Embedded System Design (ESD)

ESD is about the implementation of a set of functionalities satisfying a number of constraints ranging from performance to cost, emissions, power consumption and weight.

Embedded System Design : Some Issues

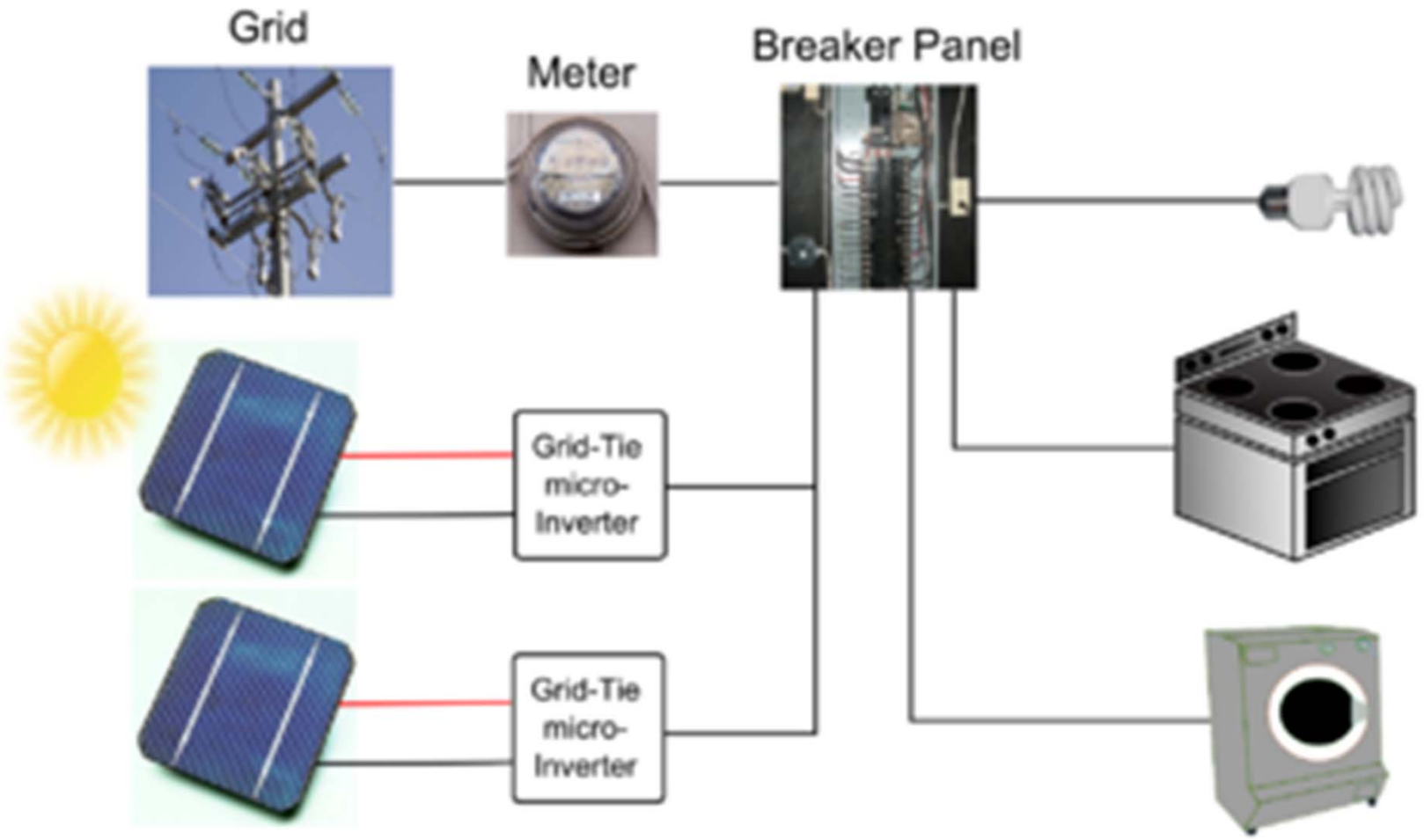
The functionalities to be implemented in ES have grown in number and complexity so much that the development time is increasingly difficult to predict and to keep in check.

The complexity increase coupled with the constantly evolving specifications has forced designers to look at implementations that are intrinsically flexible, i.e., that can be changed rapidly.

Since hardware-manufacturing cycles do take time and are expensive, the interest in software-based implementation has risen to previously unseen levels.

In software design, little attention has been traditionally paid to hard constraints on reaction speed, memory footprint and power consumption of software.

Grid Connected PV System



Solar Power plant health monitoring system to monitor

1. Irradiance
2 Ambient temperature
3 Module temperature
4 Energy generated
5 Wind speed
6 Cracks on the solar panel
7 Fire on the solar panel
8 Dust on the solar panel
9 Faulty solar cells or modules
10 Inverter health indicators
11 Grid equipment health indicators
12 Uptime and Efficiency

Need a sensor box which can be accessed from the lab or anywhere.
All sensors will be connected to the sensor box.

Important trends are emerging for the design of embedded systems

a) The use of highly programmable platforms

b) The use of the C Language for embedded software development.

c) The use of Java Language for embedded software development.

d) The use of the Unified Modeling Language (UML) for embedded software development.

Would this lead to a unified embedded system development methodology?

Would this combination magnify the effective gains in productivity and implementation?

What is a product of a development team?

- Not a set of beautiful documents
- Not a few world class meetings
- Not great slogans
- Not prize winning codes

- A software/product which satisfies the evolving needs of its users and business.

- A solution which has a solid architectural foundation that is resilient to change

- A solution which involves minimum of software scrap and rework

- A solution based on a good model to manage risk and communicate the desired structure and behavior of the system.

What is a model and Why we need it?

Model is a simplification of reality and provides a blueprint of the system.

- We build models so that we and others can better understand the system we are developing easily

  - Models help to visualize a system as we want it to be
  - Models permit us to specify the structure and behavior of a system
  - Models give us a template that guides us in constructing a system.
  - Models document the decisions we have made.

- We build models of complex systems because we cannot comprehend such a system in its entirety.

Four Principles of Modeling

First : The choice of what models to create has a profound influence on how a problem is attacked and how a solution is shaped. Example: Geocentric Vs. Heliocentric system

Second: Every model may be expressed at different levels of precision

Best kind of models are those which let you choose your degree of detail depending on who is doing the viewing and what they need to view.

Four Principles of Modeling

Third: The best models are connected to reality.

Mathematical model of an aircraft should tell us about real aircraft.

Fourth: No single model is sufficient. Every nontrivial system is best approached through a small set of nearly independent models.

Example : floor plan, elevations, electrical and heating plans and plumbing plans for a building.

Embedded System Design Methods

Algorithm Oriented Programming
Vs
Object Oriented Programming

Algorithmic Vs. Object Oriented view

Algorithmic perspective: Main building block of all software is the procedure or function. Developers focus on issues of control and decomposition of larger algorithms to smaller ones.

- Leads to brittle systems/software.
- As the requirements change(and they will) and the system grows(and it will) systems built with algorithmic focus turn out to be very hard to maintain.

Object oriented perspective: Main building block of all software systems is the object or class. Simply put, an object is a thing and class is a description of a set of common objects.

UML is an object oriented language for visualizing, specifying, constructing and documenting.

About UML

UML is an object-oriented modeling language standardized by the Object Management Group (OMG) mainly for software systems development.

It consists of a set of basic building blocks, rules that dictate the use and composition of these building blocks, and common mechanisms that enhance the quality of the UML models

Its rich notation has made UML a popular modelling language in multiple application domains for system documentation and specification, for capturing user requirements and defining initial software architecture.

The use of UML with code generation tools is still quite limited

UML in Embedded System Design

UML is capturing much attention in the ESW community as a possible solution for,

1. Raising the level of abstraction to a level where productivity can be improved,
2. Errors can be easier to identify and correct,
3. Better documentation can be provided, and
4. ESW designers can collaborate more effectively.

An essential deficiency is that UML standardizes the syntax and semantics of diagrams, but not necessarily the detailed semantics of implementations of the functionality and structure of the diagrams in software.

Object Oriented Design and Analysis

1980: Object-oriented design methods : Grady Booch published a paper with a prophetic title: *Object-Oriented Design*.

Booch was able to extend his ideas to a genuinely object-oriented design method by 1991 in his book with the same title.

1990: Object-oriented analysis methods

The UML is applicable to anyone involved in the production, deployment, and maintenance of software.

Benefits of Object Oriented Design

The benefits of object-oriented analysis and design specifically include:

- required changes are localized and unexpected interactions with other program modules are unlikely;
- inheritance and polymorphism make OO systems more extensible, contributing thus to more rapid development;
- object-based design is suitable for distributed, parallel or sequential implementation;
- objects correspond more closely to the entities in the conceptual worlds of the designer and user, leading to greater seamlessness and traceability;
- shared data areas are encapsulated, reducing the possibility of unexpected modifications or other update anomalies.

Benefits of Object Oriented Design Composition and Inheritance

“With *composition* (aka *aggregation*), you define a new class, which is composed of existing classes. With *inheritance*, you derive a new class based on an existing class, with modifications or extensions.”

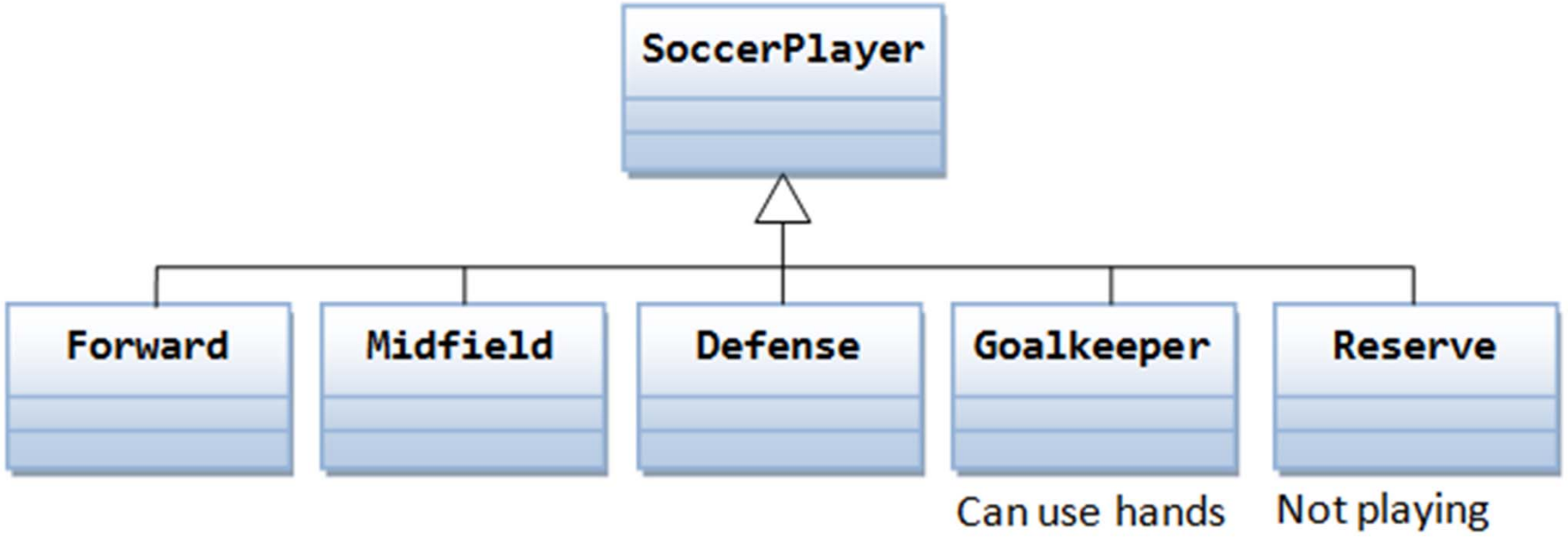
http://www.ntu.edu.sg/home/ehchua/programming/java/J3b_OOPInheritancePolymorphism.html

Benefits of Object Oriented Design
A Little more on Inheritance

"In OOP, we often organize classes in *hierarchy* to *avoid duplication and reduce redundancy*. The classes in the lower hierarchy inherit all the variables (static attributes) and methods (dynamic behaviors) from the higher hierarchies. A class in the lower hierarchy is called a *subclass* (or *derived*, *child*, *extended class*). A class in the upper hierarchy is called a *superclass* (or *base*, *parent class*). By pulling out all the common variables and methods into the superclasses, and leave the specialized variables and methods in the subclasses, *redundancy* can be greatly reduced or eliminated as these common variables and methods do not need to be repeated in all the subclasses."

http://www.ntu.edu.sg/home/ehchua/programming/java/J3b_OOPInheritancePolymorphism.html

Benefits of Object Oriented Design
A Little more on Inheritance



http://www.ntu.edu.sg/home/ehchua/programming/java/J3b_OOPInheritancePolymorphism.html

Benefits of Object Oriented Design Polymorphism

“The word "*polymorphism*" means "*many forms*". It comes from Greek word "*poly*" (means *many*) and "*morphos*" (means *form*). For examples, in chemistry, carbon exhibits polymorphism because it can be found in more than one form: graphite and diamond. Each of the form has it own distinct properties.”

http://www.ntu.edu.sg/home/ehchua/programming/java/J3b_OOPInheritancePolymorphism.html

Benefits of Object Oriented Design Polymorphism:Substitutability

“A subclass possesses all the attributes and operations of its superclass (because a subclass inherited all attributes and operations from its superclass). This means that a subclass object can do whatever its superclass can do. As a result, we can *substitute* a subclass instance when a superclass instance is expected, and everything shall work fine. This is called *substitutability*.”

http://www.ntu.edu.sg/home/ehchua/programming/java/J3b_OOPInheritancePolymorphism.html

Benefits of Object Oriented Design Polymorphism

“A subclass instance processes all the attributes operations of its superclass. When a superclass instance is expected, it can be substituted by a subclass instance. In other words, a reference to a class may hold an instance of that class or an instance of one of its subclasses - it is called substitutability.

If a subclass instance is assign to a superclass reference, you can invoke the methods defined in the superclass only. You cannot invoke methods defined in the subclass.

However, the substituted instance retains its own identity in terms of overridden methods and hiding variables. If the subclass overrides methods in the superclass, the subclass's version will be executed, instead of the superclass's version.”

http://www.ntu.edu.sg/home/ehchua/programming/java/J3b_OOPInheritancePolymorphism.html

Basic Steps in Object Oriented Design and Analysis Methods

- find the ways that the system interacts with its environment (use cases);
- identify objects and their attribute and method names;
- establish the relationships between objects;
- establish the interface(s) of each object and exception handling;
- implement and test the objects;
- assemble and test systems.

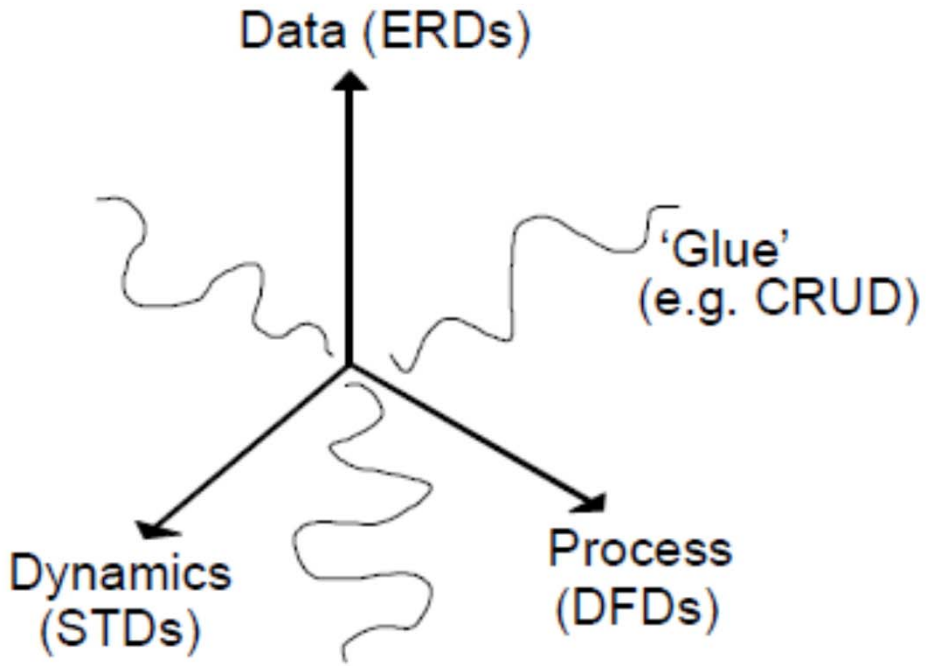


Figure 2 Three dimensions of software engineering.

ERDS : Entity Relationship Diagrams
STDs : State Transition Diagrams
DFDs : Data Flow Diagrams
CRUD: Create, Read, Update, Delete

Building Blocks of UML : Diagrams

•Diagrams group interesting collections of things

- Class diagram
  - Object diagram
  - Use case diagram
- Sequence diagram
- Collaboration diagram
- State chart diagram
- Activity diagram
- Component diagram
-Deployment diagram

The UML is a Language for Visualizing

For many programmers, the distance between thinking of an implementation and then pounding it out in code is close to zero.

You think it, you code it. In fact, some things are best cast directly in code. Text is a wonderfully minimal and direct way to write expressions and algorithms.

In such cases, the programmer is still doing some modeling, albeit entirely mentally. He or she may even sketch out a few ideas on a white board or on a napkin.

However, there are several problems with this. First, communicating those conceptual models to others is error-prone unless everyone involved speaks the same language.

The UML is a Language for Visualizing

Second, there are some things about a software system you can't understand unless you build models that transcend the textual programming language. For example, the meaning of a class hierarchy can be inferred, but not directly grasped, by staring at the code for all the classes in the hierarchy

Third, if the developer who cut the code never wrote down the models that are in his or her head, that information would be lost forever or, at best, only partially re-creatable from the implementation, once that developer moved on.

The UML is a Language for Specifying

In this context, specifying means building models that are precise, unambiguous, and complete.

In particular, the UML addresses the specification of all the important analysis, design, and implementation decisions that must be made in developing and deploying a software-intensive system.

The UML is a Language for Constructing

The UML is not a visual programming language, but its models can be directly connected to a variety of programming languages.

It is possible to map from a model in the UML to a programming language such as Java, C++, or Visual Basic, or even to tables in a relational database or the persistent store of an object-oriented database.

Things that are best expressed graphically are done so graphically in the UML, whereas things that are best expressed textually are done so in the programming language.

The UML is a Language for Documenting

A healthy software organization produces all sorts of artifacts in addition to raw executable code.

These artefacts include (but are not limited to)

- Requirements- Architecture
- Design
- Source code
- Project plans
- Tests
- Prototypes
- Releases

Computational Reflection

“Computational reflection (sometimes just called *reflection*) is a computer process involving self-awareness ---reason about one's own processes.

A reflective program has the ability to *metaprogram* : it can, itself, write programs.

When a reflective program operates, it does so in the same manner as a person. It takes variables, such as its own conditions, and contextual information into account. “

http://whatis.techtarget.com/definition/computational-reflection

Computational Reflection : Some Analogy

“Think of the operations involved in getting from your car to your house. If you see an obstacle in your path, you take in that information and adapt to it by either stepping around or over the object, or picking it up.

When you get to your door, if you find it locked, usually you don't stop and stand there, continue to turn the knob, or turn around and walk away; usually you take out your key and unlock the door. In the same way, a reflective program has the ability to think about what is happening and to alter itself to address the circumstances.”

http://whatis.techtarget.com/definition/computational-reflection

What Does Computational Reflection Mean?

"Computational reflection is the ability of a program to modify itself while running.

The source code of the program is treated as data by itself and appropriate modifications can be made by the program during runtime.

Programs capable of modifying their own source code, or the source code of some other program, during runtime are called metaprograms.

Computational reflection allows the programmer to save time implementing some parts of the program, which are generated by the program itself at runtime."

http://www.techopedia.com/definition/16370/computational-reflection

Future Robots and Role of Cognition and Reflection

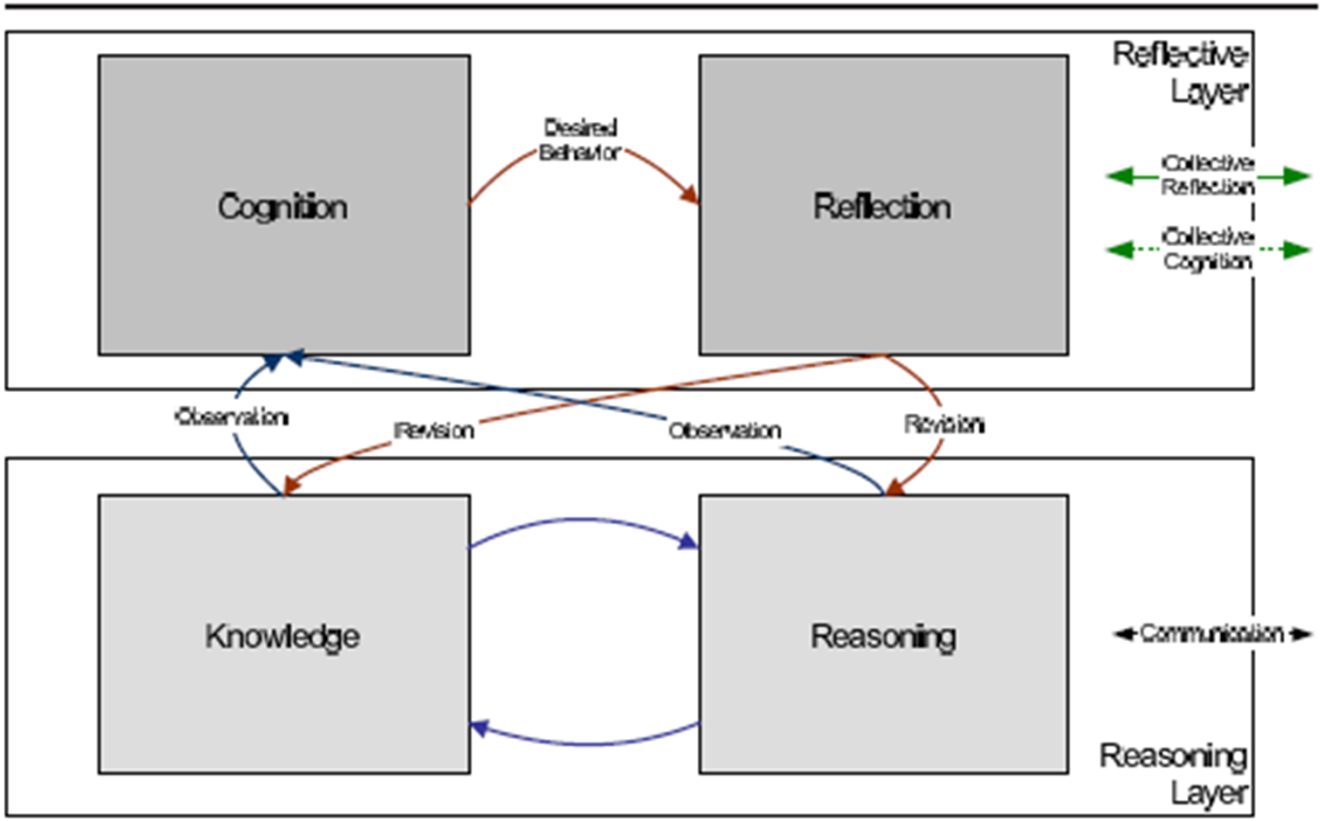


Figure 1. Role of cognition and reflection in the reflective agent.

An abstract architecture for computational reflection in multiagent systems, Martin Rahak et. al., Zech Technical University in Prague

UML Can be used Everywhere?

- Enterprise information systems
- Banking and financial services
- Telecommunications
- Transportation
- Defense/aerospace
- Retail
- Medical Electronics
- Scientific research
- Distributed Web based services
- Work flow in the legal system
- Patient healthcare system
- Design of hardware

Building Blocks of UML : Diagrams

•Diagrams group interesting collections of things

- Class diagram
  - Object diagram
  - Use case diagram
- Sequence diagram
- Collaboration diagram
- State chart diagram
- Activity diagram
- Component diagram
-Deployment diagram

UML Diagram

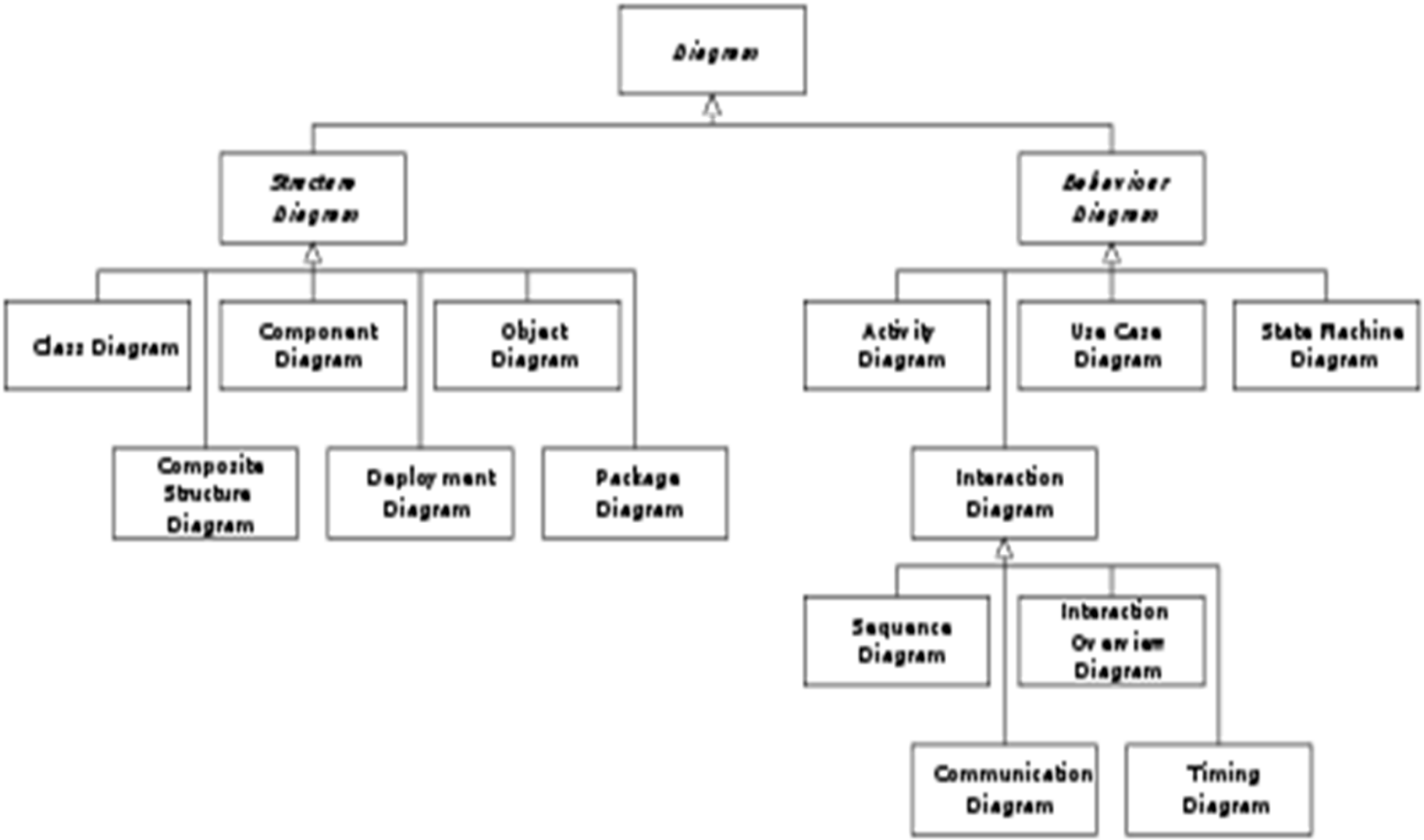
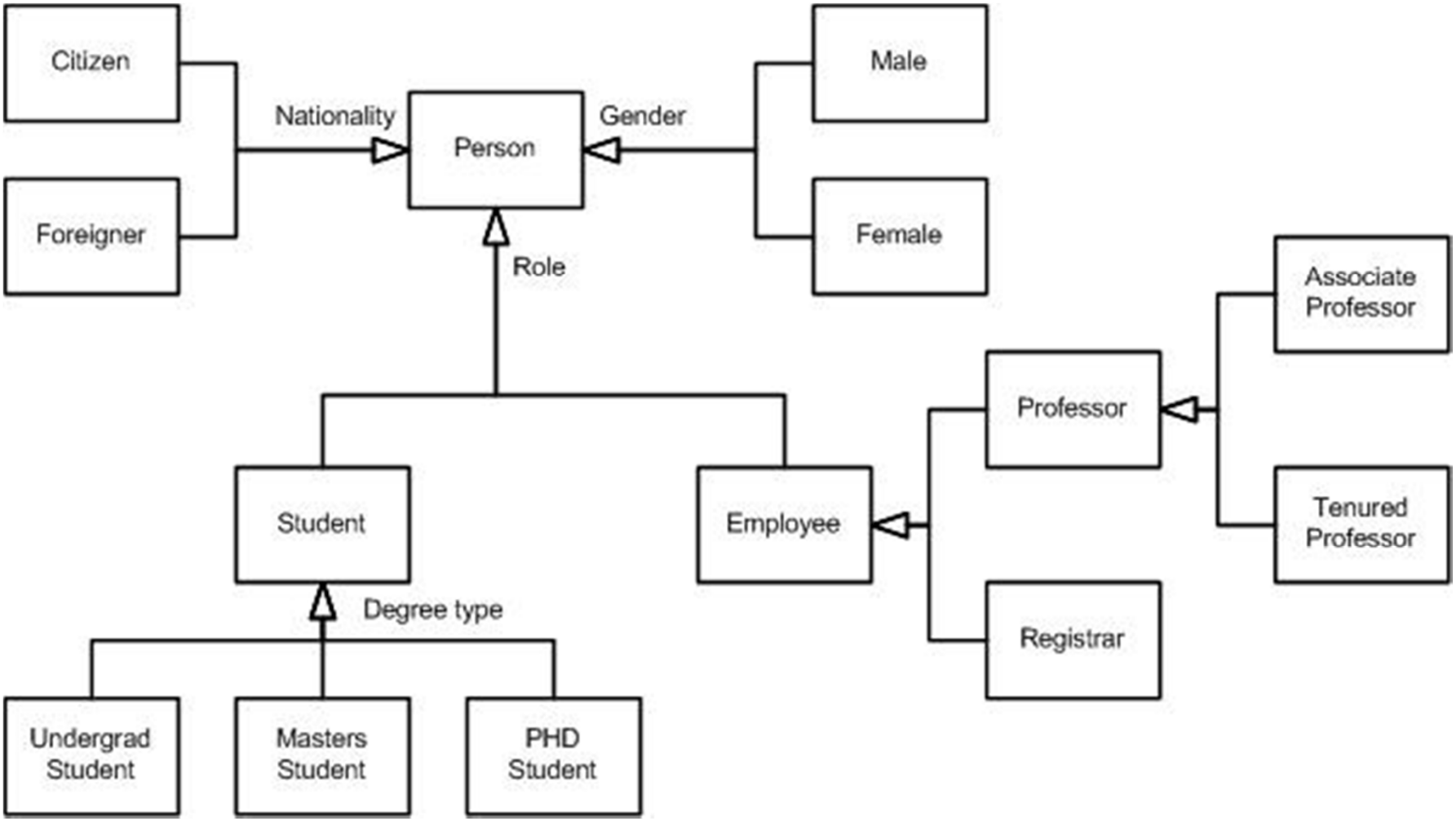


Diagram
Structure Diagram
Behavior Diagram
Class Diagram
Component Diagram
Object Diagram
Activity Diagram
Use Case Diagram
State Machine Diagram
Composite Structure Diagram
Deployment Diagram
Package Diagram
Interaction Diagram
Sequence Diagram
Interaction Overview Diagram
Communication Diagram
Timing Diagram

UML Class Diagram for a University



A Digital Sound Recorder: Requirements

A digital sound recorder is a consumer electronic appliance designed to record and play back speech.

The messages are recorded using a built-in microphone and they are stored in a digital memory.

The user can quickly play back any message at any moment trough a speaker placed in the front of the device.

It should be small, light, easy to use, and battery operated.

A Digital Sound Recorder: Appearance

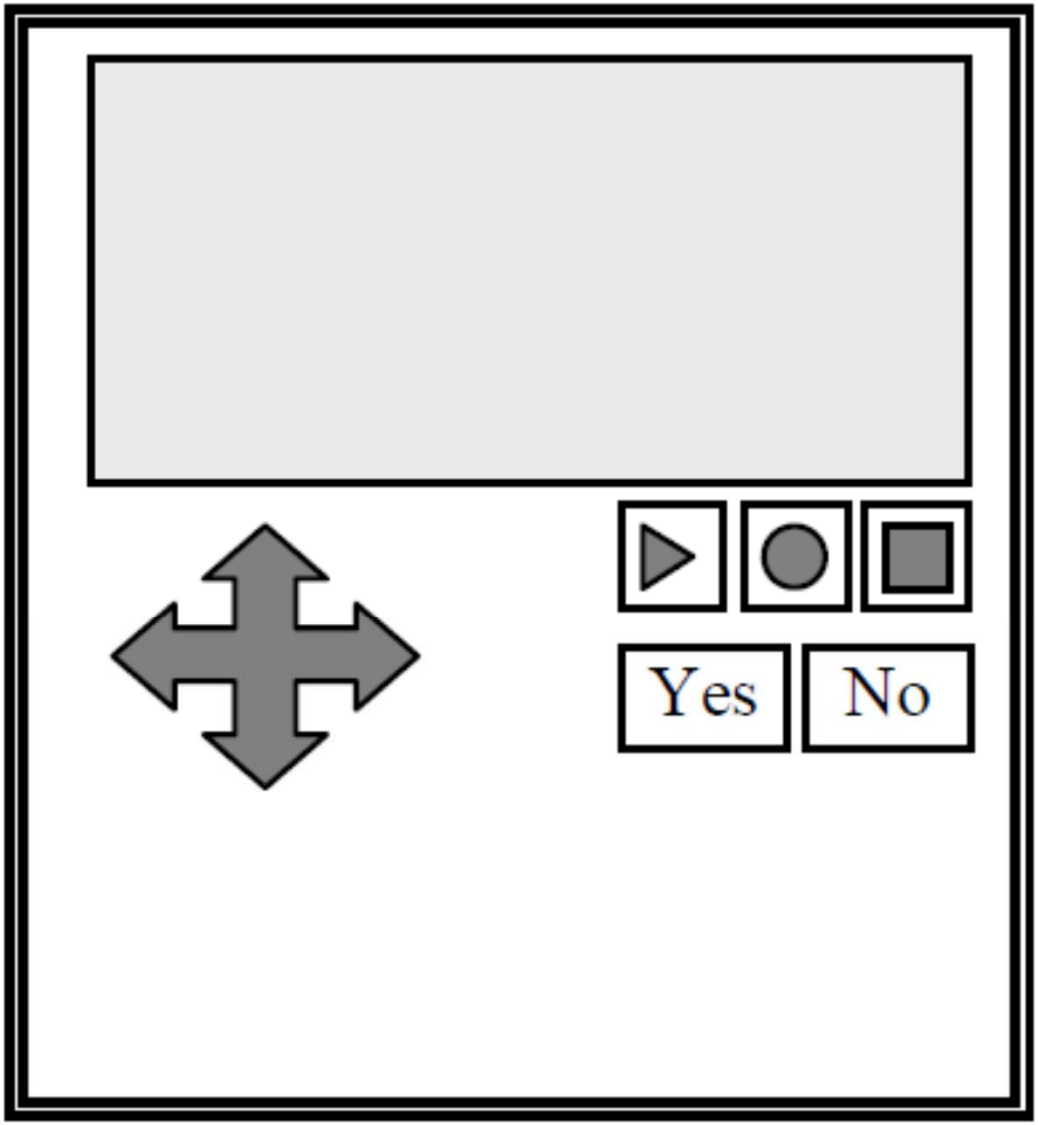


Figure 2.1: External appearance

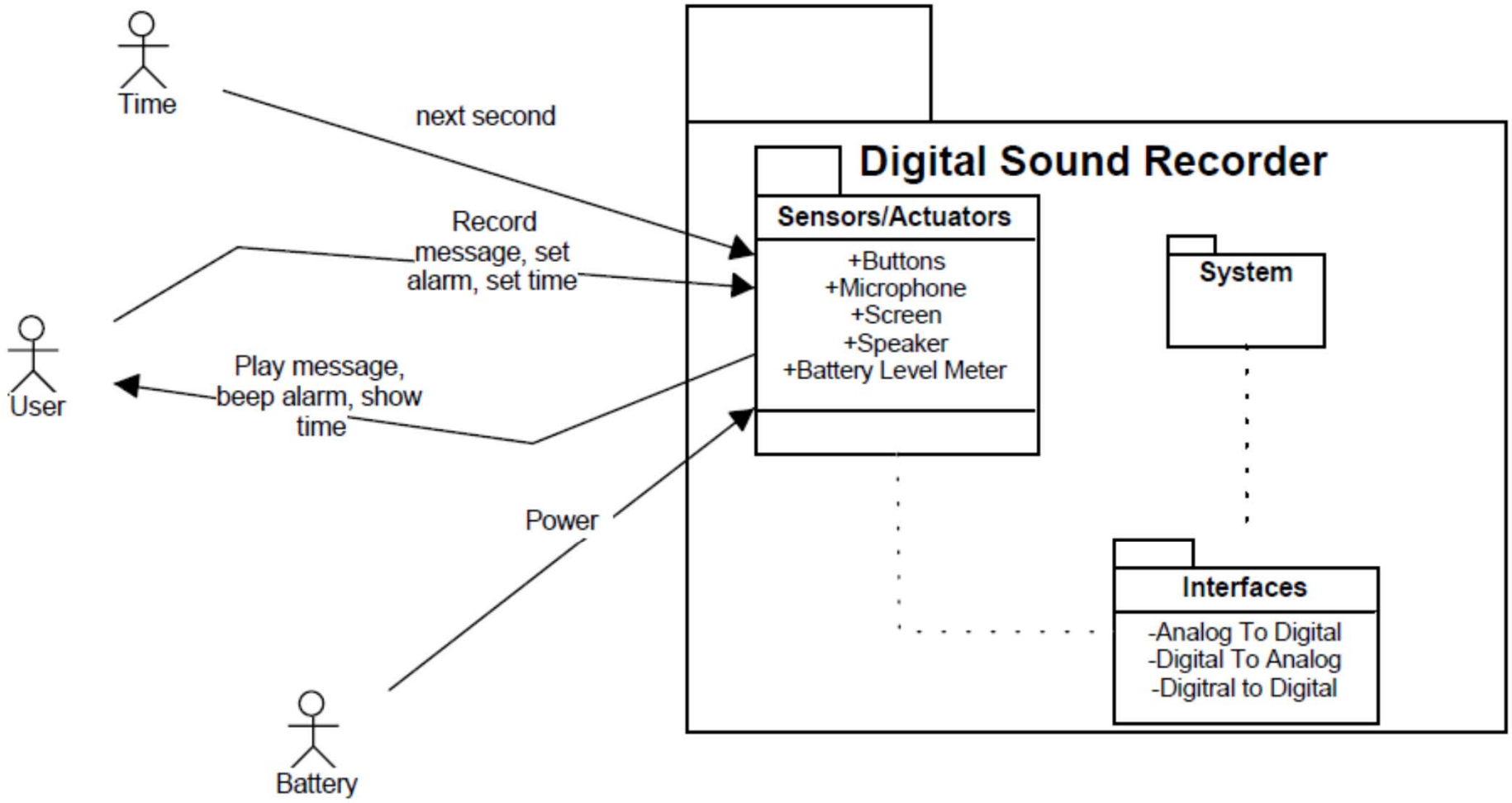
A Digital Sound Recorder: Features

- Capacity for ten different messages. The length of each message is limited by the available memory.
- Easy to use with on screen menus.
- Direct access to any message.
- Alarm clock with year-2000-ready calendar. The user can set a daily alarm. The alarm beeps until the user presses a key, or after 60 seconds.
- Full Function LCD Display. The current date and time is always shown in the display. The display also shows clear directions about how to use it and what it is doing.
- Battery-level indicator. The system beeps when the battery is low.
- Stand-by mode. It economises the battery power. The system switches off the peripherals when they are not in use. The normal operation is resumed when the user presses a key.
- Good sound quality. Sound is processed at 6Khz using eight bits per sample.

A Digital Sound Recorder: External Events

- An embedded system is constantly interacting with its environment.
- In this first stage of the analysis, we can consider our system as a black box reacting to the requests and messages from the environment.
- The environment is composed of several agents. Each agent interacts with our system with a different purpose and it exchanges a different set of messages.

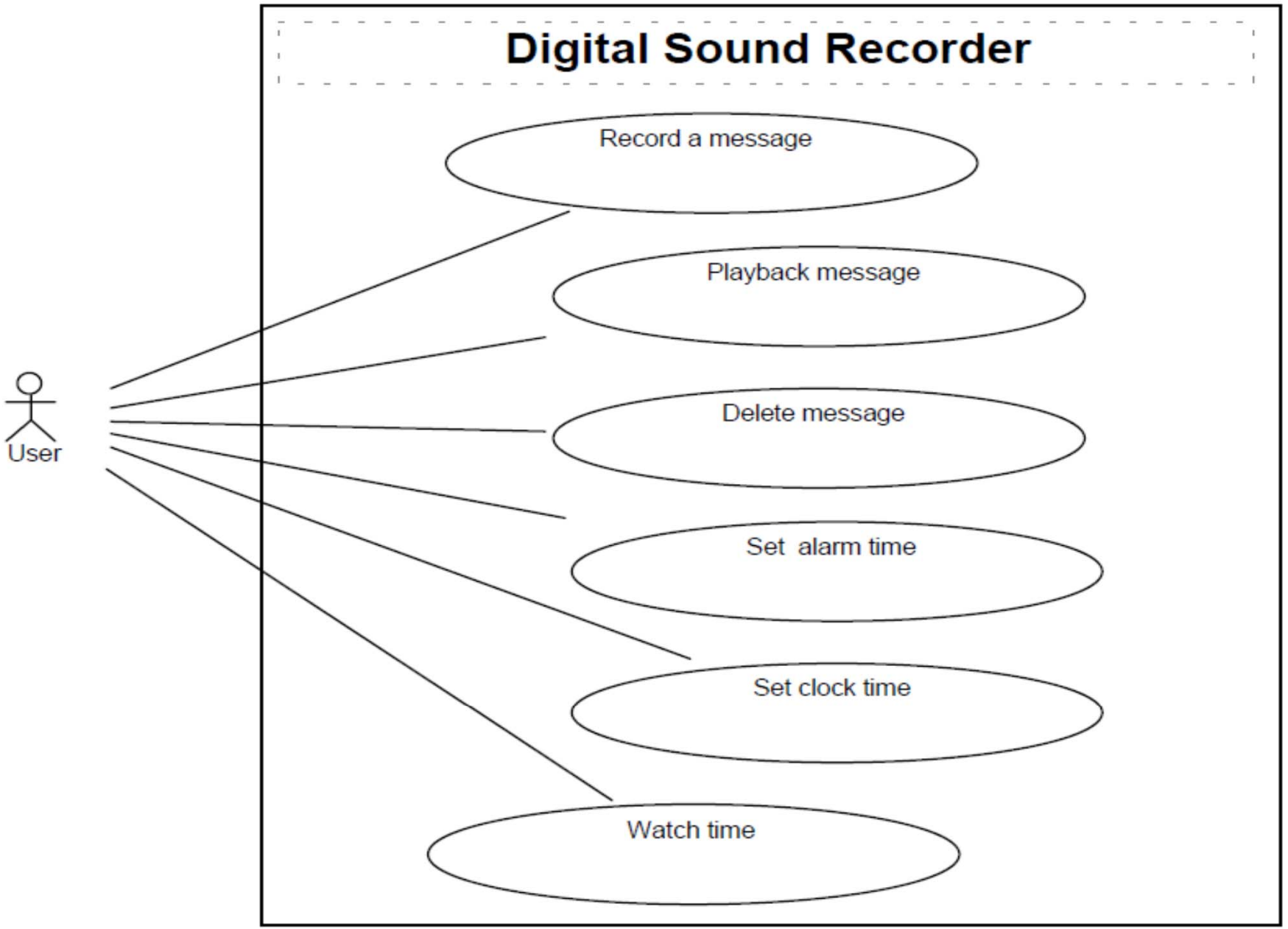
Context Level Diagram



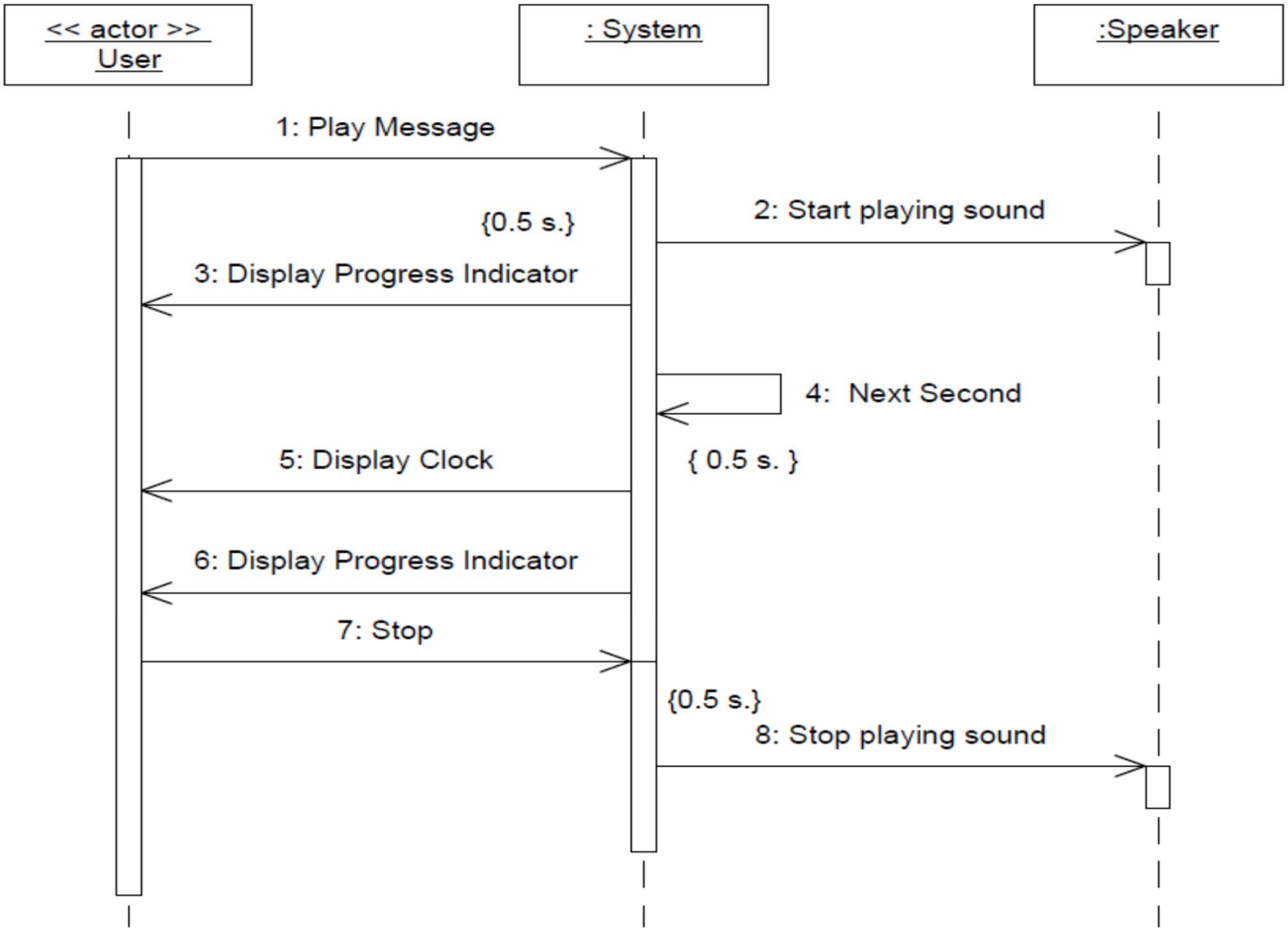
Events and Response

	Event	System Response	Direction	A	Resp.
1	A second passes	a. Update internal clock b. Check alarm c. Update clock display, d. Update task progress display.	In	P	0.5 s
2	A sample period passes	a. Play or record next sample	In	P	½ period
3	User presses a command button	a. Show task progress display b. Start recording or playing a message	In	E	0.5 s
4	User presses the "stop" button	a. Current task is stopped b. Update display	In	E	0.5 s
5	Low battery alarm	a. Warn the user and stop current task	In	E	1 s.
6	Enter stand-by mode	a. Switch off the display	In	E	1 s.
7	Wake up, user presses a button while in stand-by mode.	a. Leave stand-by mode, power up display, etc.	In	E	1 s.

Use case diagram



Playing Message Scenario



IBM Rational Software : Integrate, Collaborate, Optimize

Application lifecycle management
Manage the flow of people, process and information across the software and systems delivery lifecycle.

Enterprise modernization
Enable IT to streamline multiplatform application development, test and deployment.

Small and midsized business
Ensure software and systems delivery success for small and midsized businesses.

Complex and embedded systems
Manage and develop complex products and systems.

Solutions by industry
Rational software has a solution to support your business sector.

http://www-01.ibm.com/software/in/rational/

A case study

Oklahoma Department of Human Services deploys an IBM Rational System Architect and Rational DOORS solution to transform its disparate, redundant systems environment into a standardized, cost-efficient IT infrastructure with the scalability to adapt to the agency's ever-changing needs

Business need:
Oklahoma Department of Human Services sought to transform its disparate, redundant systems into a standardized, cost-efficient IT infrastructure with the scalability to adapt to its changing needs.
Solution:

The agency obtained a dynamic, holistic view of its IT infrastructure and business processes using IBM Rational System Architect leading to the implementation of system-wide improvements.

Benefits:
Oklahoma Department of Human Services agency reduced costs, streamlined business and IT processes, increased collaboration and improved program delivery by consolidating organizational assets.

Reference

1. The Unified Modeling Language User Guide: The ultimate tutorial to the UML from the original designers, Grady Booch, James Rumbaugh, Ivar Jacobson, 1999, Pearson Education Inc.
2. Digital Sound Recorder: A case study on designing embedded systems using the UML notation. Ivan Porres Paltor and Johan Lilius
   Åbo Akademi University, Department of Computer Science,
   Lemminkäisenkatu 14, FIN-20520 Turku, Finland
   email: Johan.Lilius@abo.fi