Caso di studio: Progetto di un Digital Sound Recorder

Giacomo Gabrielli

Adattato da [Paltor99]; altre sorgenti: [Lami07]

Sistema Embedded

- Sistema il cui funzionamento è controllato da un calcolatore la cui presenza non è immediatamente deducibile

- I sistemi embedded sono tipicamente caratterizzati dai seguenti requisiti:
  - Requisiti real-time
  - Interfacciamento con sensori/attuatori
  - Mobilità (ridotte dimensioni) e connettività
  - **Basso consumo di potenza**

Specifica dei Requisiti

- La prima rappresentazione dei requisiti è sempre in linguaggio naturale perché per la loro realizzazione devono essere coinvolti diversi *stakeholders*
- Il linguaggio naturale ha il vantaggio di essere comprensibile da tutti ma lo svantaggio di essere intrinsecamente ambiguo

Quando i requisiti necessitano di essere maggiormente dettagliati, possono essere rappresentati in modo più tecnico:

  - Modelli di sistema (UML)
  - Metodi formali
  - …

- Una recente indagine indica che il 79% dei documenti di requisiti sono scritti in linguaggio naturale, il 16% in linguaggio naturale strutturato e solo il 5% usando linguaggi formali

Qualità dei Requisiti

Caratteristiche di qualità dei requisiti:

- Correttezza
- Non ambiguità
- Completezza
- Consistenza
- Importanza
- Stabilità
- Verificabilità
- Modificabilità
- Tracciabilità
- Comprensibilità
- Fattibilità
- Livello di dettaglio adeguato

Qualità dei Requisiti (ii)

- **Correttezza**: i requisiti che sono implementati devono riflettere il comportamento atteso
- **Non ambiguità**: il requisito deve avere solo una possibile interpretazione
- **Completezza**: tutti gli elementi importanti che sono rilevanti per soddisfare l'utente devono essere considerati
- **Consistenza**: i requisiti devono essere consistenti verso loro stessi e verso i *constraint* importanti
- **Valutati per importanza**: ogni requisito deve essere valutato in termini di importanza, cioè di quanto esso è essenziale per il successo del progetto
- **Stabilità**: facilità che un requisito cambi
- **Verificabilità**: tutti i requisiti devono essere verificabili, cioè esiste un processo per controllare se il requisito è soddisfatto o no

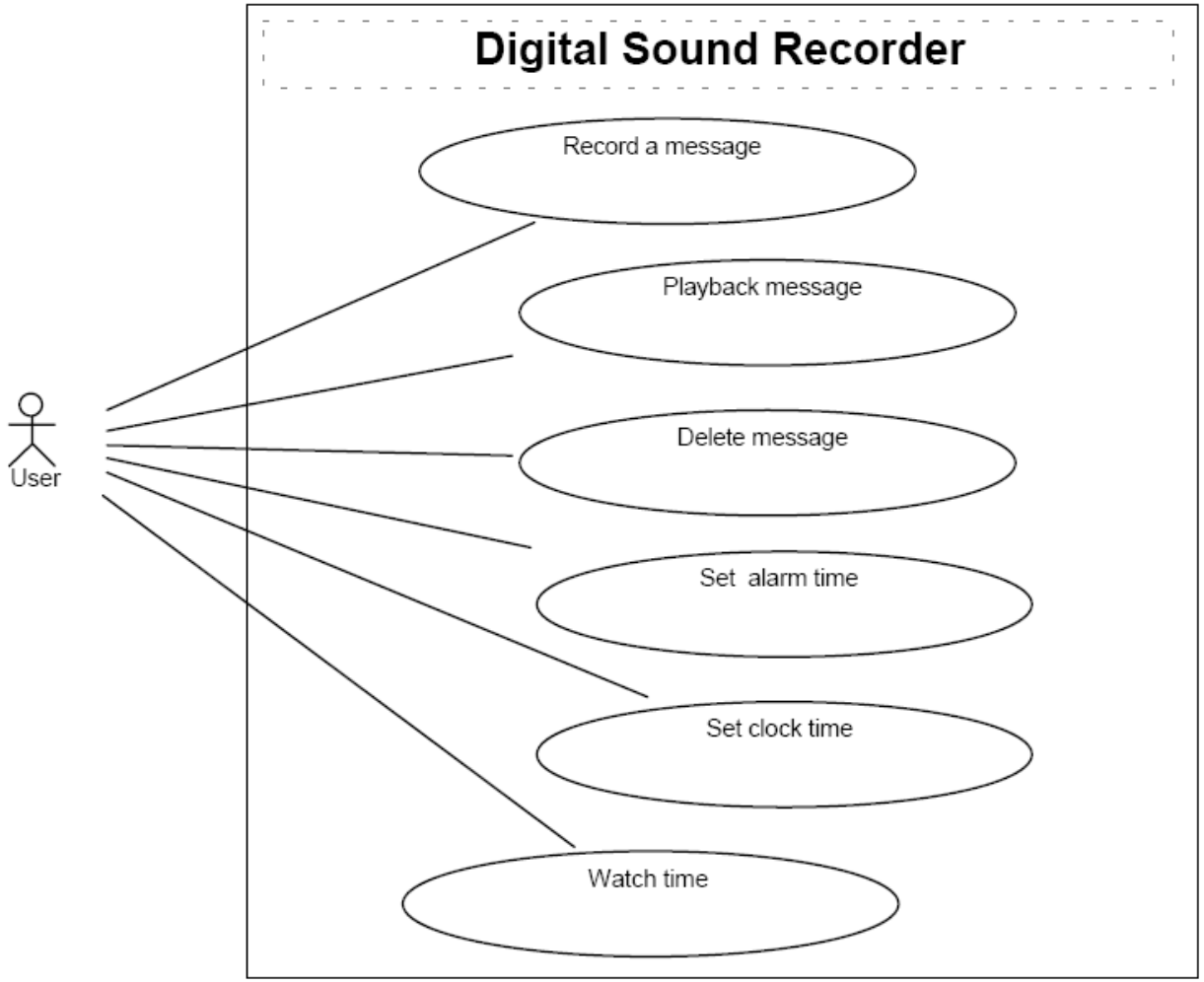
Qualità dei Requisiti (iii)

- **Modificabilità**: tutti i requisiti devono essere modificabili
- **Fattibilità**: tutti i requisiti devono essere implementati con le risorse, la tecnologia e il budget disponibili
- **Giusto livello di dettaglio**: l'informazione contenuta nel requisito permette di ottenere la giusta comprensione per dar luogo all'implementazione

Specifica dei Requisiti Utente con l'UML

- Prerequisito: fase di identificazione degli utenti (attori)
- Casi d'uso
  - In UML possono essere espressi tramite i **diagrammi dei casi d'uso**
- Scenari
  - In UML possono essere espressi tramite i **diagrammi di sequenza**

Casi d'Uso


Digital Sound Recorder
Record a message
Playback message
Delete message
Set alarm time
Set clock time
Watch time
User

Casi d'Uso (ii)

- Spesso i casi d'uso sono poi raffinati (tramite il linguaggio naturale):

USE CASE #	<the name is the goal as a short active verb phrase>	
Goal in Context	<a longer statement of the goal in context if needed>	
Scope & Level	<what system is being considered black box under design> <one of: Summary, Primary Task, Sub-function>	
Preconditions	<what we expect is already the state of the world>	
Success End Condition	<the state of the world upon successful completion>	
Failed End Condition	<the state of the world if goal abandoned>	
Primary, Secondary Actors	<a role name or description for the primary actor>. <other systems relied upon to accomplish use case>	
Trigger	<the action upon the system that starts the use case>	
Description	**Step**	**Action**
	1	<put here the steps of the scenario from trigger to goal delivery, and any cleanup after>
	2	<...>
	3	
Extensions	**Step**	**Branching Action**
	1a	<condition causing branching> : <action or name of sub-use case>
Sub-Variations		**Branching Action**
	1	<list of variations>

NOTA: Tracciabilità

- Si dovrebbe tener traccia delle informazioni che via via si introducono:
  - Traccia dei requisiti, nei vari livelli di dettaglio
  - Data una qualsiasi funzionalità implementata, si deve poter risalire al relativo requisito/caso d'uso
  - Se si introducono modifiche ad un requisito, si deve poter risalire alla funzionalità implementata (nel progetto)

Interazioni del Sistema con l'Ambiente Circostante

- Un sistema embedded tipicamente interagisce in maniera costante con l'ambiente circostante
- 1a FASE: Considerare il sistema come una black-box che reagisce agli stimoli provenienti dall'ambiente

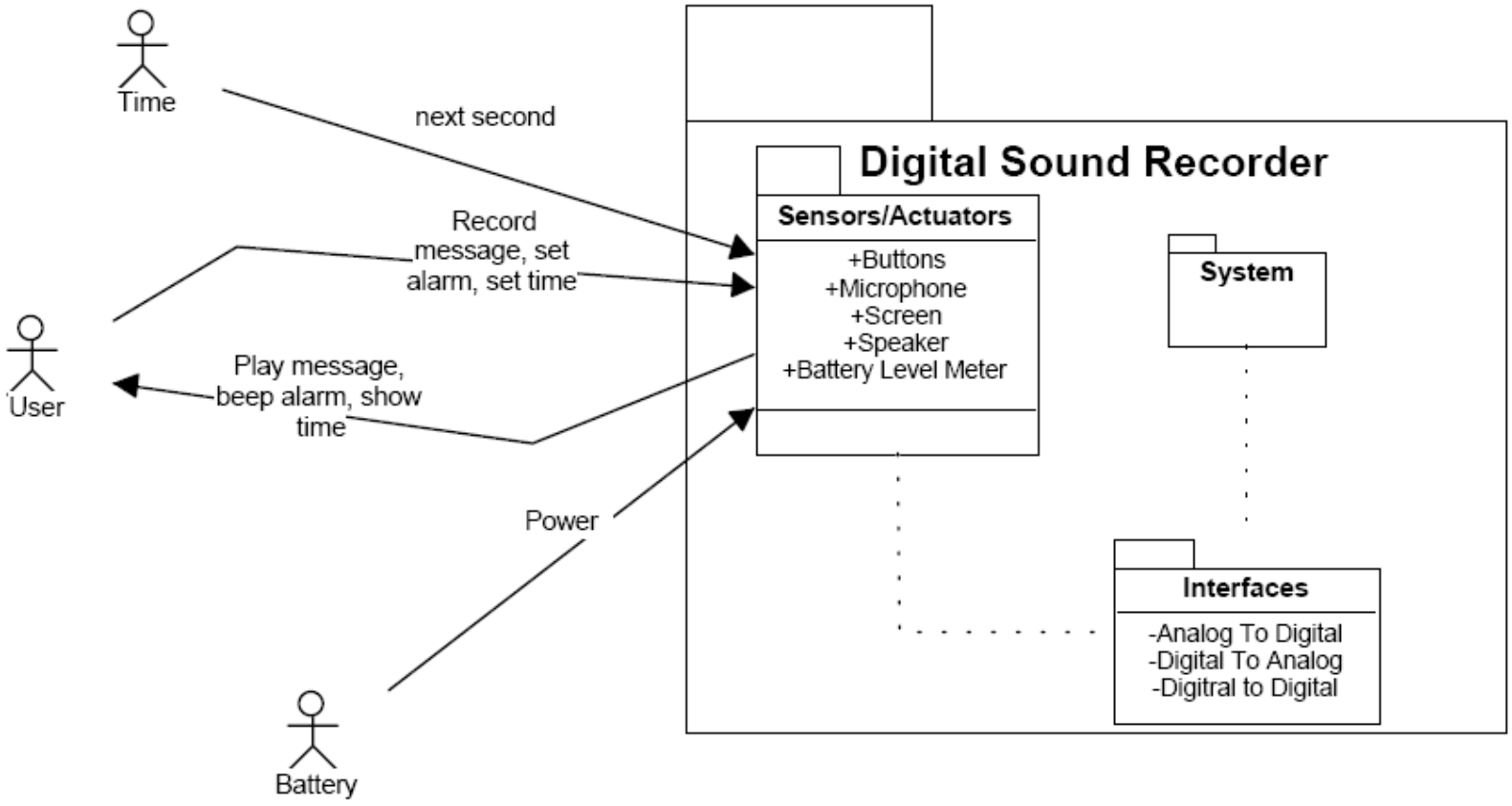
- Identificare elementi dell'ambiente (a questo livello sono considerati anch'essi degli attori):
  - Utente
  - Batteria
  - *Tempo*

Interazioni del Sistema con l'Ambiente Circostante (ii)

- Identificare le interfacce (sensori e attuatori) che consentono al sistema di scambiare messaggi con gli agenti
- Es.:
  - Sensori:
    - Pulsanti
    - Microfono
  - Attuatori:
    - Speaker
    - Schermo LCD

  Questi consentono di scambiare messaggi con l'attore 'Utente'

Interazioni del Sistema con l'Ambiente Circostante (iii)



Scenari

- Mettere in evidenza le interazioni degli attori con il sistema
- Tipico utilizzo: descrivere la sequenza degli eventi che si verificano nell'ambito di un caso d'uso
- Solitamente gli attori corrispondono agli utenti coinvolti nei rispettivi casi d'uso

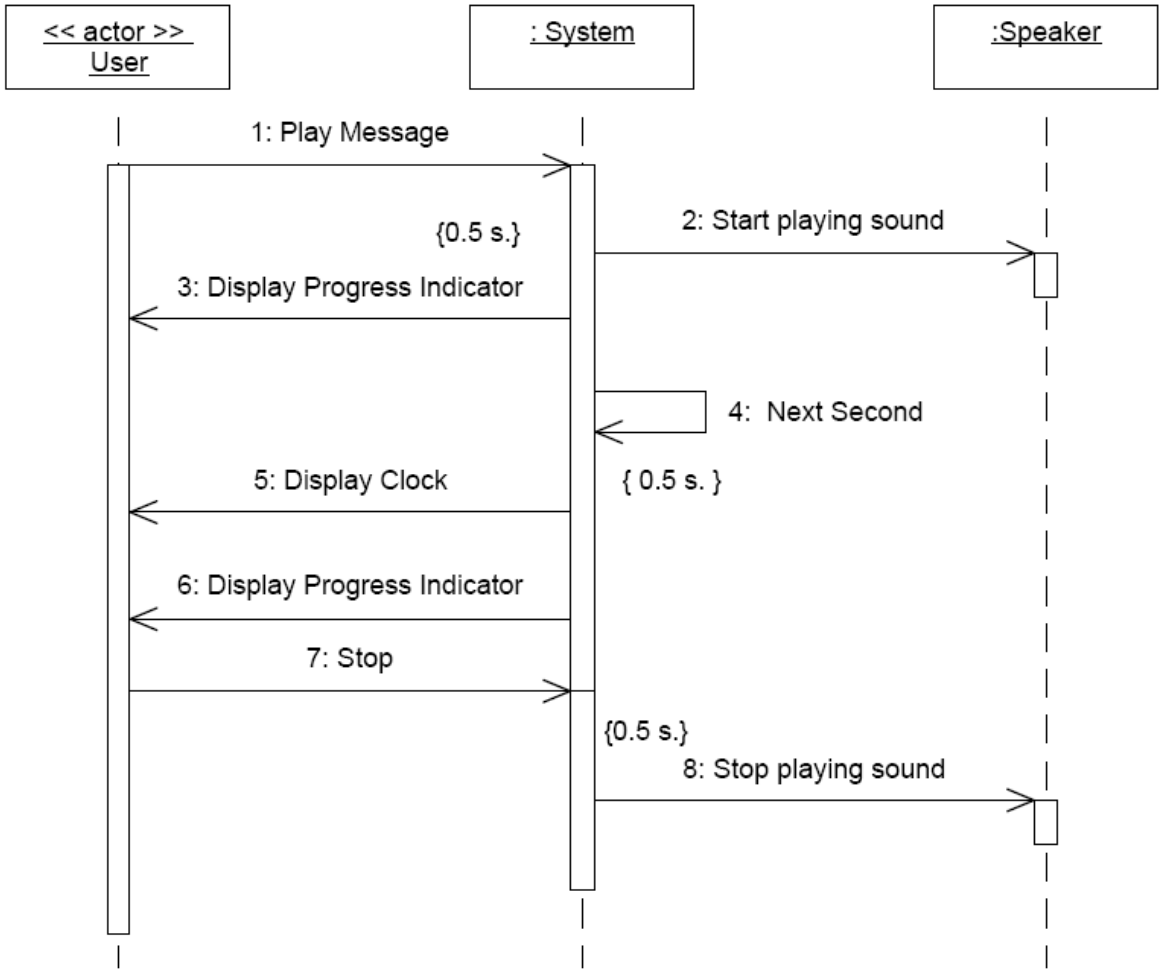
Diagrammi di Sequenza UML

- Un diagramma di sequenza cattura il comportamento di un singolo scenario
- Un diagramma di sequenza mette in evidenza gli oggetti e i messaggi che vengono scambiati tra loro
- Anche i diagrammi di sequenza possono essere utilizzati in fasi distinte del processo di sviluppo sw
  - Specifica dei Requisiti: descrizione scenari
  - Progetto: descrizione interazione tra oggetti veri e propri (istanze di classi nel codice)

Diagrammi di Sequenza UML (ii)

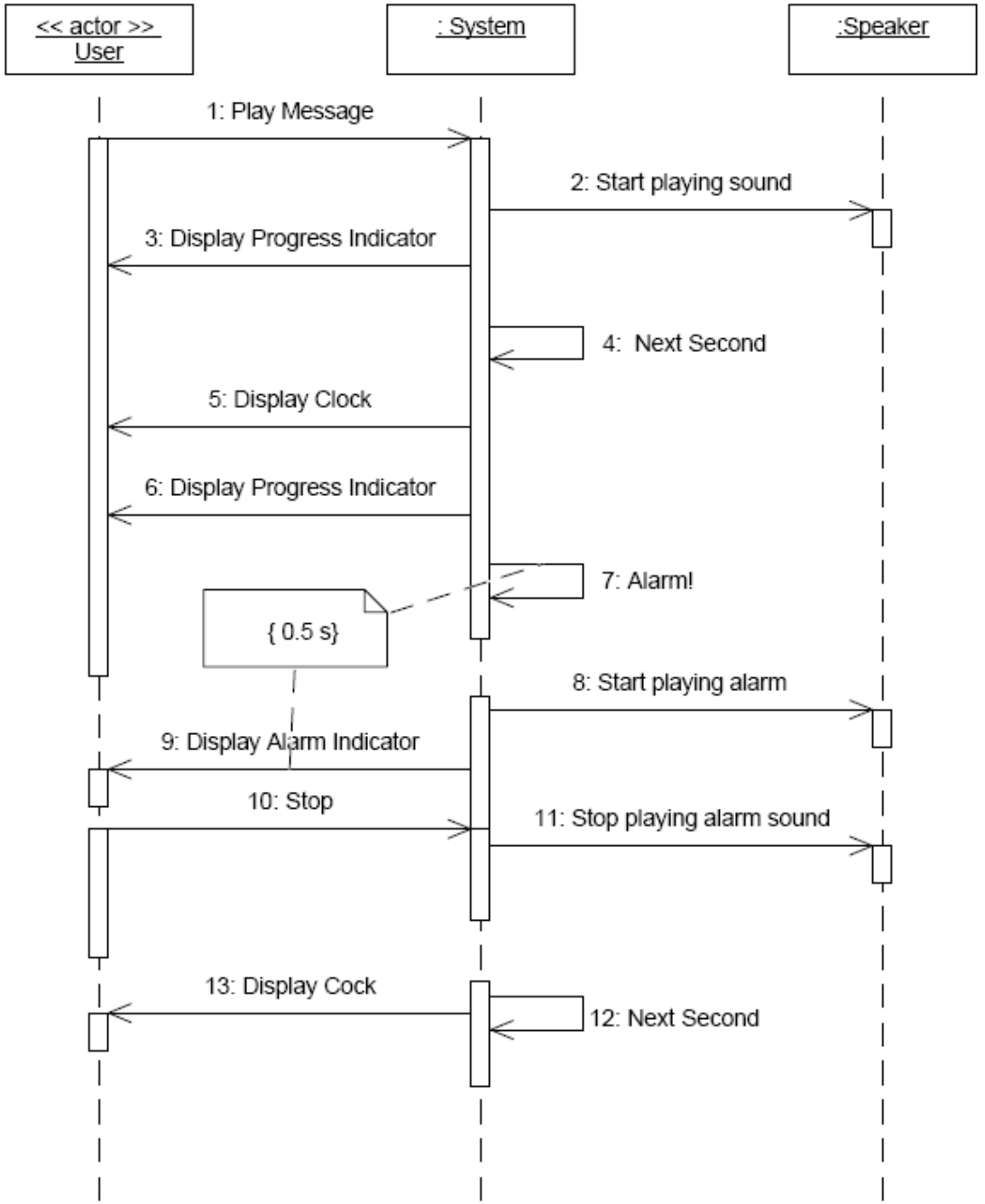
- Nella fase di progetto i diagrammi di sequenza consentono di specificare molti tipi di interazioni (= invocazione di metodi su oggetti)
  - Invocazione sincrona/asincrona
    - Multithreading
  - Flussi iterativi, condizionali, ...
  - Ritorno esplicito delle funzioni
- Molte caratteristiche introdotte dallo standard UML 2.0

Scenario:
“Riproduzione di un messaggio”

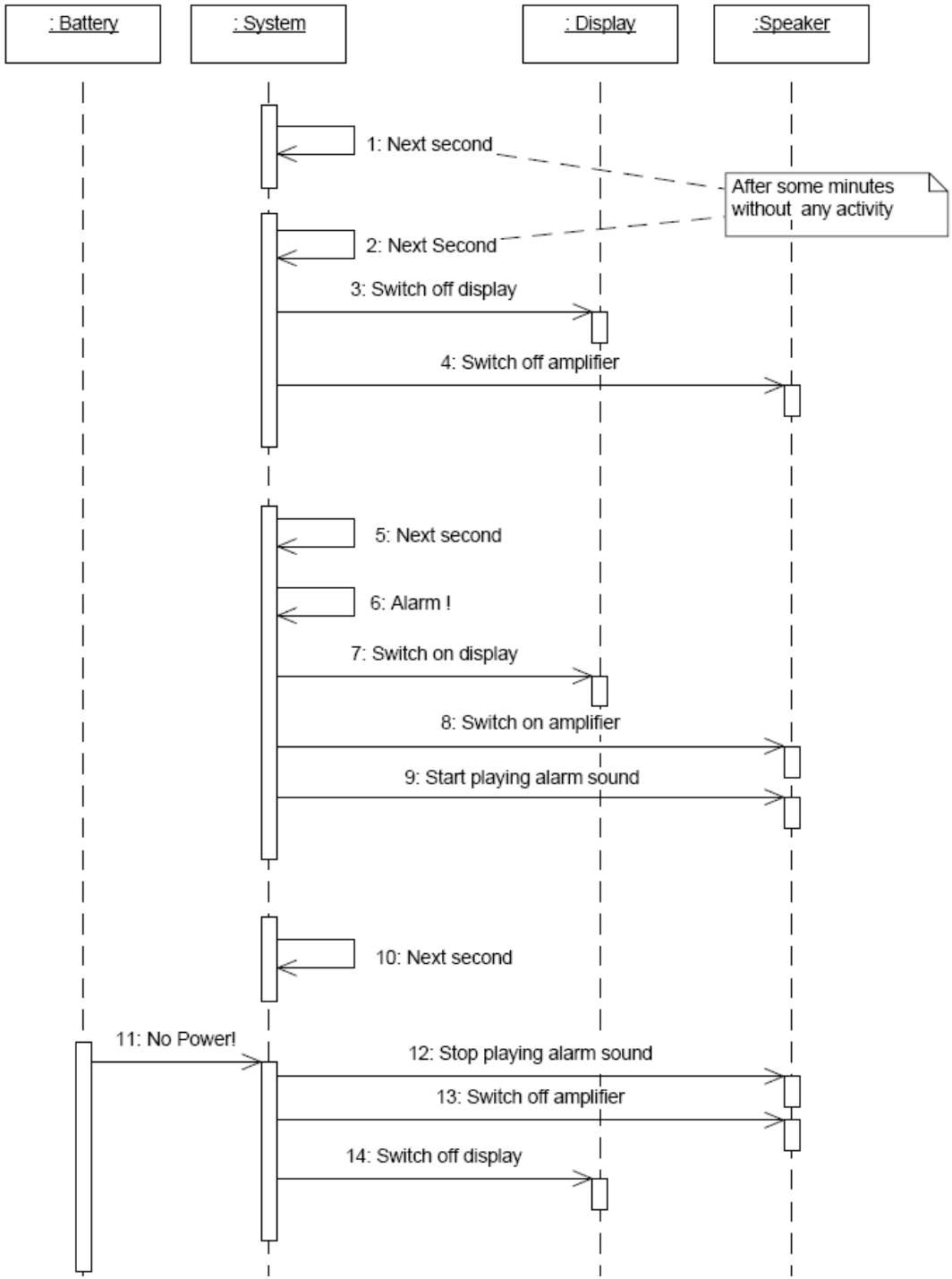


Scenario:
“Allarme della sveglia contemporaneamente alla riproduzione di un messaggio”

...


<< actor >>
User
: System
:Speaker
1: Play Message
2: Start playing sound
3: Display Progress Indicator
4: Next Second
5: Display Clock
6: Display Progress Indicator
7: Alarm!
{ 0.5 s}
8: Start playing alarm
9: Display Alarm Indicator
10: Stop
11: Stop playing alarm sound
13: Display Cock
12: Next Second

Scenario: “Entrare e uscire dalla modalità ‘stand-by’”



Fase di Progetto

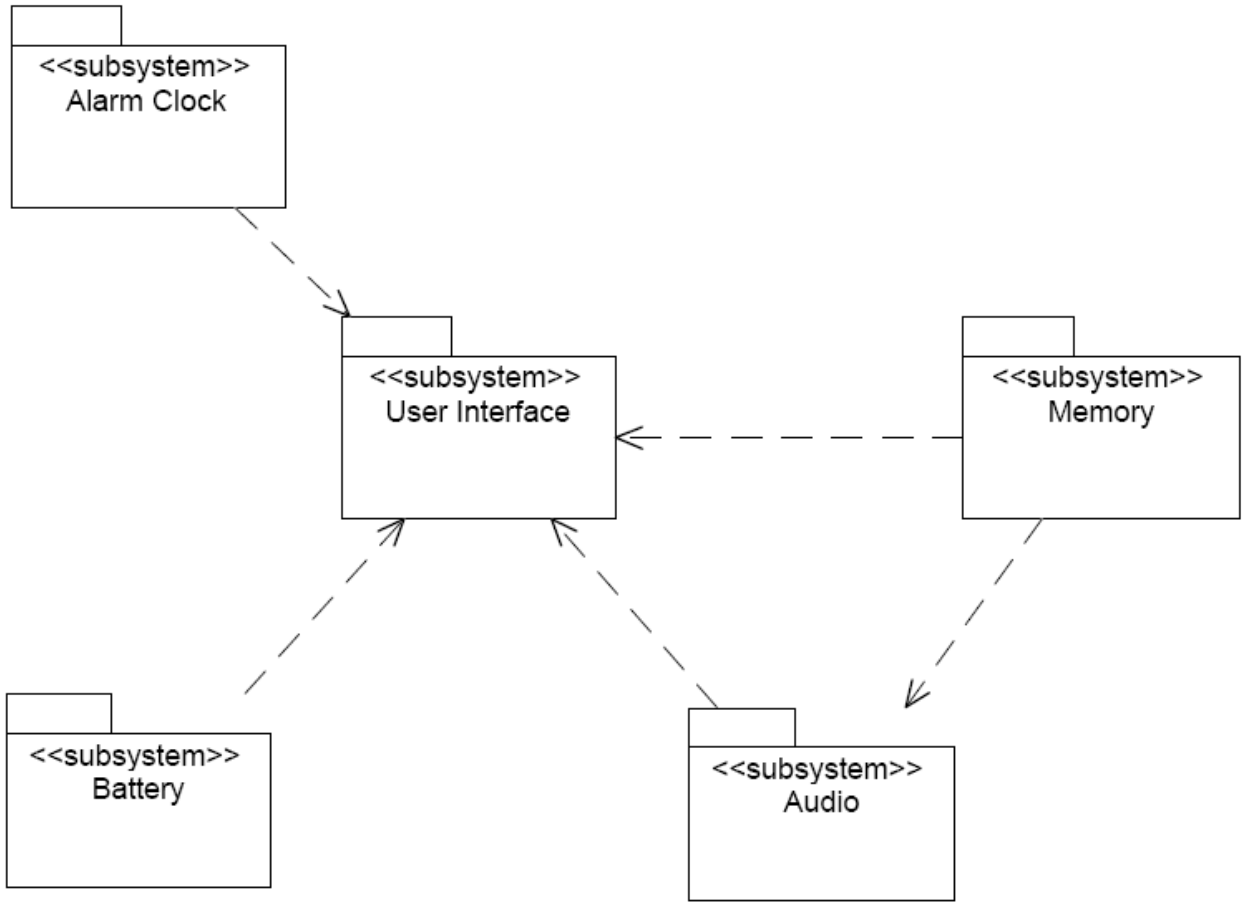
- Analizzati i requisiti, occorre capire quali elementi software possono essere utilizzati per realizzarli

- Approccio incrementale
  - Identificare i moduli software
  - Identificare le classi principali di ogni modulo
  - Raffinare tali classi generali per arrivare ad un vero e proprio design

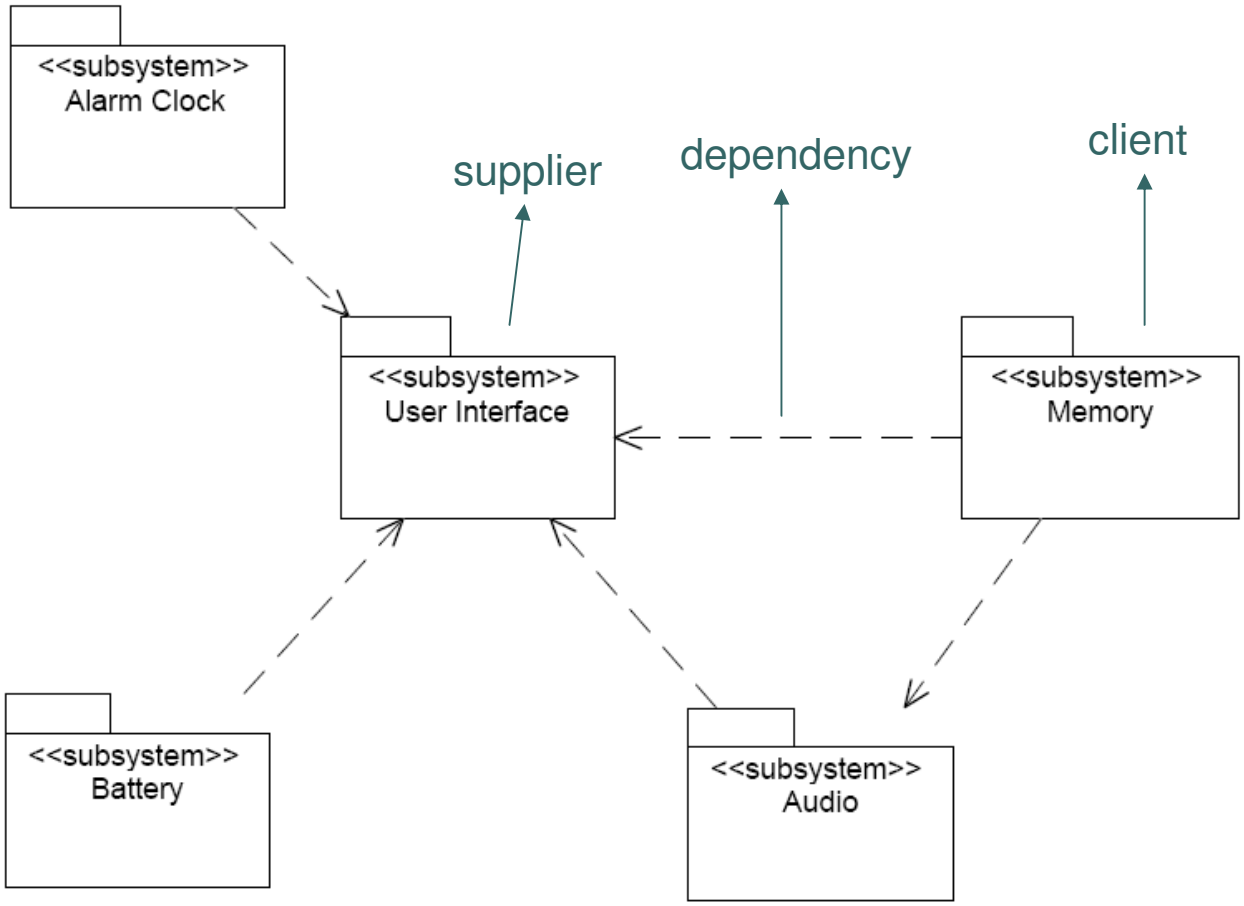
Identificazione dei Moduli SW

- Package diagrams
  Un *package* è un costrutto che permette di raggruppare elementi UML in unità di più alto livello rispetto agli elementi stessi
  - Caso più comune: raggruppare classi
  - Nel modello UML ciascuna classe è membro di un singolo package
    - Si può creare una gerarchia di package
  - I package corrispondono ai costrutti:
    - `package` (Java)
    - `namespace` (C++, C#)

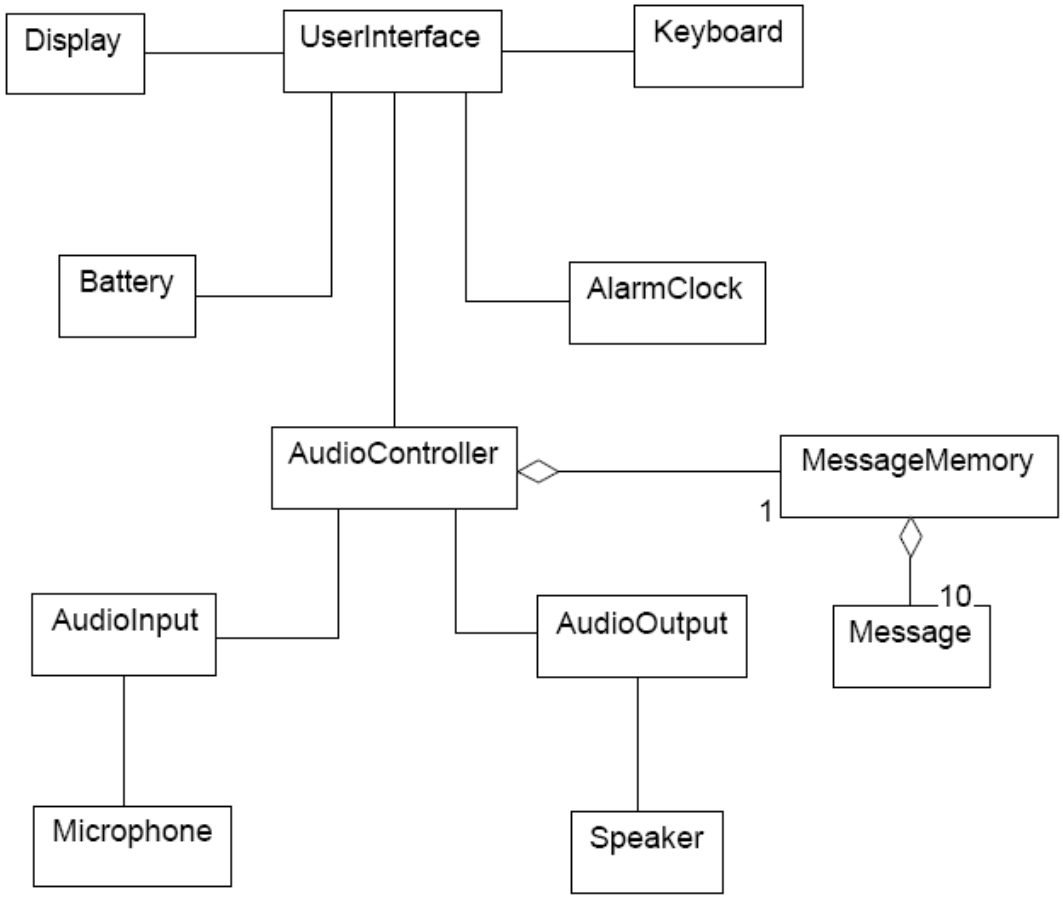
Package Diagram



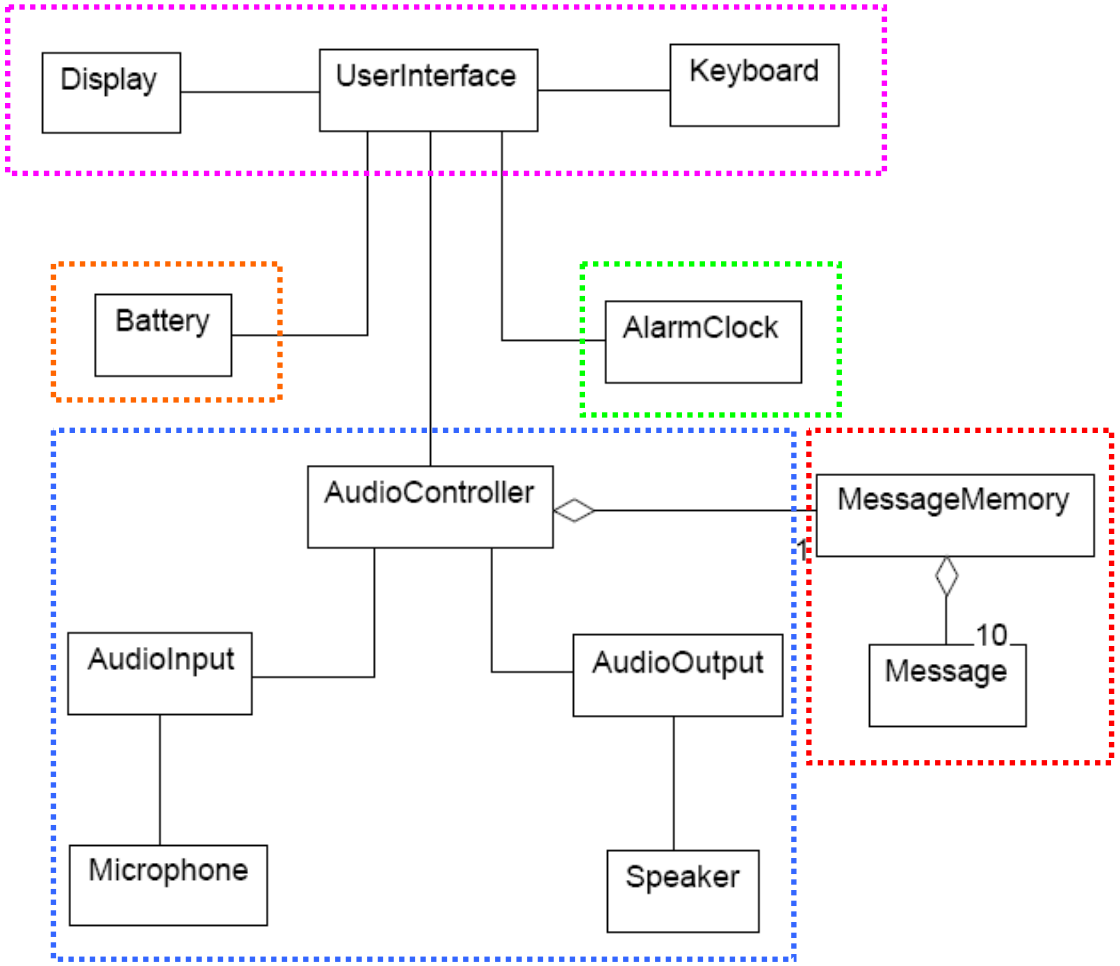
Package Diagram (ii)



Class Diagram (base)



Classi e Packages



Descrizione Classi

- **Display, Keyboard**
  elementi passivi
- **UserInterface**
  controlla l'interazione con l'utente
- **Microphone, Speaker**
  HW wrappers
- **AudioController**
  usa le classi **AudioInput** e **AudioOutput** per gestire i suoni

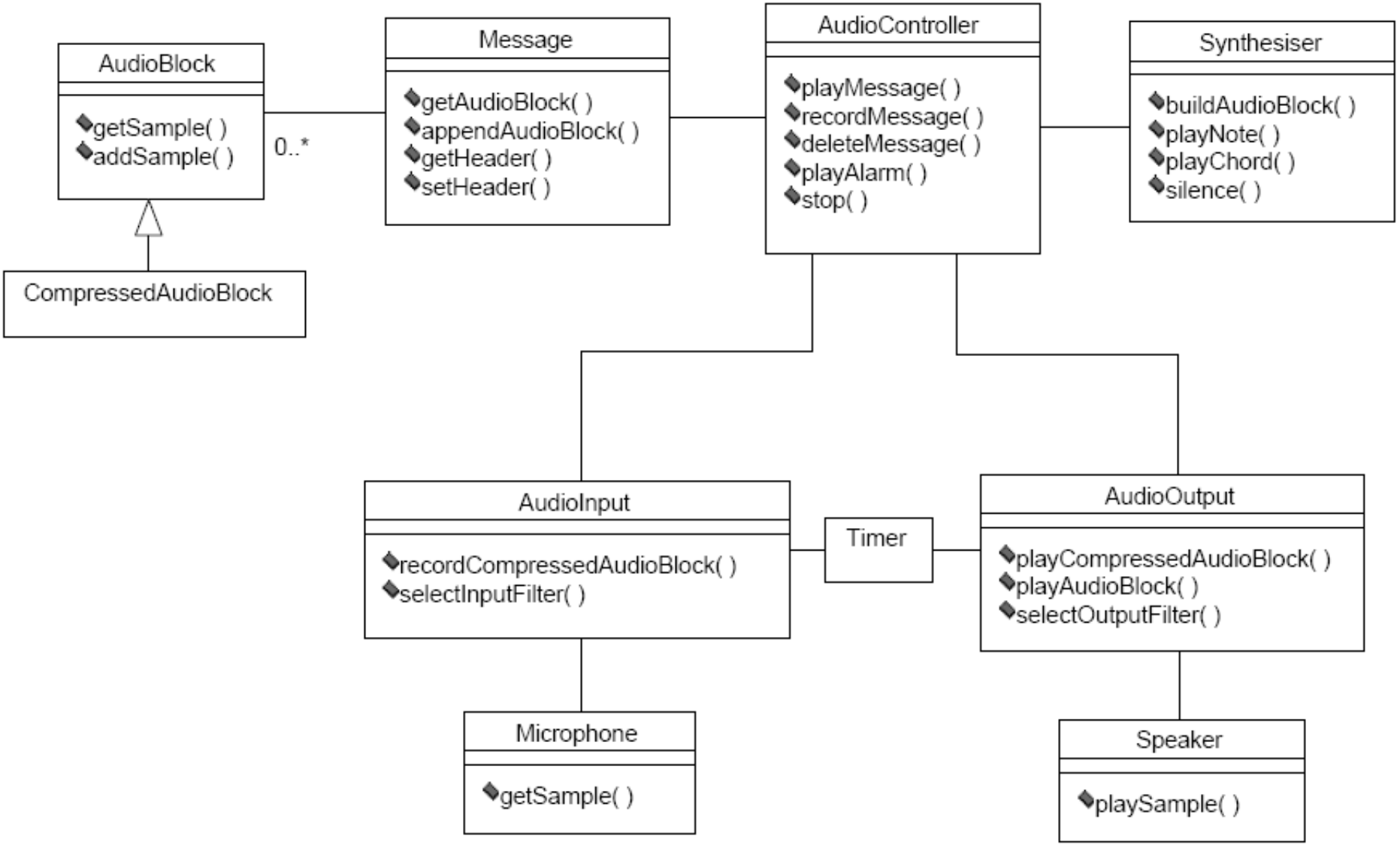
Descrizione Classi (ii)

- **AlarmClock**
  aggiorna l'orologio interno e controlla quando far scattare la sveglia (lo comunica a **UserInterface**)
- **Battery**
  misura periodicamente il livello di carica della batteria (quando è basso lo comunica a **UserInterface**)
- **Message**
  singolo messaggio
- **MessageMemory**
  gestisce la directory dei messaggi

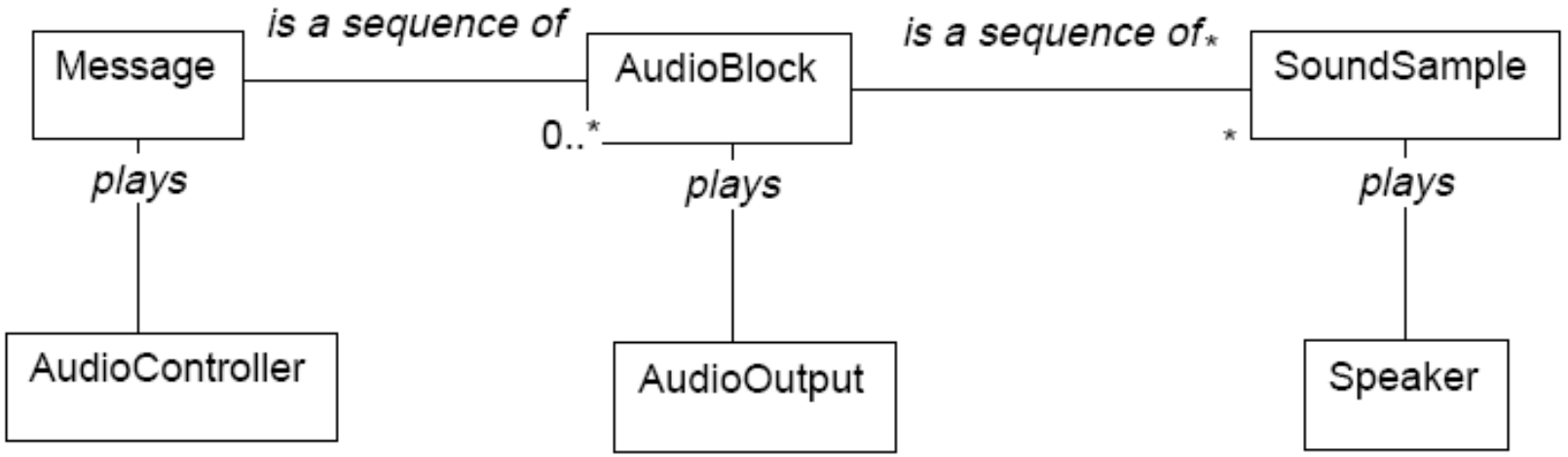
Audio Subsystem

- Ogni messaggio è composto da più blocchi (*blocks*) e ogni blocco è costituito da molti campioni audio (*sound samples*)
- Il sottosistema audio registra o riproduce sempre un blocco per volta
- **AudioInput** e **AudioOutput** hanno stringenti requisiti real-time -> classe **Timer**
  - **Timer**: wrapper per il timer hardware
- **Microphone** e **Speaker** sono in grado di lavorare su un sample per volta
- I blocchi sono compressi mentre vengono registrati

Audio Subsystem – Class Diagram

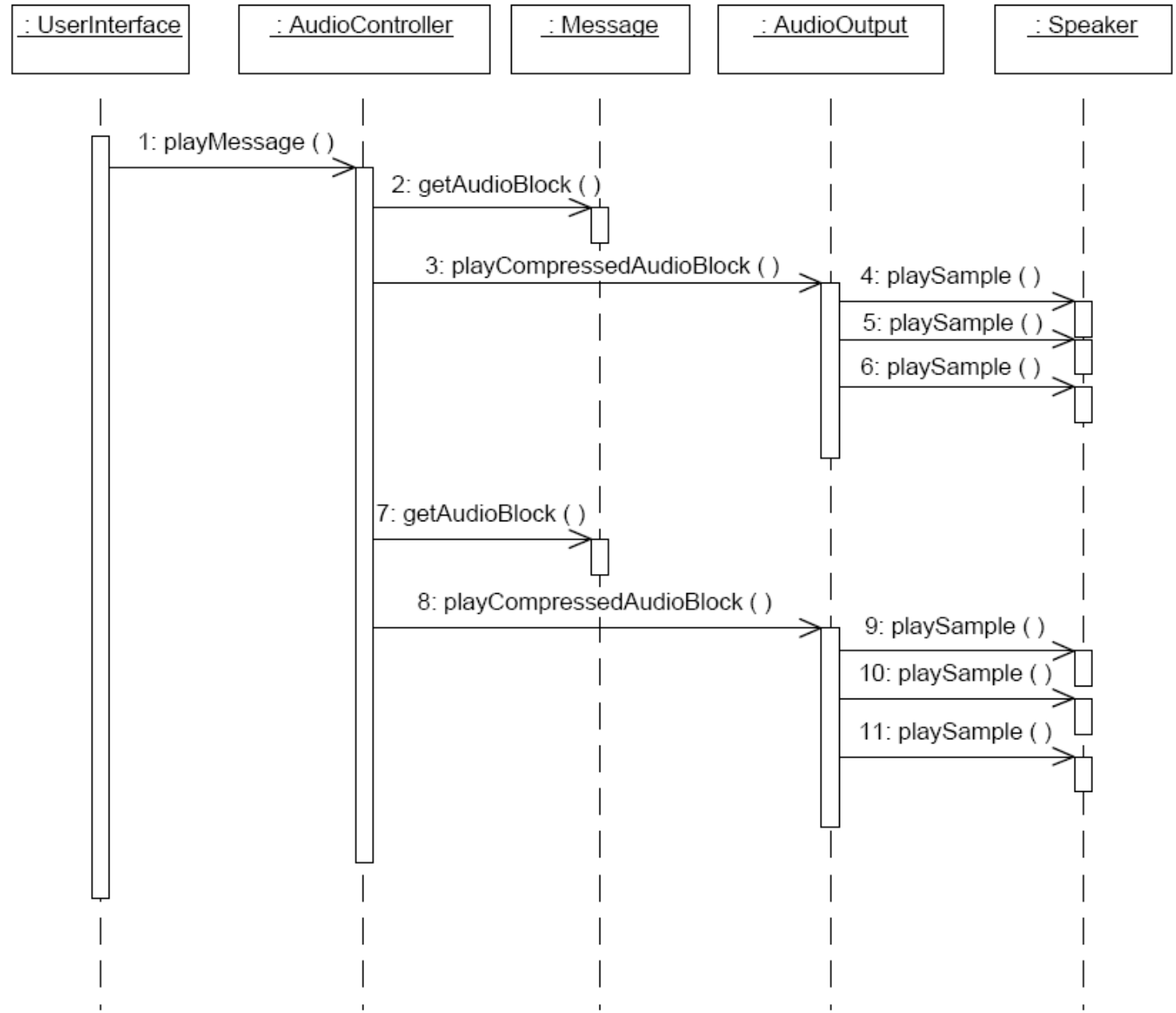


Audio Subsystem (ii)



Scenario: “Riproduzione di un messaggio”

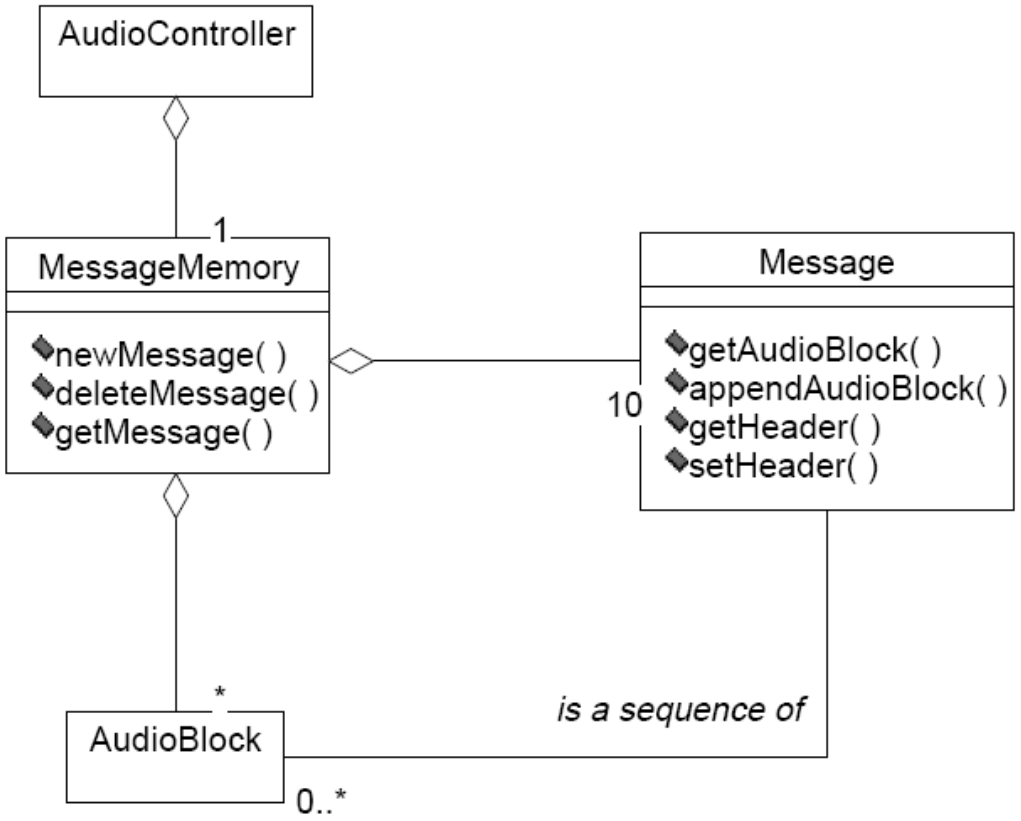
...


: UserInterface
: AudioController
: Message
: AudioOutput
: Speaker
1: playMessage ( )
2: getAudioBlock ( )
3: playCompressedAudioBlock ( )
4: playSample ( )
5: playSample ( )
6: playSample ( )
7: getAudioBlock ( )
8: playCompressedAudioBlock ( )
9: playSample ( )
10: playSample ( )
11: playSample ( )

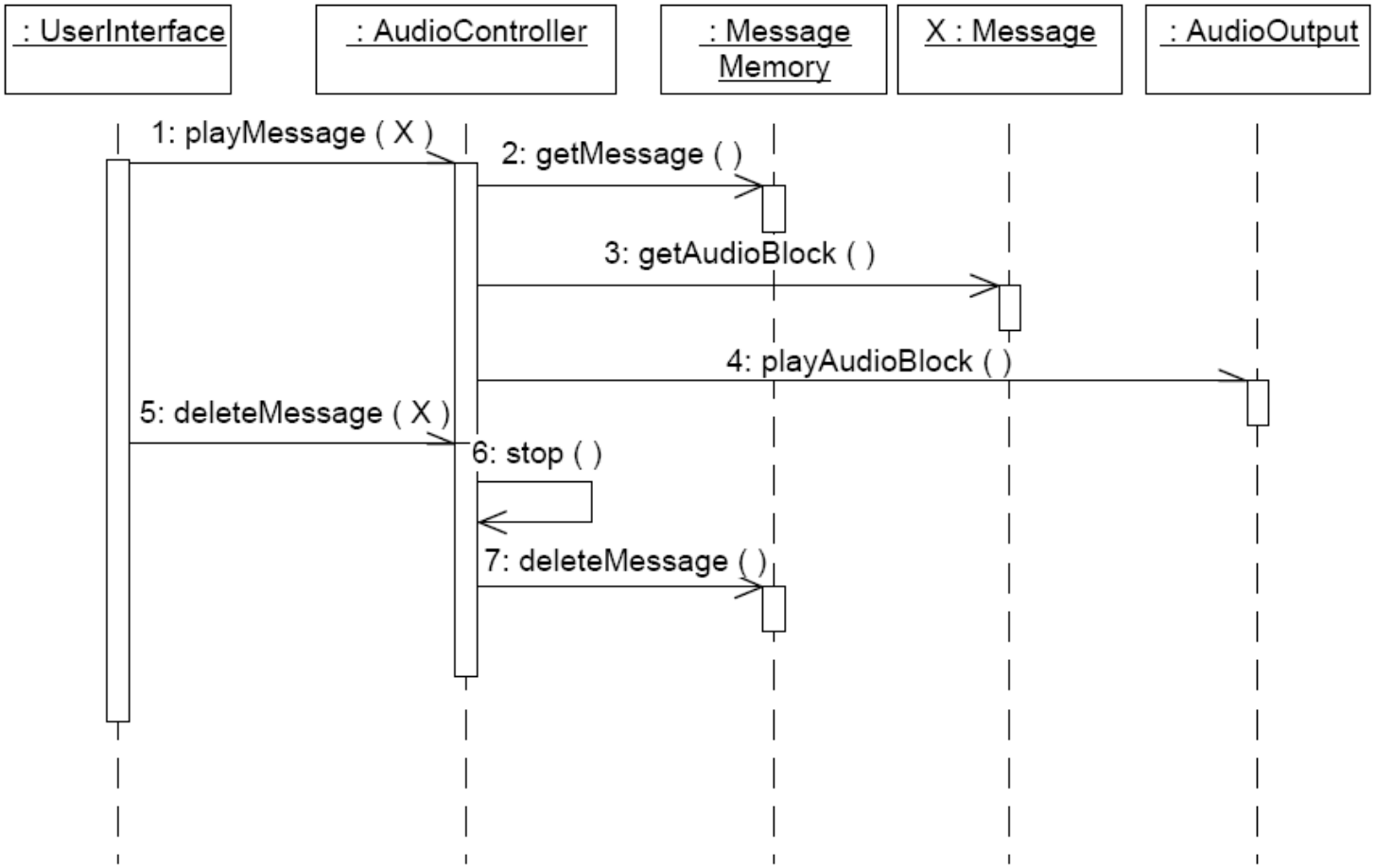
Memory Subsystem

- La classe MessageMemory gestisce la memoria non-volatile dell'apparecchio
- Mantiene una directory dei messaggi registrati e alloca spazio per nuovi messaggi
- UserInterface usa MessageMemory per ottenere la lista dei messaggi registrati, ma non può fare modifiche
  - AudioController è l'unica classe che può modificare MessageMemory
  - Se UserInterface vuole cancellare un messaggio deve invocare il metodo deleteMessage()
  - Così si evitano problemi nel caso in cui un messaggio venga cancellato mentre viene riprodotto o registrato (v. scenario)

Memory Subsystem: Class Diagram



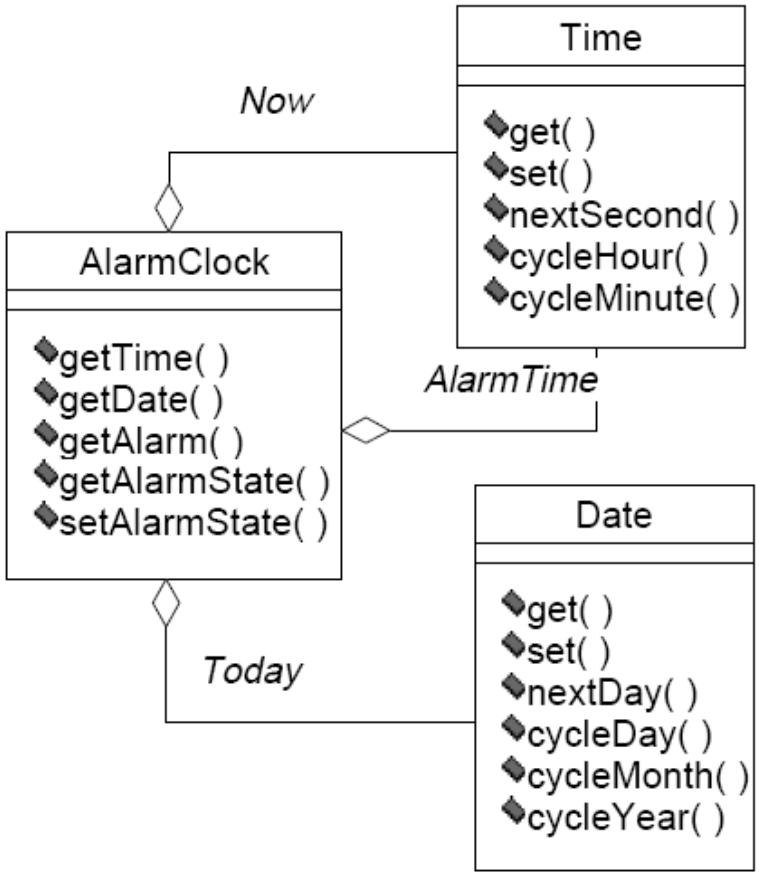
Scenario: “Cancellazione di un messaggio contemporaneamente alla sua riproduzione”



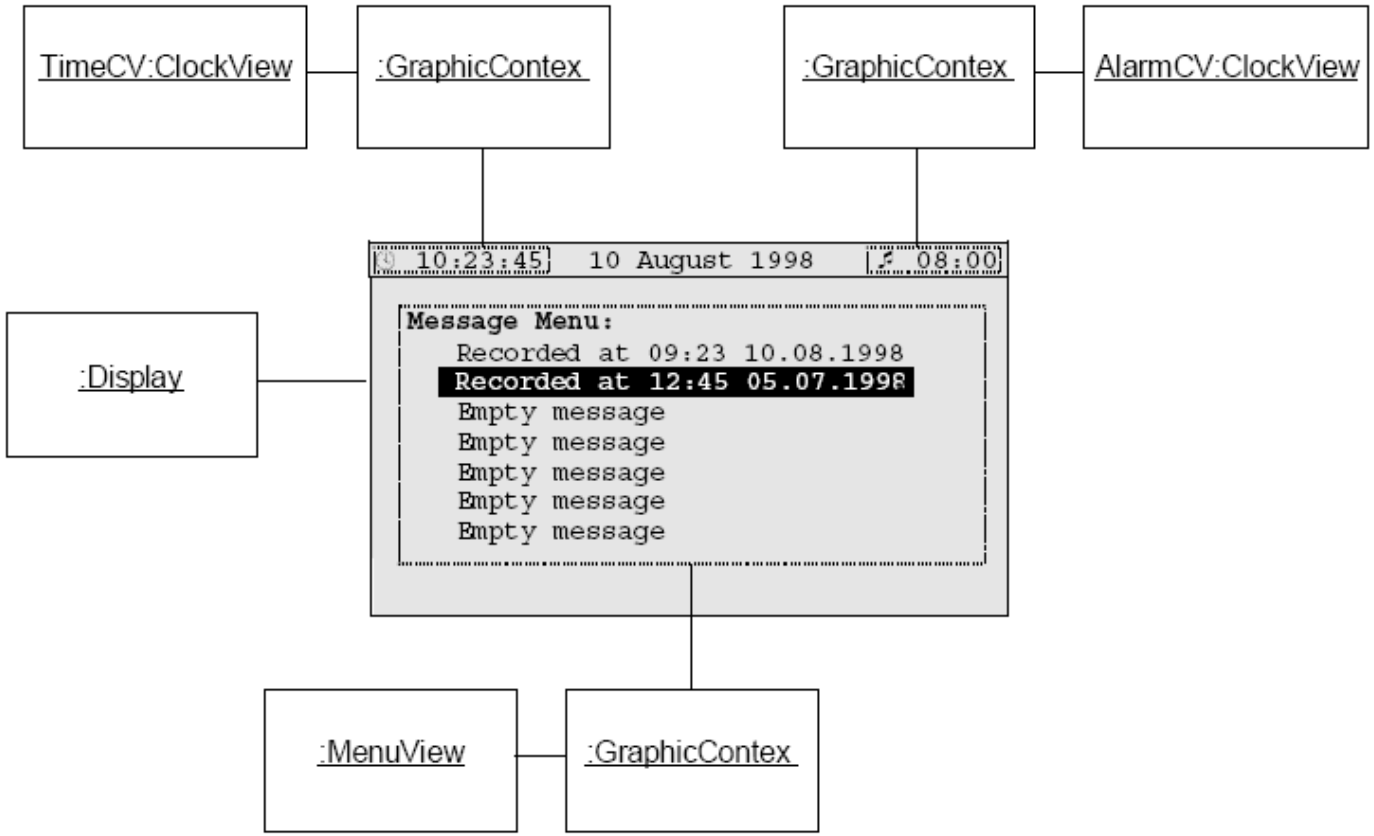
Alarm Clock Subsystem

- La classe **AlarmClock** mantiene l'ora corrente, la data e l'ora della sveglia
- **AlarmClock** usa un timer per misurare il passare del tempo
  - Ogni secondo aggiorna l'ora corrente usando il metodo nextSecond()
  - Quando l'oggetto **Time** fa *wrap-around* viene aggiornata la data corrente usando il metodo nextDay()

Alarm Clock Subsystem – Class Diagram



User Interface: Elementi Visuali



User Interface Subsystem

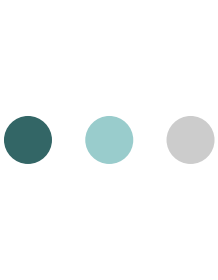
- La classe UserInterface riceve l'input dall'utente tramite la tastiera e ritorna i feedback attraverso il display
- **Display**: HW wrapper per il display; può essere disattivato per risparmiare energia
- L'astrazione **GraphicContext** è usata per "disegnare" il display
  - Fornisce primitive per disegnare punti, linee o stringhe di testo
  - Ogni **GraphicContext** è un'area rettangolare dello schermo
  - Gestisce la trasformazione geometrica da coordinate locali a quelle globali sul display

User Interface Subsystem (ii)

- Le classi **View** usano le primitive grafiche di **GraphicContext** per disegnare gli oggetti dell'applicazione
  - Ogni **View** usa un diverso **GraphicContext**
- **UserInterface** riceve i messaggi da **AlarmClock** e **Battery**
- Il sistema reagisce ad alcuni eventi, come la pressione di un tasto, in base al corrente *User Mode*
  - Per gestire il diverso comportamento dei tasti in base al contenuto attuale dello schermo
  - Per es., il tasto 'down' può voler dire:
    - Vai al messaggio sotto
    - Decrementa il numero dei minuti dell'ora

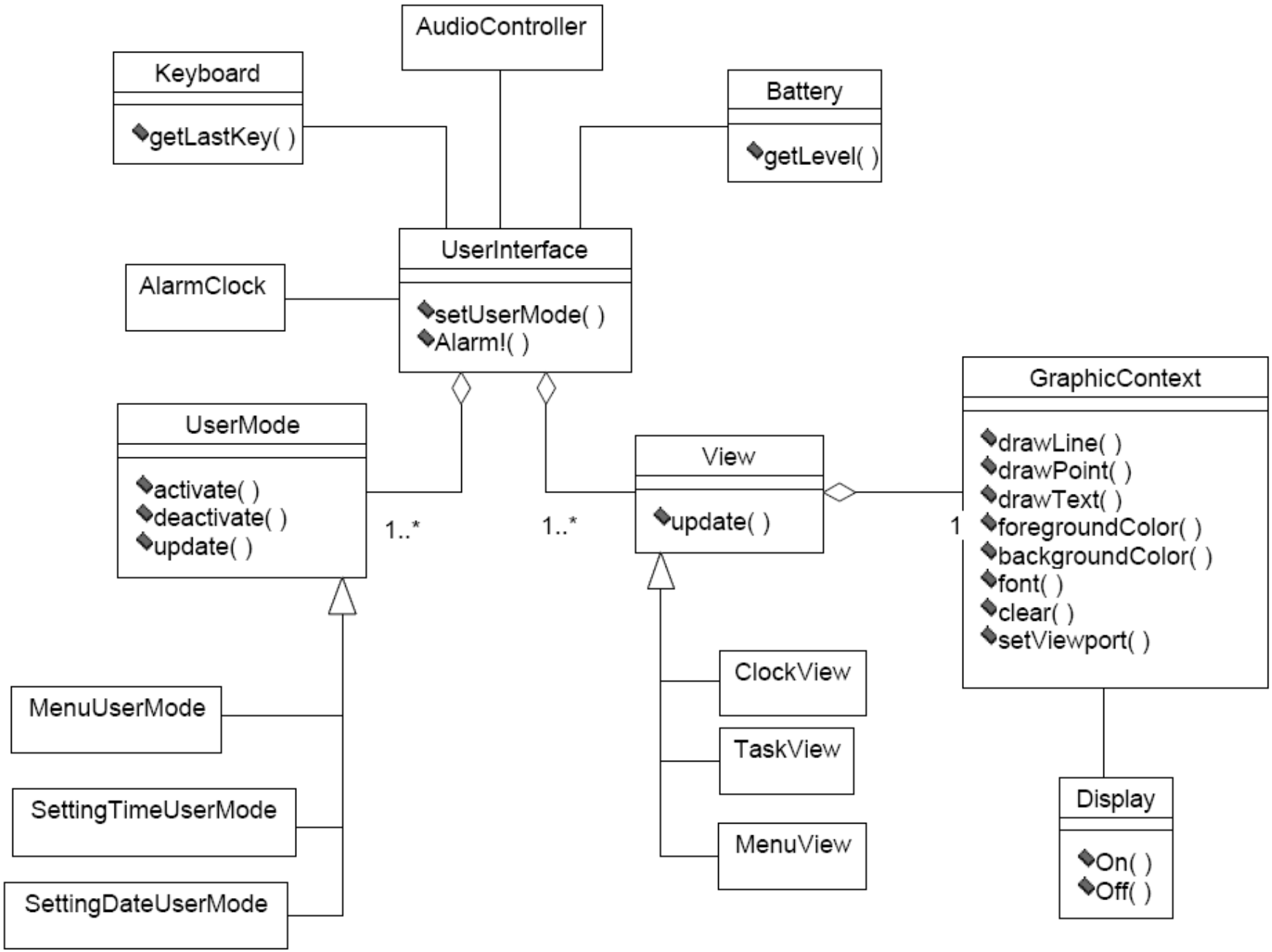
    a seconda del contesto

    Eventi come questo non vengono gestiti direttamente dall'handler **UserInterface** ma vengono inoltrati allo specifico handler in base al corrente User Mode



User Interface: Class Diagram

...


AudioController
Keyboard
getLastKey( )
Battery
getLevel( )
UserInterface
setUserMode( )
Alarm!( )
AlarmClock
GraphicContext
drawLine( )
drawPoint( )
drawText( )
foregroundColor( )
backgroundColor( )
font( )
clear( )
setViewport( )
UserMode
activate( )
deactivate( )
update( )
1..*
1..*
View
update( )
1
ClockView
TaskView
MenuView
MenuUserMode
SettingTimeUserMode
SettingDateUserMode
Display
On( )
Off( )

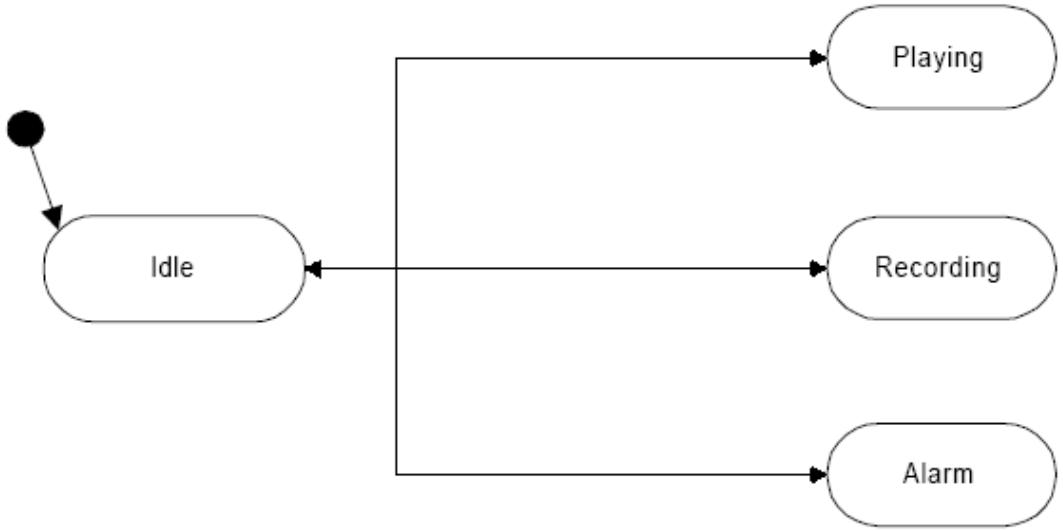
Definire il Comportamento degli Oggetti

- Statechart Diagrams

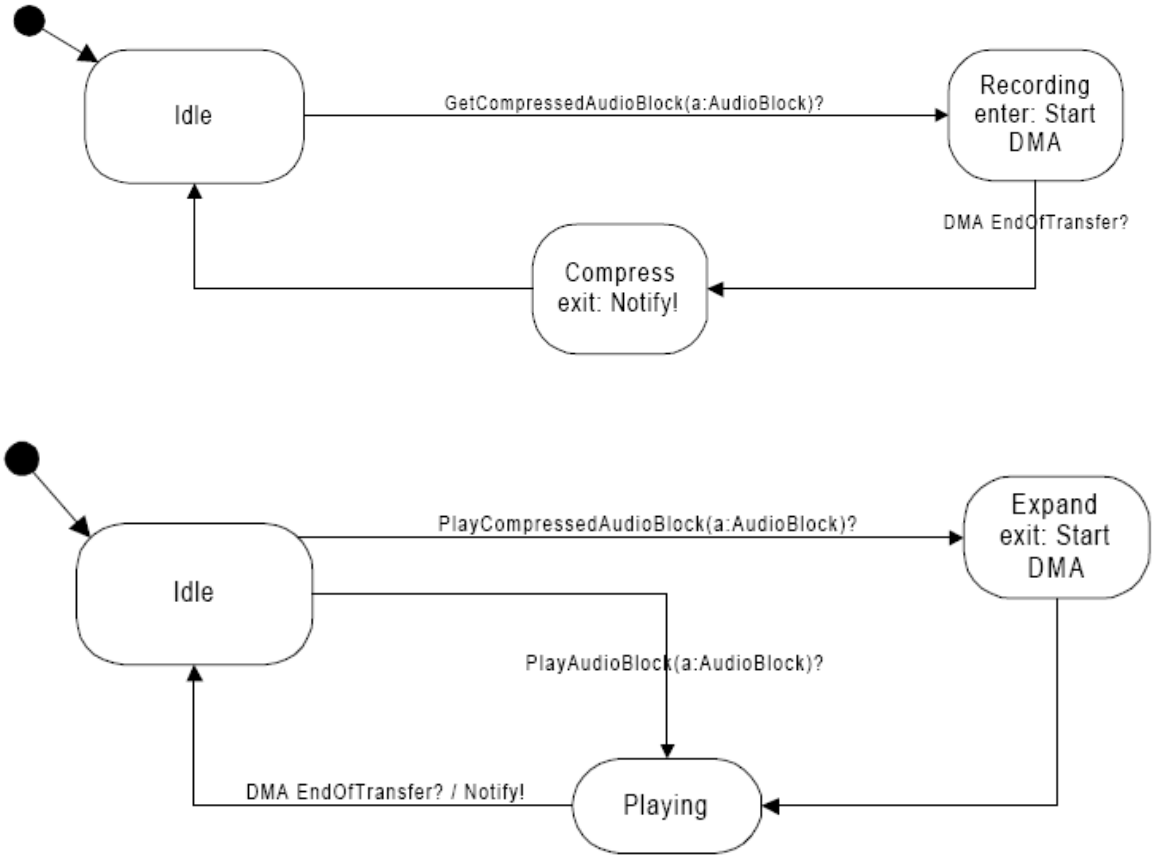
- [Collaboration Diagrams]
- [Message Sequence Diagram]

[non li vedremo]

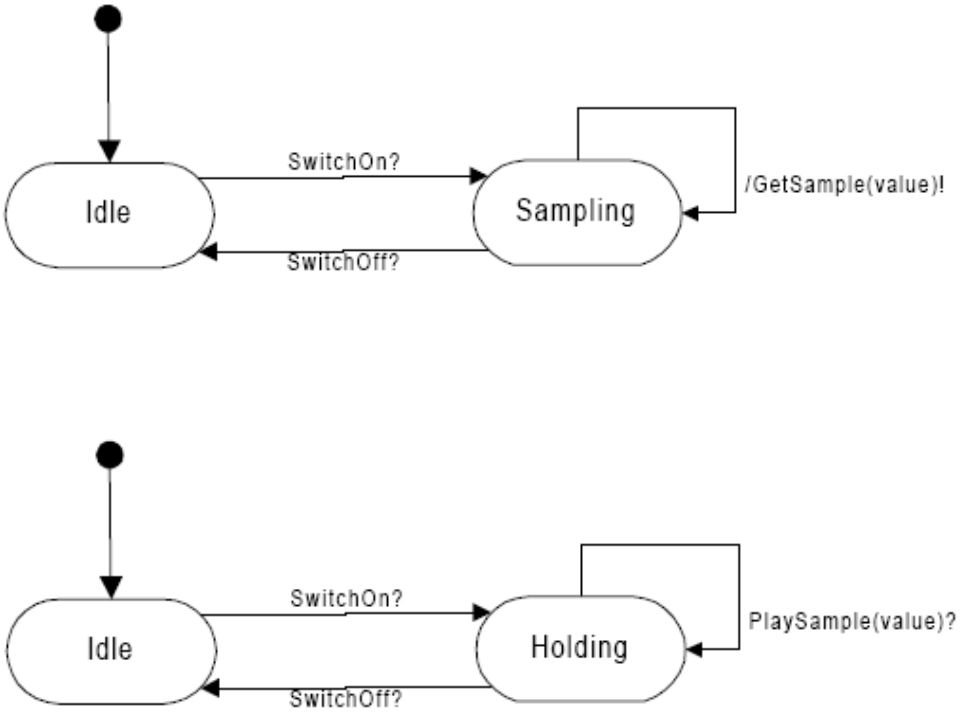
AudioController



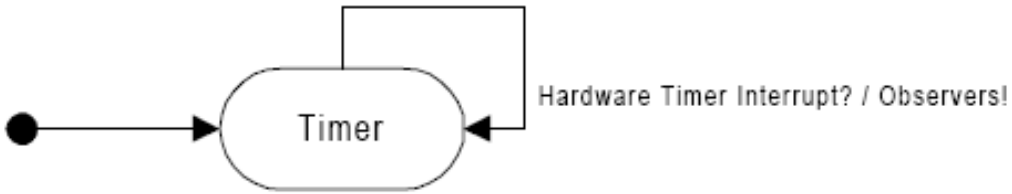
AudioInput e AudioOutput

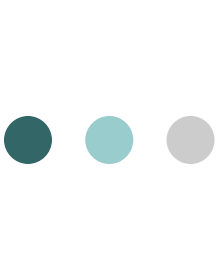


Microphone, Speaker



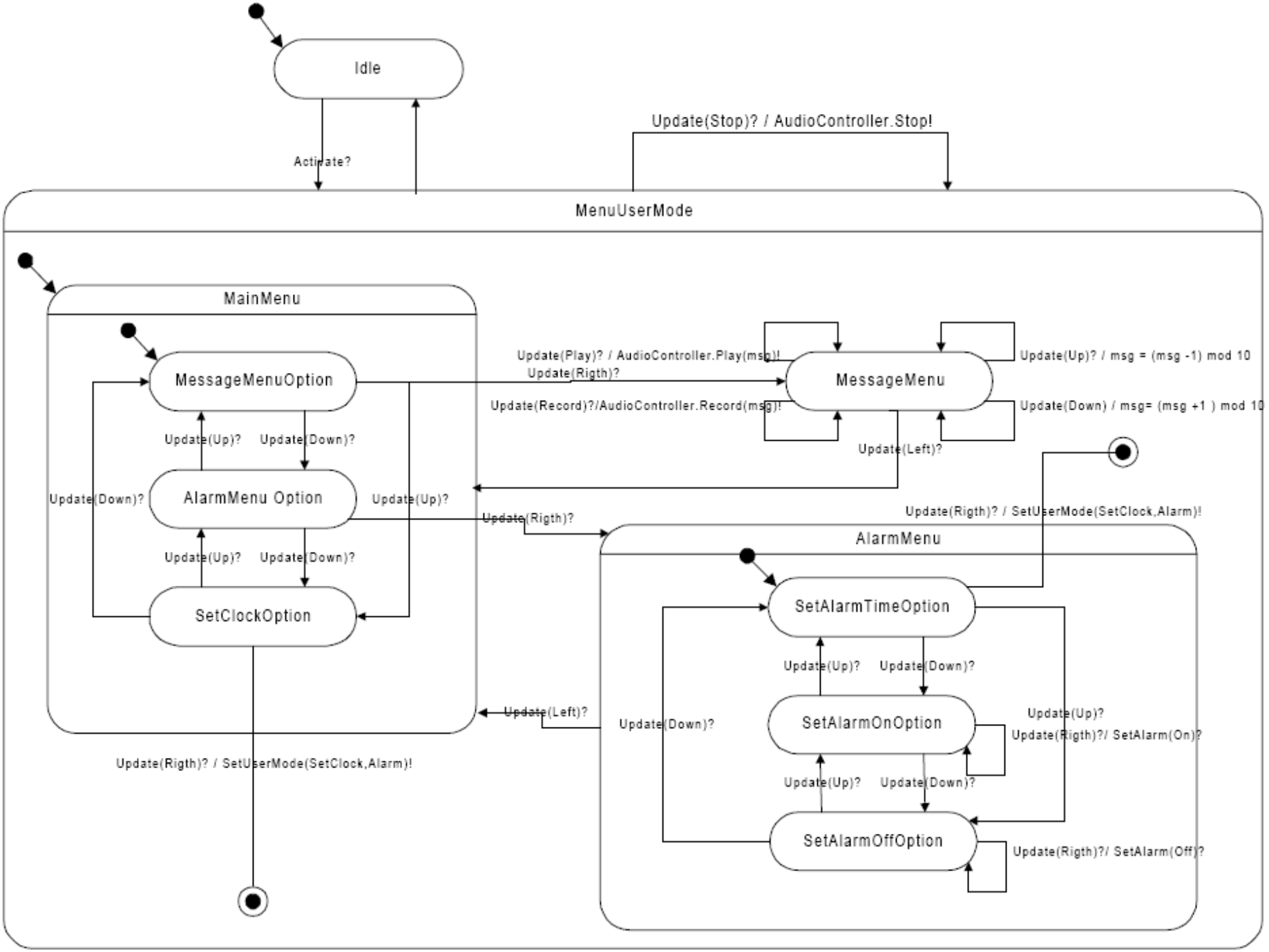
Timer



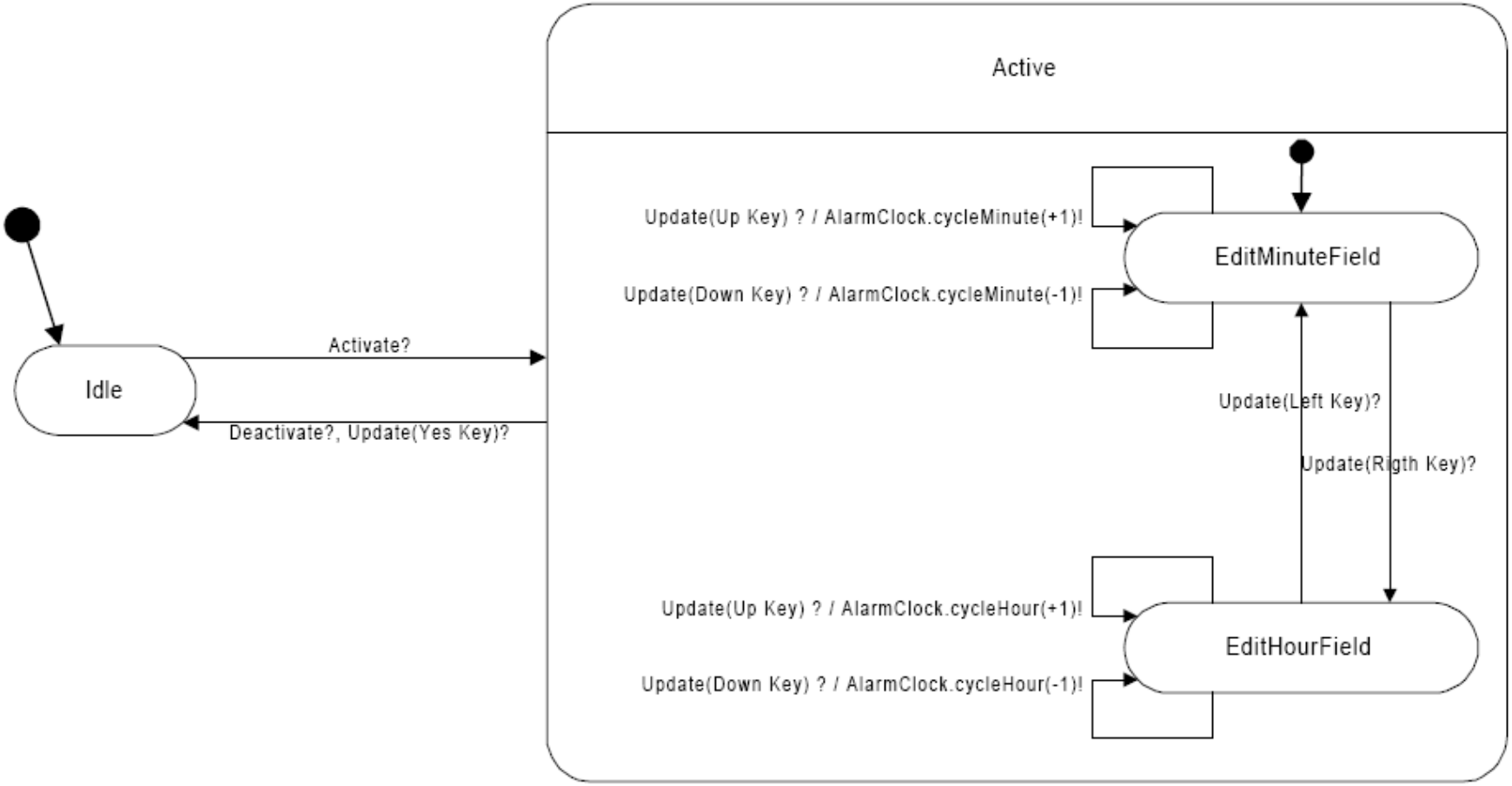


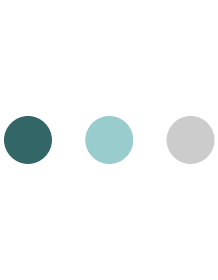
MenuUserMode

...


Idle
Activate?
Update(Stop)? / AudioController.Stop!
MenuUserMode
MainMenu
MessageMenuOption
Update(Up)?
Update(Down)?
AlarmMenu Option
Update(Down)?
Update(Up)?
Update(Up)?
Update(Down)?
SetClockOption
Update(Play)? / AudioController.Play(msg)!
Update(Rigth)?
MessageMenu
Update(Up)? / msg = (msg -1) mod 10
Update(Record)?/AudioController.Record(msg)!
Update(Down) / msg= (msg +1) mod 10
Update(Left)?
Update(Rigth)?
Update(Rigth)? / SetUserMode(SetClock,Alarm)!
AlarmMenu
SetAlarmTimeOption
Update(Up)?
Update(Down)?
Update(Left)?
Update(Down)?
SetAlarmOnOption
Update(Up)?
Update(Rigth)?/ SetAlarm(On)?
Update(Up)?
Update(Down)?
SetAlarmOffOption
Update(Rigth)?/ SetAlarm(Off)?
Update(Rigth)? / SetUserMode(SetClock,Alarm)!

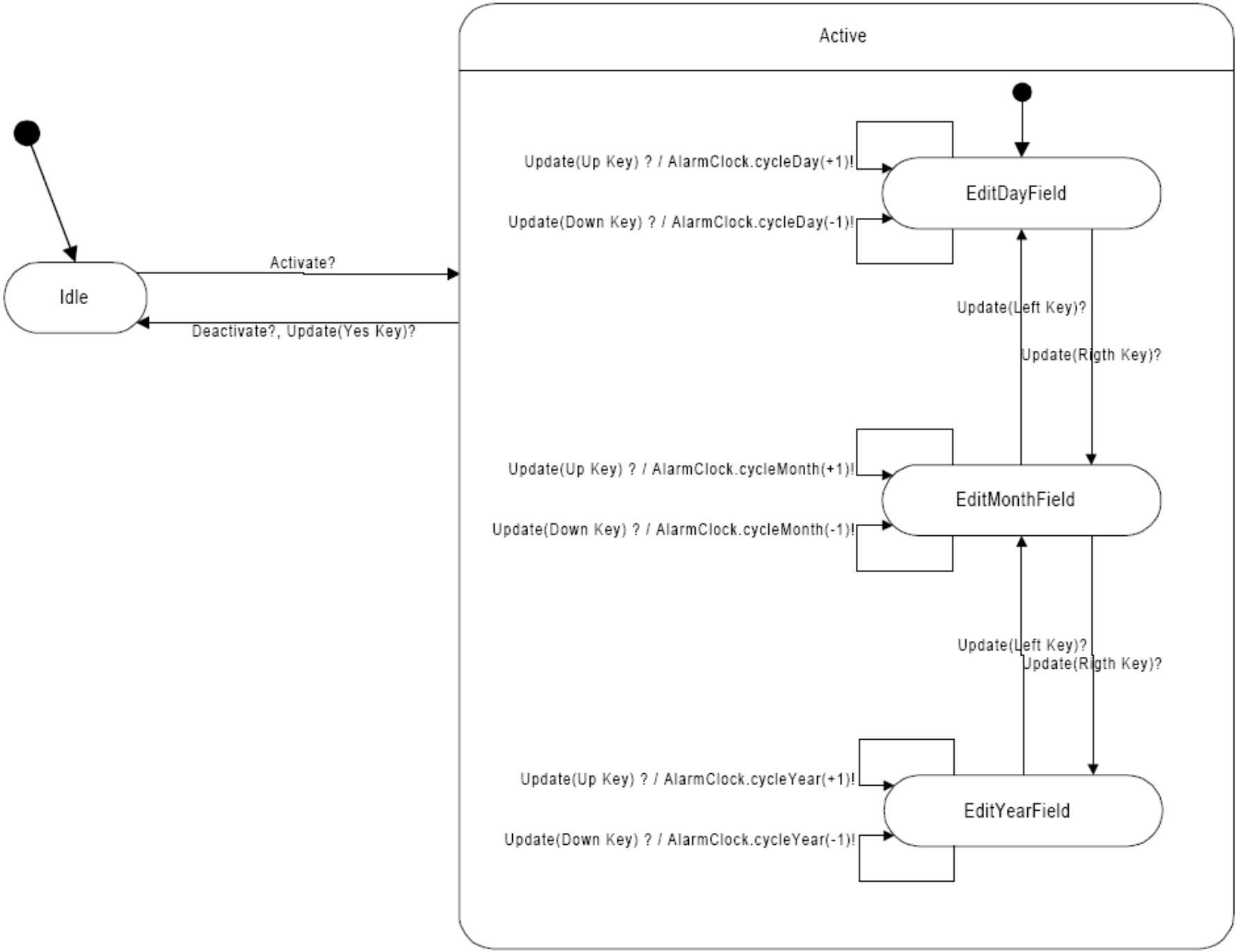
SettingTimeUserMode





SettingDateUserMode

...


Active
Idle
Activate?
Deactivate?, Update(Yes Key)?
Update(Up Key) ? / AlarmClock.cycleDay(+1)!
Update(Down Key) ? / AlarmClock.cycleDay(-1)!
EditDayField
Update(Left Key)?
Update(Rigth Key)?
Update(Up Key) ? / AlarmClock.cycleMonth(+1)!
Update(Down Key) ? / AlarmClock.cycleMonth(-1)!
EditMonthField
Update(Left Key)?
Update(Rigth Key)?
Update(Up Key) ? / AlarmClock.cycleYear(+1)!
Update(Down Key) ? / AlarmClock.cycleYear(-1)!
EditYearField

Bibliografia

[Paltor99] I. P. Paltor e J. Lilius, “Digital Sound Recorder: A case study on designing embedded systems using the UML notation”
http://www.tucs.abo.fi/publications/techreports/TR234.pdf

[Lami07] G. Lami, “Requirements Engineering: Natural Language Requirements Elicitation, Specification and Quality Evaluation”, presentazione per il Corso di Dottorato in Ingegneria dell’Informazione