con il sistema attraverso la definizione di classi personalizzate, individuate analogamente a quanto avviene per i gadget da:

- Un namespace che ne identifica l'autore, nella forma puntata inversa (es: it.unipd.dei)
- Il nome della classe, che identifica l'oggetto
- Un numero di versione nella forma Major Version, Minor Version, Build: $xx.xx.xxxx$

**Nota:** non sono ammessi namespace che iniziano per *System.* in quanto è un nome riservato.

La pubblicazione delle estensioni del framework avviene attraverso un sistema del tutto analogo a quello utilizzato nella pubblicazione dei gadget.

#### 5.7.2.1 Implementazione

La struttura di una estensione del framework sarà quindi la seguente:

**nomeClasse.rb** contenente il codice della classe "nomeClasse"

**nomeClasse.yaml** contenente le informazioni ausiliarie necessarie:

- Namespace
- Nome
- Descrizione
- Versione
- ...

# 6 Basi dati interne

*Come già accennato nel capitolo relativo al Framework, ogni coppia utente-gadget avrà a disposizione 4 diversi tipi di database: UserGadgetDB, SharedGadgetDB, UserSystemDB e SharedSystemDB (per una descrizione più approfondita si rimanda alla sezione 5.5.2 dello stesso capitolo); tuttavia queste basi dati non sono sufficienti a mantenere tutte le esigenze informative del sistema in quanto anche il microkernel, che non è organizzato in gadget, ha necessità di memorizzare dati.*

*Questa nuova base dati, in tutto e per tutto parte integrante del microkernel stesso, è da sviluppare con particolare cura non solo perché andrà a contenere informazioni particolarmente delicate, come le autorizzazioni concesse ai diversi gadget o il loro codice eseguibile, ma anche perché una mal progettazione potrebbe portare a forti limitazioni nell'uso complessivo di tutto il sistema.*

## 6.1 Basi di dati

Sebbene le basi dati siano un punto centrale del sistema, grazie alla particolare architettura scelta, la loro progettazione non è così complessa come ci si potrebbe aspettare. Una prima importante suddivisione può essere fatta tra:

- I dati necessari per il corretto funzionamento del microkernel
- I dati necessari per il corretto funzionamento dei gadget

Le basi dati necessarie ai gadget, o meglio a disposizione dei gadget, sono già state definite nella sezione dedicata alle classi System.IO del Framework (vedi 5.5) e sono:

*UserGadgetDB:* una base dati per ogni coppia utente-gadget

*SharedGadgetDB:* una base dati per ogni gadget

*UserSystemDB:* una base dati per ogni utente

*SharedSystemDB:* una base dati globale e condivisa tra tutti gli utenti e tutti i gadget per il sistema

L'implementazione suggerita rimane la creazione di un diverso database fisico per ogni database virtuale richiesto (un database per ogni coppia utente-gadget, un database per ogni gadget ecc..), nonostante vengano creati un numero decisamente elevato di basi dati molte delle quali praticamente vuote. Le ragioni di questa scelta sono da ricercare nella maggior flessibilità che questo approccio garantisce: non sono necessari particolari meccanismi di

virtualizzazione, non possono sorgere conflitti tra nomi di tabelle appartenenti a basi dati logicamente diverse, non è necessario imporre particolari restrizioni nella struttura dei dati (ad esempio è lecito che per lo stesso gadget, ma per due diversi utenti, le strutture dei rispettivi UserGadgetDB siano diverse) e un qualunque errore rimarrebbe confinato all'interno della base dati che lo ha prodotto.

Per quanto riguarda il microkernel invece, i compiti (e i dati) di questo componente sono ridotti al minimo. In figura 6.1 è presente un primo schema concettuale della relativa base dati; si noti:

- La presenza centrale dell'anagrafica dei gadget, corredati da tutte le informazioni ausiliarie necessarie, e dell'anagrafica degli utenti.
- La tabella *Authorizations* contiene il massimo livello concesso e il minimo livello negato per ogni tripla *{gadget, utente, risorsa}*.
- Le 4 tabelle contenenti le chiavi di configurazione di tutto il sistema

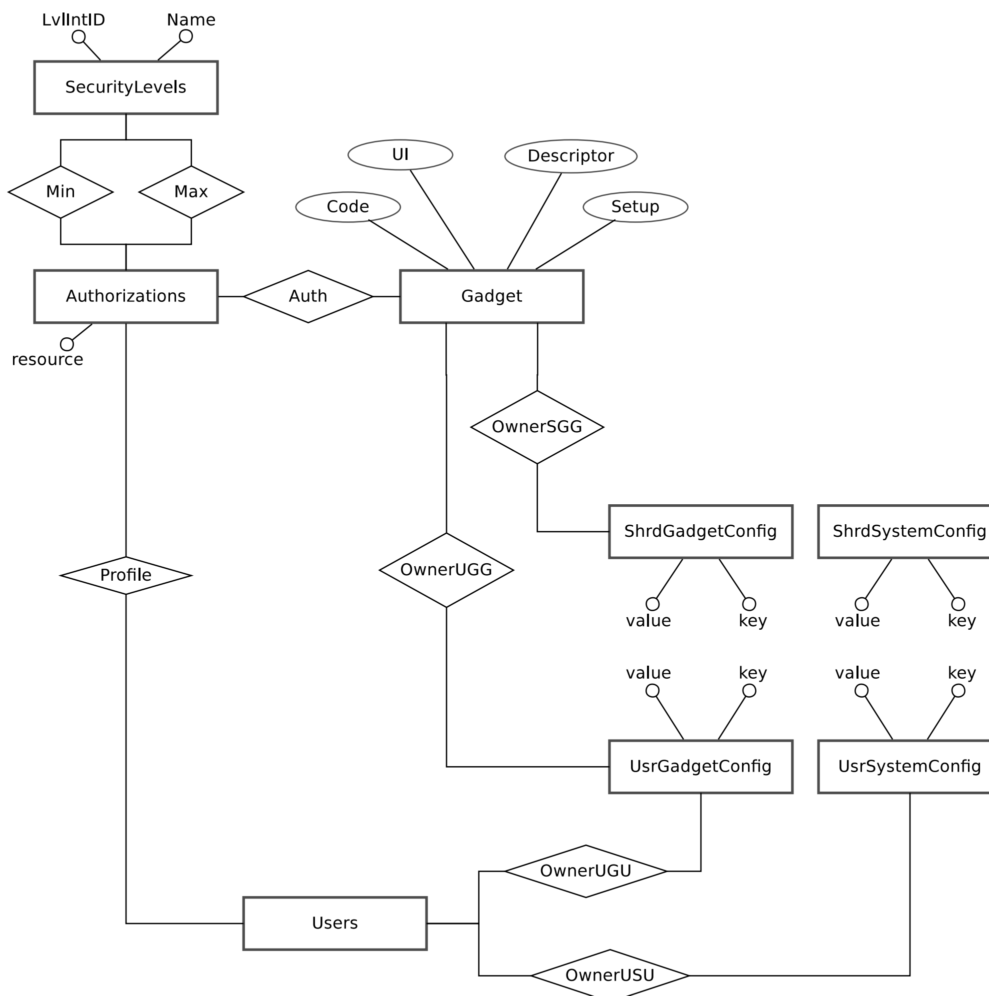


Figura 6.1: Basi dati interne: Modello ER - livello 0

# Parte II

# Progettazione dei gadget

# 7 Farmacie

*Uno degli usi tipici dei gadget applicativi è il recupero d'informazioni: saranno moltissimi infatti quelli sviluppati per essere intermediari tra le diverse sorgenti dati e l'utente finale. Come primo esempio di gadget si è quindi scelto di presentare proprio un applicativo che fornisca all'utilizzatore informazioni riguardanti le farmacie presenti in un comune italiano di suo interesse.*

*Si tratta di un gadget molto semplice, sia concettualmente che strutturalmente, ma che permette di iniziare a familiarizzare con l'architettura, le tecniche e le modalità di sviluppo introdotte dal 2lvlROS: la rete di Petri espressa come array di transizioni, il bind con sorgente dati, l'interfaccia grafica astratta ecc..*

*Interessate ed esplicativo, inoltre è il confronto tra il codice presentato e gli screenshot di come realmente il gadget potrebbe apparire.*

## 7.1 Obiettivi

Progettazione di un gadget che fornisca all'utente le informazioni relative alle farmacie italiane, in particolare nome e indirizzo delle farmacie presenti in un determinato comune italiano.

- Il gadget non deve fornire a terzi alcun servizio
- Il gadget dovrà provvedere meccanismi per evitare all'utente di dover specificare ad ogni esecuzione il comune di suo interesse.

## 7.2 RdP

Il gadget, molto semplice, si può organizzare in soli due stati:

**FilterState:** fornisce all'utente gli strumenti che gli consentono di selezionare il comune italiano

**ResultState:** visualizza le informazioni sulle farmacie presenti nel comune selezionato.

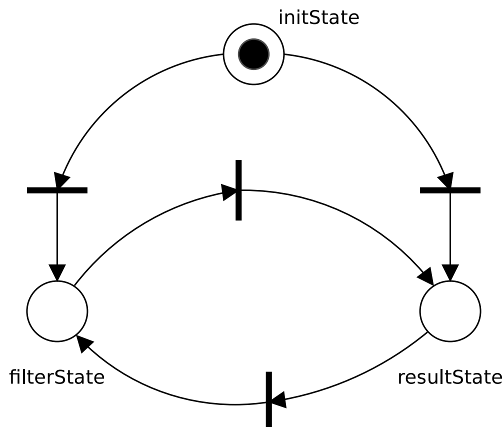


Figura 7.1: Gadget di esempio Farmacie: Rete di Petri

## 7.3 Sorgenti dati

Il gadget non necessita di basi dati locali, tuttavia è necessario una sorgente dati remota per il recupero delle informazioni aggiornate sulle farmacie. Si tratterà quindi di un web service capace di fornire i dati secondo il seguente formato:

```
1  <farmacie>
2     <regione id='1' nome='veneto'>
3        <provincia id='10' name='padova' />
4        <provincia id='11' name='venezia' >
5           <comune id='110' name='mirano'>
6              <farmacia nome='' indirizzo='' />
7              <farmacia nome='' indirizzo='' />
8              <farmacia nome='' indirizzo='' />
9              <farmacia nome='' indirizzo='' />
10          </comune>
11       </provincia>
12    </regione>
13    <regione id='2' nome='trentino' />
14    <regione id='3' nome='piemonte' />
15 </farmacie>
```

## 7.4 Implementazione

### 7.4.1 Code_Farmacia.rb

```
1  class Farmacia < GadgetBase
2
```

```
3     #dicchiaro controlli grafici per poterli associare ai dati
4     @selRegione = SingleChoice.new
5     @selProvincia = SingleChoice.new
6     @selComune = SingleChoice.new
7     @grdResults = GridResults.new
8
9     #specifico databind per i controlli grafici
10    #in questo caso, si tratta di controlli a scelta singola
          caratterizzari per lavorare su una coppia di valori [chiave
          , valore], ove valore è l'oggetto mostrato, mentre chiave è
           l'oggetto che funge da identificatore.
11    # xml => sorgente dati
12    # itemKey => quale parte dell'albero xml costituisce la chiave
           che identifica l'elemento
13    # itemValue => quale parte dell'albero xml costituisce il
          valore, l'oggetto mostrato
14
15    @selRegione.dataBind do |xml, itemKey, itemValue|
16        xml = webServices.get(gadgetConfiguration.get("webService")
              )
17        itemKey = "/farmacie/regione[@id]"
18        itemValue = "/farmacie/regione[@nome]"
19    end
20
21    @selProvincia.dataBind do |xml, itemKey, itemValue|
22        xml = webServices.get(gadgetConfiguration.get("webService")
              , :regione => @selRegione.selectedItemKey)
23        itemKey = "/farmacie/regione[@id="+@selRegione.
              selectedItemKey+"]/provincia[@id]"
24        itemValue = "/farmacie/regione[@id="+@selRegione.
              selectedItemKey+"]/provincia[@nome]"
25    end
26
27    @selComune.dataBind do |xml, itemKey, itemValue|
28        xml = webServices.get(gadgetConfiguration.get("webService")
              , :regione => @selRegione.selectedItemKey, :province =>
              @selProvincia.selectedItemKey)
29        itemKey = "/farmacie/regione[@id="+@selRegione.
              selectedItemKey+"]/provincia[@id="+@selProvincia.
              selectedItemKey+"]/comune[@id]"
```

```
30        itemValue = "/farmacie/regione[@id="+@selRegione.
              selectedItemKey+"]/provincia[@id="+@selProvincia.
              selectedItemKey+"]/comune[@name]"
31    end
32
33    @grdResults.dataBind do |xml|
34        xml = webServices.get(gadgetConfiguration.get("webService")
              ,
35        :regione => @selRegione.selectedItemKey,
36        :province => @selProvincia.selectedItemKey
37        :comune => @selComune.selectedItemKey)
38    end
39
40    #elenco delle transizioni, da valutare se spostarle in nuovo
          file separato
41    transitions = {
42        Transition.new :inStates => {:initState}, :outStates => {:
              filterState},                                  # 0
43        Transition.new :inStates => {:initState}, :outStates => {:
              resultState},                                  # 1
44        Transition.new :inStates => {:filterState}, :outStates =>
              {:resultState} do                              # 2
45            adaptiveEngine.set :choiceId => 1, :choise => {
                  @selRegione, @selProvincia, @selComune}
46        end,
47        Transition.new :inStates => {:resultState}, :outStates =>
              {:filterState}                                   # 3
48    }
49
50
51    #definizione degli stati
52    class InitState < State
53        def onEnter
54            #chiedo all'adapter engine da 0 a 1 proposta
55            proposta = adaptiveEngine.get :choiseId => 1, :
                  maxResults => 1
56            #se effettivamente mi ha restituito una proposta vado
                  direttamente a visualizzare i risultati
57            if proposta == null then
58                transitions[0].enable
59            else
```

```
60                transitions[1].enable
61            end
62        end
63    end
64
65    class FilterState < State
66    end
67
68    class ResultState < State
69    end
70 end
```

### 7.4.2 UI\filterState.main.rui

```
1 :filterState_main = Panel.new do
2     @selRegione.render
3     @selProvincia.render
4     @selComune.render
5     Button.new :id =>"confirm", :text=>"OK", :transition=>2
6 end
```

### 7.4.3 UI\resultState.main.rui

```
1 :resultState_main = Panel.new do
2     @grdResults.render
3     Button.new :id=>"filter", :text=>"Nuova␣Ricerca", :transition
          =>3
4 end
```

## 7.5 ScreenShots

L'aspetto finale ovviamente dipende dall'implementazione del kernel e del gadget che realizza il motore di rendering, ma indicativamente l'utente potrebbe vedere qualcosa di simile a questo:

Figura 7.2: Gadget di esempio Farmacie: screenshot del posto “filterState”

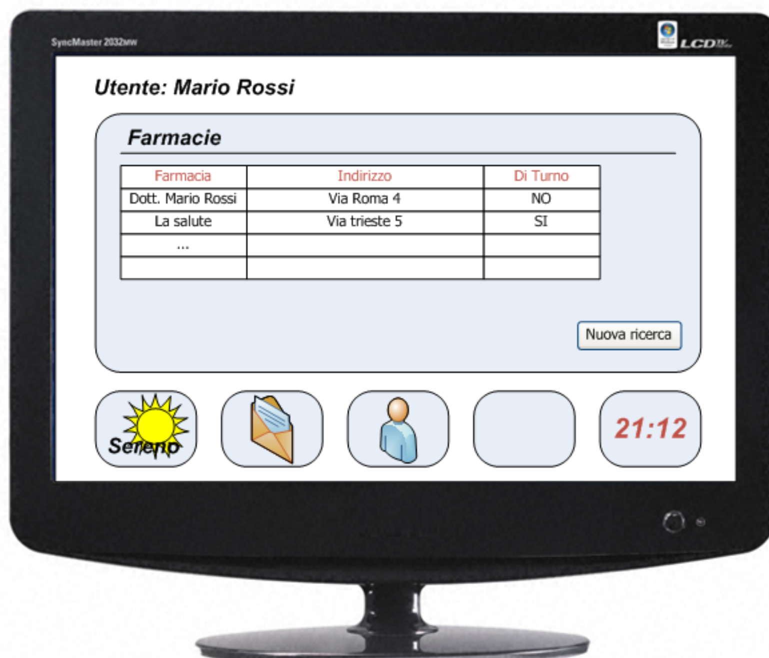


Figura 7.3: Gadget di esempio Farmacie: screenshot del posto “resultState”

# 8 Orologio

*Una seconda classe di gadget applicativi, della quale vale la pena fornire almeno un esempio, è composta da quelli che si prefiggono lo scopo di rendere più facile la vita dei propri utilizzatori, organizzandone la giornata, automatizzandone operazioni ripetitive e noiose, ecc..*

*In questa sede ne viene presentato un tipico esempio, un orologio digitale con funzioni di promemoria, interessante anche da un punto di vista più tecnico: siamo in presenza di una rete di Petri non più banalissima, caratterizzata da un posto esterno di tipo timerState e dalla possibile compresenza di più stati contemporaneamente attivi, infine il gadget richiede un piccolo database locale per la memorizzazione dei promemoria.*

## 8.1 Obiettivi

Progettazione di un gadget che realizzi un piccolo orologio digitale con funzioni di promemoria.

## 8.2 RdP

L'organizzazione del gadget può prevedere:

**normalState:** uno stato in cui è visibile solo l'orologio digitale ridotto a icona

**alertMemoState:** uno per le notifiche dei promemoria

**setMemoState:** uno stato in cui si da possibilità all'utente di impostare un nuovo promemoria

**viewMemoState:** uno stato in cui si visualizzano i promemoria impostati ed eventualmente eliminarli

**tmrState:** un timerState che notifichi la scadenza del prossimo promemoria

In figura 8.1 è mostrata la rete di Petri del gadget. Si noti il parallelismo tra normalState/alertState e gli stati che permettono la gestione delle memo, necessaria per ricevere la notifica della scadenza di un promemoria anche mentre l'utente è impegnato a fare altro.

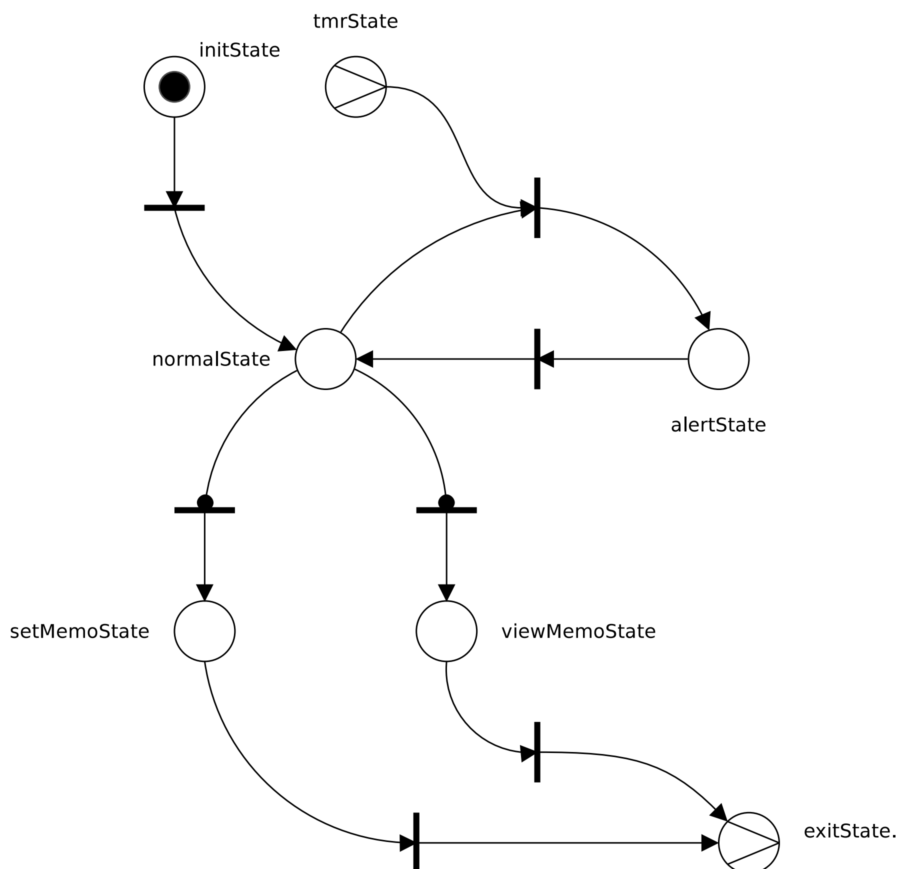


Figura 8.1: Gadget di esempio Orologio: Rete di Petri

## 8.3 Sorgenti Dati

Il gadget necessita di una base dati (userGadgetDB, vedi 5.5.2) locale per la memorizzazione dei promemoria:

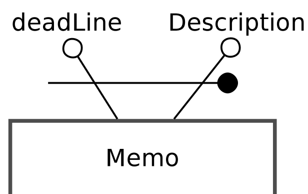


Figura 8.2: Gadget di esempio Orologio: Schema ER della base dati userGadgetDB

## 8.4 Implementazione

```
1 class Orologio < GadgetBase
2
```

```
3     #elenco delle transizioni, da valutare se spostarle in nuovo
          file separato
4     transitions = {
5         Transition.new :inStates => {:initState}, :outStates => {:
              normalState},                              # 0
6         Transition.new :inStates => {:normalState, :tmrState}, :
              outStates => {:alertState},                # 1
7         Transition.new :inStates => {:alertState}, :outStates => {:
              normalState},                              # 2
8         Transition.new :inStates => {:normalState}, :outStates =>
              {:normalState, :setMemoState},             # 3
9         Transition.new :inStates => {:normalState}, :outStates =>
              {:normalState, :viewMemoState},            # 4
10        Transition.new :inStates => {:setMemoState}, :outStates =>
              {:exitState},                              # 5
11        Transition.new :inStates => {:viewMemoState}, :outStates =>
               {:exitState}                              # 6
12    }
13
14    #definizione degli stati
15    class InitState < State
16        def onEnter
17            transitions[0].enable
18        end
19    end
20
21    class NormalState < State
22        showClock()
23        def onClick do
24            transition[3].enable
25        end
26        def onRightClick do
27            transition[4].enable
28        end
29    end
30
31    class AlertState < NormalState
32        def onEnter do
33            @msg = MsgBox.new
34            @msg.value = getCurrentMemoContent()
35        end
```

```
36          def onClick do
37              transition[2].enable
38          end
39      end
40
41      class SetMemoState < State
42          @txtMemo = TextBox.new
43          @txtMemo.value = adaptiveEngine.get :choiseId => :
                memoContent, :maxResults => 1
44
45          @time = TimePicker.new
46          @time.value = adaptiveEngine.get :choiseId => :memoTime, :
                maxResults => 1, :filter => "@value > Time.new"
47
48          @btnOk = Button.new do
49              data = HiperCube.new :time => @time, :memo => @txtMemo
50              System.IO.Database.UserGadgetDB.set("Memo", data)
51              nextMemo = getNextMemo()
52              :tmrState.enableOn nextMemo
53              transition[5].enable
54          end
55
56          @btnCancel = Button.new do
57              transition[5].enable
58          end
59      end
60
61      class ViewMemoState < State
62          @data System.IO.Database.UserGadgetDB.get("Memo")
63          @grdResult = GridResults.new
64          @grdResult.databind do |hq|
65              hq = @data.select :filter => "@time > Time.now"
66          end
67          @btnCancel = Button.new do
68              transition[5].enable
69          end
70      end
71
72      class TmrState < TimerState
73          @nextMemo = getNextMemo()
74          def onEvent do
```

```
75            @nextMemo = getNextMemo()
76            self.enableOn(@nextMemo)
77            transition[1].enable
78        end
79    end
80 end
```

### 8.4.1 UI\normalState.main.rui

```
1 :normalState_dock = Panel.new do
2     @lblTime
3 end
```

### 8.4.2 UI\alertState.main.rui

```
1 :alertState_dock = Panel.new do
2     @lblTime,
3     @msg
4 end
```

### 8.4.3 UI\setMemoState.main.rui

```
1 :setMemoState_main = Panel.new do
2     @txtMemo.render
3     @time.render
4     @btnOk.render
5     @btnCancel.render
6 end
```

### 8.4.4 UI\viewMemo.main.rui

```
1 :viewMemoState_main = Panel.new do
2     @grdResult.render
3     @btnCancel.render
4 end
```

## 8.5 ScreenShots

Anche in questo caso si fornisce uno screenshot indicativo su come, in una reale implementazione, questo gadget potrebbe apparire agli occhi dell'utente: da notare la conpresenza del gadget in dock view (*normalState*, nessun allarme attivo) e in main view (*viewMemoState*, l'utente sta visualizzando i memo che ha impostato).

Figura 8.3: Gadget di esempio Farmacie: screenshot del gadget in visualizzazione eventi con nessun allarme attivo

# 9 Adaptive Engine

*Dopo aver analizzato la progettazione di un paio di gadget applicativi è opportuno fornire almeno un esempio di gadget di sistema, quei particolari componenti di cui altri gadget possono utilizzarne le funzionalità. L'oggetto dell'analisi di questo capitolo è un gadget fondamentale per il successo dell'intero progetto, da cui dipendono le capacità del sistema di adattarsi all'utente: l'adaptiveEngine.*

*Come già preannunciato nella sezione dedicata ai system gadget (vedi sezione 4.3) la progettazione di un simile componente non è per nulla semplice e, sebbene la soluzione qui proposta inizi a mostrarne tutta la complessità, è ancora lontana dall'essere un gadget pronto per l'esercizio.*

## 9.1 Obiettivi

Si vuole realizzare un gadget che fornisca un supporto alla predizione delle scelte dell'utente, mediante due primitive:

| Chiamata base | Descrizione |
|---|---|
| *get(id_scelta, n_prop, filter)* | Ottiene le *n_prop* opzioni che più probabilmente verranno scelte dall'utente (in funzione del precedente comportamento) nella scelta identificata da *id_scelta*, che rispettino i vicoli imposti da *filter*.<br>Nel caso in cui non si disponga di informazioni sufficienti per determinare n_prop diverse opzioni, la chiamata ne restituirà il massimo numero possibile (quindi in generale la funzione restiuisce da 0 a *n_prop* opzioni). |
| *set(id_scelta, scelte_fatta)* | Mette a storia che l'utente ha scelto *scelte_fatte* (array di opzioni) nella scelta identificata da *id_scelta*. Si noti che non viene passato al motore l'elenco completo delle opzioni ma solo quelle selezionate in quanto ritenuto non utile. |

**Nota d'uso:** poichè le opzioni disponibili per una determinata scelta possono variare nel tempo, la chiamata *get()* non garantisce che le proposte restiuite siano effettivamente tra le opzioni possibili nella scelta in oggetto.

## 9.2 RdP

L'intero gadget è costiuito da un unico posto esterno di tipo softwareState che espone entrambe le primitive:

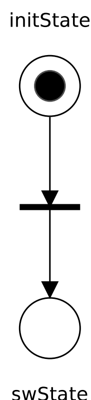


Figura 9.1: Gadget di esempio Adaptive Engine: Rete di Petri

## 9.3 Sorgenti Dati

I dati necessari al gadget sono limitati ad un unico userGadgetDB volto a memorizzare per ogni scelta le opzioni selezionate dall'utente nel passato.

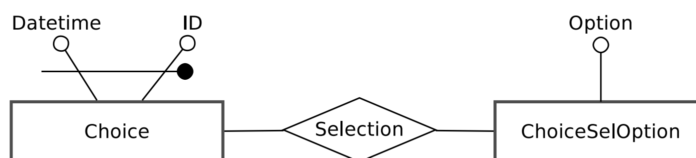


Figura 9.2: Gadget di esempio Adaptive Engine: ER

## 9.4 Implementazione

```
1 class AdaptiveEngine < GadgetBase
2
3   #elenco delle transizioni, da valutare se spostarle in nuovo
      file separato
4   transitions = {
5     Transition.new :inStates => {:initState}, :outStates => {:
        swState}
6   }
7
8
9   #definizione degli stati
```

```
10    class InitState < State
11        def onEnter
12            transitions[0].enable
13        end
14    end
15
16    # Posto di tipo SoftwareState
17    # Tutte le funzioni contenute sono esportate
18    # e devono accettare come parametro un ipercubo
19    class swState < SoftwareState
20        # HQ deve contenere 1 solo punto definito su:
21        # - dimensione: choiceID
22        # - dimensione: n_prop
23        # - dimensione: filter
24        def get(HQ)
25            dataHQ = System.IO.Database.UserGadgetDB.get("Choice")
26            dataHQ.select(HQ[choiceID])
27            dataHQ.evaluateFilter(HQ[filter])
28            num = min dataHQ.count, 2*HQ[n_prop]
29            max = nil
30            mavValue = 0
31            while num > 0 do
32                if not optHash.include? dataHQ[options] then
33                    optHash[dataHQ[options]]=0
34                end
35                optHash[dataHQ[options]] += 1
36            end
37            resAr = optHash.sort {|a,b| a[1]<=>b[1]}
38            resultHQ = HiperCube.new
39            num = min dataHQ.count, HQ[n_prop]
40            i = 0
41            optNum = resAr.lenght - 1
42            while optNum > 0 and i < num do
43                optNum -= 1
44                j = 0
45                while j < resAr[optNum].lenght and i < num do
46                    i = i+1
47                    j = j+1
48                    resultHQ.merge(HiperCube.new resAr[optNum])
49                end
50            end
```

```
51             return resultHQ
52         end
53
54         # HQ deve contenere:
55         # - dimensione: choiceID
56         # - dimensione: choice
57         def set(HQ)
58             HQ.merge(HiperCube.new datetime=>datetime)
59             System.IO.Database.UserGadgetDB.set("Choice", HQ)
60         end
61     end
62 end
```

# 10 Conclusioni

La progettazione di questo complesso sistema di certo non può dirsi conclusa, tuttavia dovrebbe essere sufficiente per dare quello sguardo d'insieme che permetta di apprezzare la flessibilità della soluzione, che in origine era stata concepita per facilitare sviluppo, diffusione ed uso di applicativi basati sulla manipolazione di informazioni (quindi recuperare, pubblicare e organizzare la conoscenza dell'utente), una classe di software che seppur ampia impone forti limitazioni alla struttura degli applicativi stessi, semplificando di molto la progettazione del 2lvlROS.

Già con ciò che è stato fatto però, ci si può facilmente rendere conto che la soluzione proposta, grazie alla particolare struttura del sistema sviluppato, esce dai confini per i quali era stata ideata e può essere utilizzata per un target molto più ampio: quella che prima era una piattaforma per applicazioni ad uso personale da installare in ogni PC casalingo, per fornire un supporto automatizzato a tutta una serie di azioni lunghe e noiose che ognuno di noi è tenuto a fare nella propria vita quotidiana (quali liste della spesa, ricette mediche, recupero informazioni stradali, sugli orari di apertura e chiusura di negozi e servizi, compilazione modulistica ed in generale tutto ciò che rientra nel concetto di maggiordomo virtuale), in realtà ad oggi è più corretto pensarla come ad un semilavorato, che è possibile implementare ed applicare ad esempio anche per la realizzare di totem più semplici da utilizzare, stazioni bancomat, per dotare di GUI grafica standardizzata servizi oggi in terminal server testuali (grazie all'interfaccia grafica astratta la loro realizzazione sarebbe quasi banale), per la gestione delle così dette "porte intelligenti" che operano il riconoscimento facciale per determinare se l'accesso è o meno consentito, come per la realizzazione di quadri digitali che "indovinino" quale porzione dell'archivio foto e video sia opportuno mostrare in un determinato intervallo temporale e per molte altre cose.

Il mondo che si è aperto di fronte a questo progetto, a ben pensare, è veramente vasto e variegato ma non per questo va dimenticato che la fase più critica per ogni software è la primissima diffusione: la possibilità di installare 2lvlROS sopra ad ogni sistema operativo ne renderà molto più morbido il lancio, tanto che potrebbe diventare comune acquistare PC già dotati di OS a doppio livello se uno o più distributori decidessero di convertire in gadget tutti quei programmini che già ora vanno a completare Windows (tipicamente) ma che non sempre sono apprezzati dagli utenti: in questo modo da un lato il distributore riuscirebbe a svilupparli più velocemente (quindi con costi estremamente ridotti) e dall'altro l'utente li troverebbe confinati in un ambiente più ristretto, facilitandone la rimozione se non ne è interessato o facilitandone l'estensione se desidera integrare le funzionalità fornite con altri gadget liberi e/o commerciali (situazione presumibilmente più comune).

Chiaramente la concreta realizzazione del sistema necessita di ancora un po' di lavoro, non tanto a livello concettuale quanto sul piano tecnico: se la struttura dei gadget, le competenze del microkernel e le classi basi del framework sono ben poste e determinate, lo stesso non si può dire dell'architettura interna di molti moduli importanti, ma abbastanza indipendenti, che compongono sia i system gadget che e le librerie, come la gestione della rete (Network Manager), il supporto alla modifica del behaviour in funzione delle precedenti scelte dell'utente (Adaptive Engine) e la struttura dati principale mezzo la quale la conoscenza si muove all'interno del sistema (HiperCube). Si tratta tuttavia di mancanze che non vanno ad intaccare l'architettura di base del sistema e che, avendo a disposizione risorse adeguate, si prevede possano essere colmate velocemente, passando in (relativamente) breve tempo ad una prima reale implementazione del progetto.

# Parte III

# Appendici

# A Processo di sviluppo

*I manuali di ingegneria del software sostengono che in un buon progetto è necessario porre particolare attenzione alla definizione del processo di sviluppo, documentandolo con precisione come ogni altra componente del progetto. In questa appendice è stata quindi raccolta tutta la documentazione del processo di sviluppo utilizzato, pianificando i tempi definendo e monitorando scadenze e analizzando i rischi a cui il progetto è sottoposto.*

*Se il processo di sviluppo più adeguato per il progetto è emerso abbastanza naturalmente dopo breve tempo, così che sua la documentazione è risultata semplice e veloce, lo stesso non si può dire per la pianificazione temporale: il carico di studio variabile durante l'anno accademico e la disposizione casuale di numerosi "impegni imprevisti" hanno reso quasi completamente inattendibile ogni pianificazione temporale; tuttavia come suggerito dall'ingegneria del software, si è scelto di riportare ugualmente tutte le pianificazioni effettuate così che il futuro del progetto possa trarre vantaggio dagli errori di valutazione già compiuti.*

## A.1 Definizione processo di sviluppo

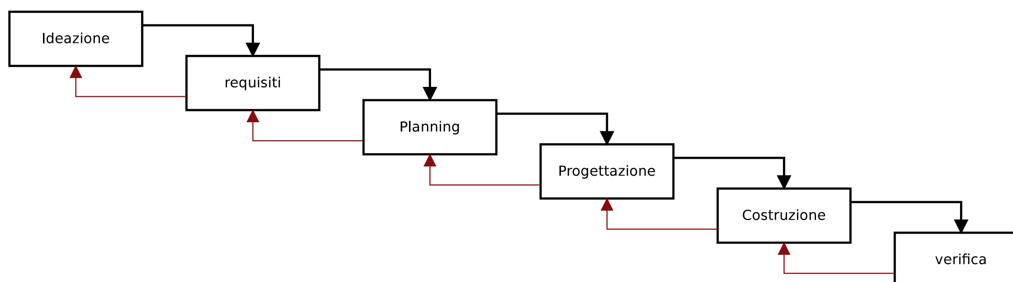


Figura A.1: Processo di Sviluppo

Lo sviluppo del sistema procederà attraverso un processo *a cascata ricorsivo.*

Verranno eseguiti in cascata, ma con la possibilità di rivedere i passi già fatti:

**Ideazione:** presa di coscienza del problema e soluzioni esistenti. Determinazione dell'approccio alla soluzione proposta.

**Requisiti:** raccolta e analisi.

**Planning** di lungo termine riguardante tutto il progetto.

**Progettazione:** traduzione dei requisiti in specifiche software, design architettura e definizioni delle classi

**Costruzione:** scrittura del codice

**Verifica:** testing e debugging

### A.1.1 Processo di sviluppo di una singola fase

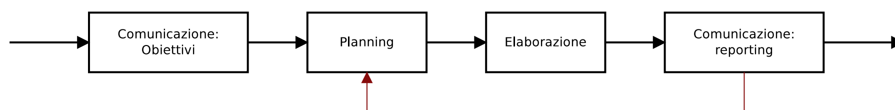


Figura A.2: Processo di Sviluppo: gestione di un singolo step

Inoltre ogni singola fase del processo di sviluppo verrà affrontata come segue:

**Comunicazione** preventiva con il relatore per determinare gli obiettivi a breve termine.

**Planning** a breve termine per stimare e definire i tempi di realizzazione della singola fase.

**Elaborazione** della fase.

**Comunicazione** a posteriori per il reporting di quanto fatto.

Se durante la progettazione si avrà necessità di sviluppare parti del sistema come sotto-progetti, questi verranno gestiti sempre dal medesimo processo di sviluppo qui illustrato.

## A.2 Planning

Il progetto deve essere pronto per il 1 luglio 2010. L'obiettivo di questa fase è rendere questa scadenza rispettabile, considerando carico di studio previsto durante tutto l'anno accademico e impegno richiesto dalle singole fasi del progetto.

Mano a mano che il progetto avanza verranno rese note anche le date di effettivo completamento delle singole fasi ed eventuali ripianificazioni causa ritardi eccessivi.

### A.2.1 Planning v.0

| Fase | Inizio previsto | Inizio effettivo | Completamento previsto | Completamento effettivo |
|---|---|---|---|---|
| Progettazione | 16/11/2009 | 16/11/2009 | 15/01/2010 | - |
| Costruzione | 18/01/2010 | - | 19/03/2010 | - |
| Verifica | 22/03/2010 | - | 20/05/2010 | - |

### A.2.2 Planning v.1

| Fase | Inizio previsto | Inizio effettivo | Completamento previsto | Completamento effettivo |
|---|---|---|---|---|
| Progettazione | 16/11/2009 | 16/11/2009 | 15/04/2010 | 01/05/2010 |
| Costruzione | Annullata | - | - | - |
| Verifica & Stesura Tesi | 16/04/2010 | 05/05/2010 | 30/04/2010 | - |

### A.2.3 Planning v.2

| Fase | Inizio previsto | Inizio effettivo | Completamento previsto | Completamento effettivo |
|---|---|---|---|---|
| Progettazione | 16/11/2009 | 16/11/2009 | 15/04/2010 | 01/05/2010 |
| Costruzione | Annullata | - | - | - |
| Verifica & Stesura Tesi | 16/04/2010 | 05/05/2010 | 30/05/2010 | 10/07/2010 |

## A.3 Analisi dei rischi

Si analizzano ora i rischi potenziali per il progetto stimando probabilità che si verifichino e impatto per ciascun rischio. Per calcolarne l'impatto si è utilizzata la seguente tabella:

| Conseguenze | Valore d'impatto |
|---|---|
| 1: Catastrofico | probabilità x 10 |
| 2: Critico | probabilità x 5 |
| 3: Marginale | probabilità x 2 |
| 4: Trascurabile | probabilità x 1 |

Tabella A.1: Equivalenza conseguenze del rischio e impatto relativo

| ID | Rischio | Categoria | Prob | Conseguenze | Impatto |
|---|---|---|---|---|---|
| A.3.0.1 | Processo mal definito e mal gestito | Processo | 40% | 2 | 200 |
| A.3.0.2 | Progetto non d'interesse per la commissione di laurea | Business | 20% | 1 | 200 |
| A.3.0.3 | Dimensioni eccessive del progetto | Dimensioni | 30% | 2 | 150 |
| A.3.0.4 | Perdita interesse da parte del relatore | Business | 10% | 1 | 100 |
| A.3.0.5 | Capacità troppo limitate di Shooes | Tecnologia | 40% | 3 | 80 |
| A.3.0.6 | Linguaggio non permette sviluppo veloce (causa debugging) | Tecnologia | 15% | 2 | 75 |
| A.3.0.7 | Lentezza apprendimento del linguaggio | Tecnologia | 5% | 2 | 25 |
| A.3.0.8 | Linguaggio non permette costrutto indispensabile | Tecnologia | 3% | 2 | 15 |

Tabella A.2: Rischi del progetto SW

#### A.3.0.1 Processo mal definito e mal gestito

| Rischio: | (Processo) Processo mal definito o mal gestito |
|---|---|
| Conseguenze (2): | Può portare ritardi non limitati, bassa qualità della documentazione, scelte inopportune. In sostanza il fallimento del progetto. |
| Probabilità: | 40% |
| RMMI: | Inserimento nel planning di uno slot da 2 settimane per rivedere, ridefinire e migliorare il processo di sviluppo |

#### A.3.0.2 Progetto non d'interesse per la commissione di laurea

| Rischio: | (Business) Progetto non d'interesse per la commissione di laurea |
|---|---|
| Conseguenze (1): | Il "cliente" non paga. |
| Probabilità: | 20% |
| RMMI: | - |

#### A.3.0.3 Dimensioni eccessive del progetto

| Rischio: | (Dimensioni) Dimensioni eccessive del progetto |
|---|---|
| Conseguenze (2): | Incapacità di sviluppare documentazione, codice e slide di presentazione entro luglio e conseguentemente:<br>• Rimozione implementazione dal progetto (si presenterà solo progettazione)<br>• Spostamento laurea a settembre |
| Probabilità: | 30% |
| RMMI: | Sviluppo planning per monitoraggio costante avanzamento e ritardi |

#### A.3.0.4 Perdita interesse da parte del relatore

| Rischio: | (Business) Perdita interesse da parte del relatore |
|---|---|
| Conseguenze (1): | Annullamento progetto |
| Probabilità: | 10% |
| RMMI: | Frequenti controlli di avanzamento e convalida del lavoro svolto |

#### A.3.0.5 Capacità troppo limitate di Shoes

| Rischio: | (Tecnologia) Capacità troppo limitate di Shoes |
|---|---|
| Conseguenze (3): | Shoes può non essere adeguato al progetto perchè troppo giovane. L'usarlo ugualmente può provocare consistenti perdite prestazionali, effetti visivi scadenti o fastidiosi per l'utente (refresh continui, lentezza di risposta, errori nel rendering.. ecc..) |
| Probabilità: | 40% |
| RMMI: | Inserire nell'architettura un layer di interfaccia con il motore di rendering (Shoes).<br>In caso di problemi sarà possibile sostiuire Shoes con un altro motore di rendering (esempio Irrlicht vedi http://irr.rubyforge.org/) |

#### A.3.0.6 Linguaggio non permette sviluppo veloce

| Rischio: | (Tecnologia) Linguaggio non permette sviluppo veloce |
|---|---|
| Conseguenze (2): | Ruby può (sotto determinate condizioni) presentare un debugging lento e difficile a causa di una gestione degli errori non ottima. |
| Probabilità: | 15% |
| RMMI: | Affidarsi ad una buona progettazione. |

#### A.3.0.7 Lentezza apprendimento del linguaggio

| Rischio: | (Tecnologia) Lentezza apprendimento del linguaggio |
|---|---|
| Conseguenze (2): | Ruby è un linguaggio nuovo per lo sviluppatore. Se l'apprendimento non è sufficientemente rapido i tempi di sviluppo aumentano anche di molto. |
| Probabilità: | 5% |
| RMMI: | - |

#### A.3.0.8 Linguaggio non permette costrutto indispensabile

| Rischio: | (Tecnologia) Linguaggio non permette costrutto indispensabile |
|---|---|
| Conseguenze (3): | Ruby è un linguaggio nuovo per lo sviluppatore. Esiste la possibilità che questo per sua natura non permetta l'uso di un costrutto indispensabile per lo sviluppo. |
| Probabilità: | 2% |
| RMMI: | Cambio linguaggio in corso d'opera. Estremamente costoso in termini di tempo! (ma fattibile recuperando tutta la modellazione e progettazione) |

# B Prototipi

*Durante l'analisi dei requisiti e la prima parte di design capita spesso di trovarsi di fronte a scelte progettuali importanti ma allo stesso tempo di non chiara soluzione: la scarsa conoscenza di una nuova tecnologia, di un algoritmo o semplicemente del supporto che un framework fornisce relativamente ad un ambito di interesse porta inevitabilmente a dover effettuare delle "prove sul campo".*

*Questa appendice raccoglie queste "prove sul campo", in gergo chiamate prototipi, documentandoli affichè in qualsiasi momento sia possibile risalire all'esito del test.*

## B.1 Ruby: Accesso dinamico ai WS

### B.1.1 Scopo

I web Services sono una componente importante nei moderni applicativi, in quanto costituiscono uno dei principali vettori per lo scambio di informazioni sulla rete. In un progetto che ha come punto cardine l'accesso alle informazioni, non poteva mancare una valutazione sulle capacità intrinseche del linguaggio che si andrà ad utilizzare per quanto riguarda l'accesso ai web services.

Lo scopo di questo prototipo è quindi quello di verificare le modalità di accesso ai web service SOAP da parte del linguaggio ruby (puro).

### B.1.2 Descrizione

Accedere ai WS SOAP con ruby è decisamente semplice:

```
1  #Creazione dell'oggetto per la RPC, passandogli come argomento
2  #l'url del descrittore WSDL
3  url = "http://www.webservicex.net/periodictable.asmx?WSDL"
4  @soap = SOAP::WSDLDriverFactory.new(url).create_rpc_driver
5
6  #ora @soap ha tutti i metodi del WS come metodi di oggetto
7  #chiamo il metodo GetAtomicNumber
8  result = @soap.GetAtomicNumber( "ElementName" =>"Argon")
9
10 #result è un oggetto che incapsula i risultati dell'operazione
```

```
11 #tali risultati sono estraibili con l'unico metodo di istanza
12
13 puts result.getAtomicNumberResults
```

E' anche possibile accedere in modo più dinamico allo stesso web service senza conoscere il nome dei metodi (il nomero e il nome dei parametri deve essere conosciuto però)

```
1  url = "http://www.webservicex.net/periodictable.asmx?WSDL"
2  @soap = SOAP::WSDLDriverFactory.new(url).create_rpc_driver
3
4  #ottengo un array con i nomi dei metodi supportati
5  array_metodi_str = @soap.singleton_methods
6
7  #scelto il metodo da chiamare posso
8
9  result = @soap.send(array_metodi_str[3], hash_di_parametri)
10
11 recupero dinamicamente i risultati
12
13 puts result.send(result.singleton_methods[0])
```

## B.2 Rails: Esporre WS

### B.2.1 Scopo

Dopo aver verificato le capacità di Rubi di accedere alle sorgenti dati via Web Services, si vuole verificare la possibilità di esporne di nuovi mediante la stessa tecnologia (in altre parole si vuole verificare se è possibile sviluppare e accedere ai WS in modo tecnologicamente uniforme). Rails è uno dei framework Rubi di maggior successo per lo sviluppo di applicazioni web e, nello scenario più comune, costituisce il punto di partenza nella costruzione di servizi di rete.

Si vuole pertanto verificare le capacità di Rails di fornire un accesso ai dati tramite Web Services.

### B.2.2 Descrizione

Rails implementa nativamente i WS di tipo REST: è sufficiente infatti aggiungere l'estensione .xml alle richieste per ottenere non solo un output basato su xml appunto ma anche pienamente conforme allo standard restful.

Da rails 2 è stata rimossa invece la possibilità di esporre ws d tipo soap, perchè ritenuti troppo complessi per le operazioni che supportano (rest fa le stesse cose con decisamente minor complessità); tuttavia è possibile implementare ws soap con opportune estensioni.

## B.3 Shoes: convivenza widget

### B.3.1 Scopo

Shoes è un piccolo toolkit, sviluppato quasi a livello home-made da Why, che permette di svluppare in modo semplice e rapido interfacce grafiche in Ruby.

Un'applicativo che sfrutta shoes per la propria UI può essere eseguito senza alcuna modifica al codice (in accordo con i principi filosofici di Ruby) in Windows, Linux e Mac, ottenendo in ogni caso l'interfaccia grafica nativa del sistema operativo sottostante.

Per valutare la possibilità di sfruttare Shoes all'interno di questo progetto, tuttavia, è necessario testare la sua capacità di gestire più entità contemporee e parallele senza perdite di prestazioni e/o consistenza, in particolare si vuole verificare il comportamento di un applicativo che faccia convivere all'interno di una stessa finestra più widget (parti componenti) anche molto diversi tra loro.

### B.3.2 Descrizione

Si è provato a inserire più esepi all'interno di una stessa finestra:

1. Si è sostituito la classe base da Shoes.app a Widget
2. Si è incluso tutto il codice dicchiarativo (quello fuori dai metodi) nel metodo inizialize
3. Si sono inseriti due diversi Widget così creati in uno stack dentro la stessa finestra

Il comportamento che si è verificato è il seguente:

- L'uso dei widget è possibile solo se definiti all'interno dello stesso file del programma principale, se i widget si trovano in file diversi e si tenta di importarli con *require*, shoes segnala un errore.
- Non si è riusciti a passare *height* e *width* ai widget: self.height rimane sempre 0 all'interno del widget. La proprietà inoltre non è impostabile dall'interno e pare "di sola lettura non segnalata".
- Nessun problema riscontrato in multithreading: animazioni contemporanee in widget diversi vengono correttamente eseguite.

## B.4 Ruby: eventi

### B.4.1 Scopo

Molti dei linguaggi moderni hanno fatto proprio il concetto di evento, evoluzione del vecchio puntatore a funzione, soprattutto per quanto riguarda la gestione delle interfacce grafiche e delle funzioni di call-back associate agli input degli utenti.

In ruby non esiste nativamente il concetto di evento, tuttavia si vuole indagare sulla possibilità di definire un costrutto dal comportamento analogo.

### B.4.2 Descrizione

In ruby èpossibile implementare degli event handler nel seguente modo:

```
1  class EventBase
2      def initialize
3          @listeners = Hash.new
4      end
5
6      def listen_event(name, *func, &p)
7          if p
8              (@listeners[name] ||= Array.new) << p
9          else
10             (@listeners[name] ||= Array.new) << func[0]
11         end
12     end
13
14     def ignore_event(name, func)
15         return if !@listeners.has_key?(name)
16         @listeners[name].delete_if { |o| o == func }
17     end
18
19     def trigger_event(name, *args)
20         return if !@listeners.has_key?(name)
21         @listeners[name].each { |f| f.call(*args) }
22     end
23 end
24
25
26 class MyClass < EventBase
27     def raise_event1(*args)
28         trigger_event(:event1, *args)
29     end
30
31     def raise_event2(*args)
32         trigger_event(:event2, *args)
33     end
34 end
35
36 class TestListener
37     def initialize(source)
38         source.listen_event(:event1, method(:event1_arrival))
```

```
39          source.listen_event(:event2) do |*a|
40              puts "event 2 arrival, args #{a}"
41          end
42      end
43
44      def event1_arrival(*a)
45          puts "Event 1 arrived, args #{a}"
46      end
47  end
48
49  s = MyClass.new
50  l = TestListener.new(s)
51
52  s.raise_event1("here is event 1")
53  s.raise_event2("here is event 2")
```

## B.5 Ruby: ereditarietà multipla

### B.5.1 Scopo

L'ereditarietà multipla non sempre è benvista dai progettisti di linguaggio, ma talvolta può essere utile, ad esempio può essere un modo di definire le funzionalità che un gadget implementa (se eredita da una particolare classe allora espone una ben determinata funzionalità).

Si desidera pertanto verificare la possibilità di usare l'ereditarietà multipla (o un comportamento analogo) in ruby.

### B.5.2 Descrizione

Ruby non consente l'ereditarietà multipla, tuttavia supporta il concetto di *mixin* che può aggirare il problema. Segue esempio illuminante:

```
1  module Inutile
2      def hello
3          "Saluti da #{self.class.name}"
4      end
5  end
6
7  class Tokyo
8      include Inutile
9  end
```

Risultato dell'esecuzione:

```
1 > tk = Tokyo.new
2 > tk.hello
3 => "Saluti da Tokyo"
```

# Bibliografia


[1] Roger S.Pressman. *Principi di Ingegneria del Software* McGraw-Hill, Milano, quinta edizione, 2008.

[2] G. Clemente, F. Filira, M.Moro. *Sistemi Operativi. Architettura e programmazione concorrente.* Libreria Progetto, Padova, seconda edizione, 2006.

[3] A. S. Tanenbaum. Computer Networks. Practise Hall, quarta edizione, 2002.

[4] *Treddi.com, il portale italiano sulla grafica 3D* (Internet). Disponibile all'indirizzo www.treddi.com. (consultato il 14/07/2010)

[5] *Shooes! cross-platform toolkit for writing graphical apps easily and artfully using Ruby* (Internet). Disponibile all'indirizzo http://shoes.heroku.com/.

[6] Rubyonrails.org (Internet). *Ruby on Rails: Screencast. Show, don't tell: Seeing is believing.* Disponibile all'indirizzo http://rubyonrails.org/screencasts

[7] Ruby-lang-org (Internet). *Ruby in Twenty Minutes.* Disponibile all'indirizzo www.ruby-lang.org/en/documentation/quickstart

[8] Websters-online-dictionary.org (internet). *Webster's Online Dictionary, Definition of computer literacy.*
Disponibile su http://www.websters-online-dictionary.org/definitions/computer+literacy

# Indice analitico

2lvlOS: Second Level Operating System, 15

Adaptive Engine, 59, 97

Bootloading, 47, 49

Classi, organizzazione, 27
Culture, 60

Data flow, 65
Desktop application, 7
Dock view, 33, 34
Duck Typing, 16

Event Handler, 58

File System, 71
Filtro, 65, 67
Framework vs System Gadget, 47

Gadget, 15, 17, 78
Gadget amministrativi, 29
Gadget applicativi, 29
Gadget desktop, 29
Gadget di servizio, 29
Gadget, comunicazione, 53
Gadget, nomenclatura, 29
Gadget, tipi, 29
Globalization, 60

HiperCube, 68

Internationalization, 60
Ipercubi, classe HiperCube, 68
Ipercubo, 66
Ipercubo, punti pieni, 66
Ipercubo, punti vuoti, 66
Irrlicht, 56

Java, 67

Kernel, 45, 81
Kernel a Strati, 45
Keyrings, 49

Localization, 60

Main view, 33, 34
Matz, 16
Microkernel, 45, 81

Network manager, 61

Planning, 106
Processo di sviluppo: a cascata ricorsivo, 105
Prototipi, 111
Provider, 18

Rails, 112
Render Engine, 56
Reti di Petri, 30, 79
Rischi, 108
Ruby, 15

Scelta, 65, 67
Security, 49, 70
SharedGadgetConfig, 63
SharedGadgetDB, 73, 81
SharedSystemConfig, 63
SharedSystemDB, 73, 81
Shoes, 43, 113

Sistema di pubblicazione, 36

Universo, 37
UserGadgetConfig, 63
UserGadgetDB, 73, 81
UserSystemConfig, 63
UserSystemDB, 73, 81
Utenti, Sistema, 48

Visione, 37

Web Application, 8
Web Services, 111

Yukihiro Matsumoto, 16

# Elenco delle figure


1.1 Tipica Desktop Application: Microsoft Excel . . . . . . . . . . . . . . . . . . 9
1.2 Tipica Web Application: il foglio elettronico di Google Docs . . . . . . . . . 10
1.3 Uno dei più completi IDE disponibili in rete: Eclipse . . . . . . . . . . . . . 11
1.4 Architettura classica e con 2lvlOS a confronto . . . . . . . . . . . . . . . . . 15
1.5 Esempio di come potrebbe apparire il 2lvlROS ed alcuni suoi Gadget . . . . 17
1.6 Gadget - Use Case Diagram: Utente - Livello 0 . . . . . . . . . . . . . . . . 18

2.1 Architettura - DFD - Livello 0 . . . . . . . . . . . . . . . . . . . . . . . . . 25
2.2 Architettura - DFD - Livello 1 . . . . . . . . . . . . . . . . . . . . . . . . . 26
2.3 Architettura - DFD - Livello 1 - Aree di responsabilità . . . . . . . . . . . . 27
2.4 Diagramma delle classi - Livello 0 . . . . . . . . . . . . . . . . . . . . . . . 27

3.1 Gadget: Struttura concettuale . . . . . . . . . . . . . . . . . . . . . . . . . 30
3.2 Gadget - UI - Use Case Diagram: Gadget - Livello 0 . . . . . . . . . . . . . 33
3.3 Gadget - UI: Esempio di interfacce Main view e Dock view . . . . . . . . . . 34
3.4 Gadget - UI - Diagramma di sequenza: Processo di rendering . . . . . . . . 35
3.5 Pubblicazione dei gadget: esempio di visione . . . . . . . . . . . . . . . . . 37
3.6 Pubblicazione dei gadget: esempi di assegnazione dei gadget alle visioni . . 38

4.1 Kernel: architetture classiche di riferimento . . . . . . . . . . . . . . . . . . 45
4.2 Microkernel, Sicurezza: esempio d'uso dei Lock nel caso di un client e-Mail . 51
4.3 Microkernel, Sicurezza: esempio di foresta di risorse . . . . . . . . . . . . . 51
4.4 Tipica comunicazione tra gadget locali. Caso SIGadget . . . . . . . . . . . 55
4.8 Diagramma delle classi - Livello 1 - Kernel . . . . . . . . . . . . . . . . . . 55
4.5 Tipica comunicazione tra gadget locali. Caso MIGadget con errore. . . . . . 56
4.6 Tipica comunicazione tra gadget remoti. Caso SIGadget. . . . . . . . . . . . 57
4.7 Tipica comunicazione tra gadget remoti. Caso SIGadget con errore di rete. . 57
4.9 Irrlicht: diverse GUI create utilizzando lo stesso motore 3D. Si apprezzi la flessibilità della tecnica. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58

5.1 Schema dei diversi tipi di chiavi di configurazione . . . . . . . . . . . . . . 63
5.2 Diagramma delle classi - Livello 1 - Configuration . . . . . . . . . . . . . . 64
5.3 Diagramma delle classi - Livello 3 - Configuration . . . . . . . . . . . . . . 64
5.4 HiperCube: Struttura implementativa . . . . . . . . . . . . . . . . . . . . . 68
5.5 Diagramma delle classi - Livello 1 - Data . . . . . . . . . . . . . . . . . . . 70

5.6 Diagramma delle classi - Livello 1 - Security . . . 70
5.7 Diagramma delle classi - Livello 2 - Security . . . 70
5.8 Diagramma delle classi - Livello 1 - UI . . . 71
5.9 Schema su come vengono interpretati i permessi di accesso . . . 71
5.10 Schema dei diversi database supportati e loro ruoli . . . 73
5.11 Diagramma delle classi - Livello 1 - IO . . . 74
5.12 Diagramma delle classi - Livello 2 - IO . . . 74
5.13 Network layers . . . 75
5.14 System.IO.Net: Connessioni tra macchine remote . . . 77
5.15 Diagramma delle classi - Livello 2 - IO: Network Layers . . . 78
5.16 Diagramma delle classi - Livello 1 - Gadget . . . 78
5.17 Diagramma delle classi - Livello 2 - Gadget . . . 79

6.1 Basi dati interne: Modello ER - livello 0 . . . 82

7.1 Gadget di esempio Farmacie: Rete di Petri . . . 86
7.2 Gadget di esempio Farmacie: screenshot del posto “filterState” . . . 90
7.3 Gadget di esempio Farmacie: screenshot del posto “resultState” . . . 90

8.1 Gadget di esempio Orologio: Rete di Petri . . . 92
8.2 Gadget di esempio Orologio: Schema ER della base dati userGadgetDB . . . 92
8.3 Gadget di esempio Farmacie: screenshot del gadget in visualizzazione eventi con nessun allarme attivo . . . 96

9.1 Gadget di esempio Adaptive Engine: Rete di Petri . . . 98
9.2 Gadget di esempio Adaptive Engine: ER . . . 98

A.1 Processo di Sviluppo . . . 105
A.2 Processo di Sviluppo: gestione di un singolo step . . . 106

# Elenco delle tabelle

1.1 Desktop e Web application a confronto . . . . . . . . . . . . . . . . . . . . . . 8

A.1 Equivalenza conseguenze del rischio e impatto relativo . . . . . . . . . . . . 108
A.2 Rischi del progetto SW . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 108