Object Design Document (ODD)

Computer emulator for digital preservation

Version : 1.1
Author : J.R. van der Hoeven
Date : 25-06-2007
Project : Emulation project

Koninklijke Bibliotheek, department HRD-DD
Nationaal Archief of the Netherlands

I. Revision history

Revision number	Revision date	Author	Summary of changes
1.1	25/06/2007	B. Lohman	Minor changes to text; clarifications

II. Related documents

Document name	Date	Author
User Requirements Document (URD)	21-02-2006	B. Lohman
Architectural Design Document (ADD)	03-03-2006	B. Lohman
Emulation – a viable preservation strategy	20-06-2005	J.R. van der Hoeven

III. Table of contents

1 Introduction

This Object Design Document (ODD) defines the object-level design of the emulator to be developed. It is based on the initial concept of the modular architecture, proposed in *Emulation – a viable preservation strategy,* and requirements and ADD defined in previous project stages.

The goal of this project is to develop an emulator based on the initial design shown in figure 1.1. Although this design depicts many different parts, only the modular emulator and a controller will be within scope.

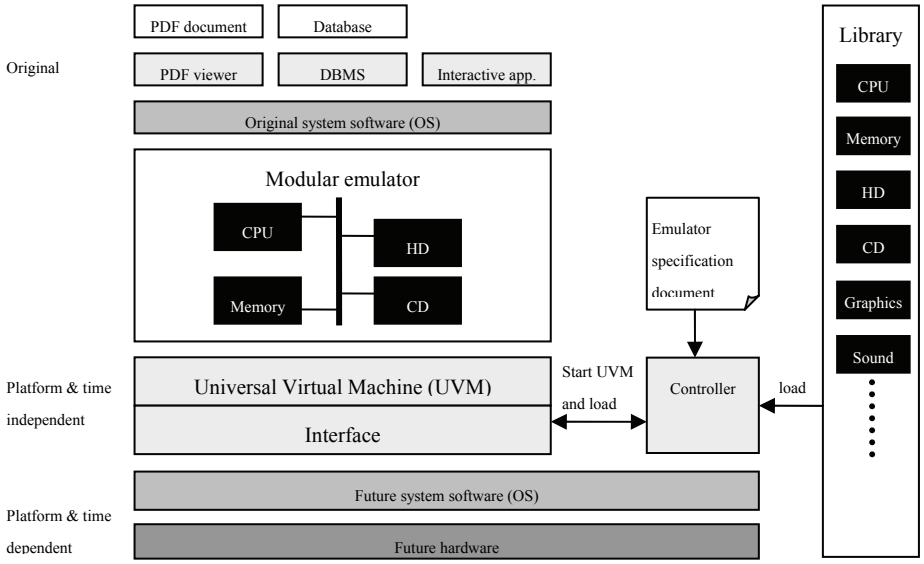

Figure 1.1 Initial design of a modular emulator for digital preservation

1.1 Object design trade-offs

Developing an emulator designed for digital preservation will be different from mainstream emulators. This is because the focus will be less on performance and more on durability and flexibility.

1.1.1 Durability versus platform dependance

Having an emulator running with great speed today doesn't help users in the future if the emulator doesn't work on newer platforms. Therefore, the emulator should have as few dependencies on the underlying computer platform as possible. In the initial design, a virtual machine is depicted to create a separate interface between the emulator and host platform.

1.1.2 Performance versus flexibility

Often, emulators are designed to achieve the highest possible execution speed, as emulators are normally operating at a slower pace than the real machine. But within the light of digital preservation, emulation will be used only if the original computer environment is not available anymore. As Moore's law is still applicable today, it is likely that computer systems in the future will turn out to be faster. Speculating on this line of thought, building an

emulator for digital preservation doesn't have to be focused on performance too much. Instead it would be useful to have an emulator that is easy to configure and less tight to get the maximum performance out of it. Therefore, flexibility is of higher priority than performance.

1.2 Definitions, acronyms, and abbreviations

The following definitions will be used:

Modular emulation	:	full emulation of hardware by emulating the components of a hardware architecture as individual emulators and interconnecting them to form a full emulation process. In this, each emulator forms a distinct module of the total emulation process offering the possibility to make flexible configurations of different modules.
Module	:	a software representation of the functionality of a hardware component.
Type of module	:	a classification of a module by type of hardware component. Examples of a type: cpu, memory, motherboard, etc. There can only be one type of module, but several implementations of a type may exist. For example: cpu is a type and may have implementations based on Intel 8086 or IBM PPC.
Name of module	:	the name of the module. This is similar as the name of the hardware component that it emulates.
ID of module	:	the unique identifier of the module.

2 Package and class diagrams

In overview, the emulation application will consist of several packages. This results in the following package overview (see figure 2.1). Note: not all module packages and classes are shown below. For ease of use, package root “emulator” is chosen, but this may differ from the final implementation of packages.

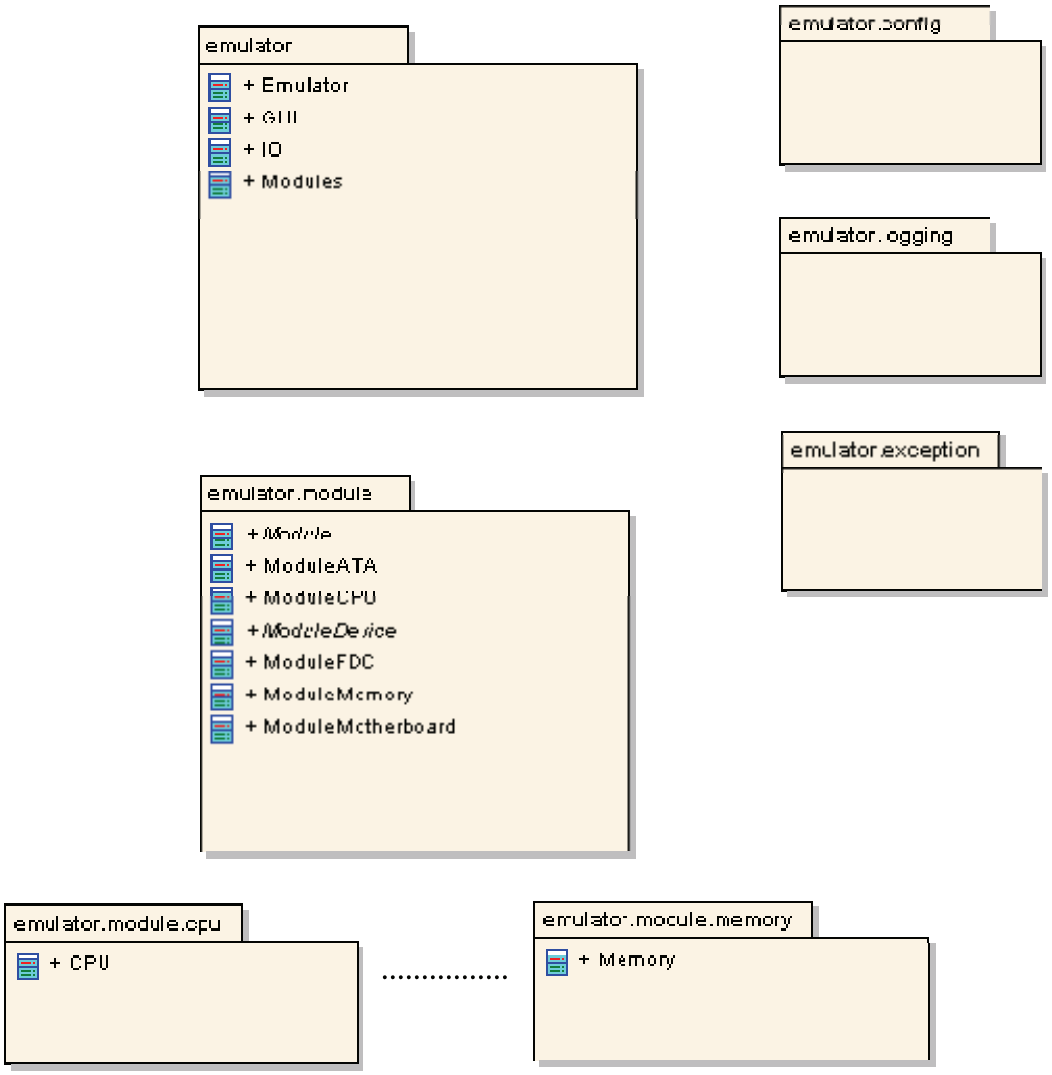

Figure 2.1: Emulator package overview

2.1 General application functionality

At the highest level, one package can be found:

`emulator`

Package `emulator` will contain classes such as `GUI` for the user interface, `IO` for importing and exporting data between emulator and environment, and `Emulator` for the actual control of the emulation process. Furthermore, a class `Modules` will be required which will contain all instances of modules in use by the emulation process. Below, a class diagram is shown for this package. Note: operations (methods) are not shown here.

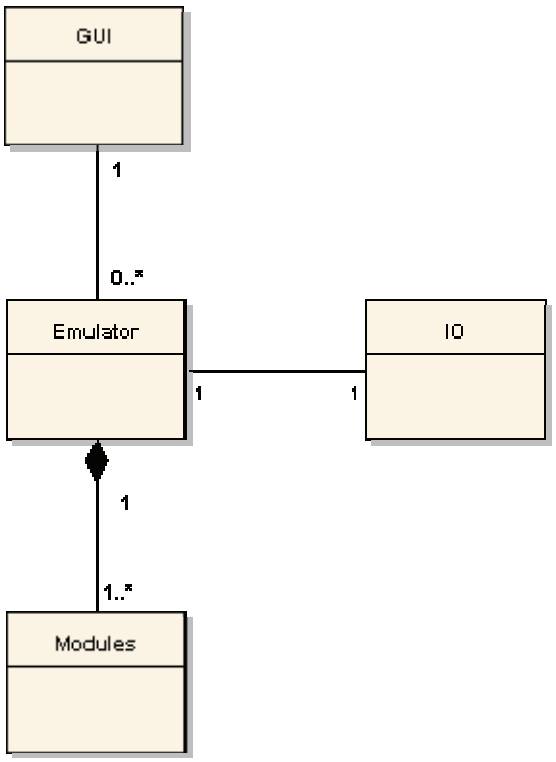

Figure 2.2: class diagram of package emulator

Besides the package `emulator`, other helper packages are required. These are:

```
emulator.config
emulator.logging
emulator.exception
```

These packages are responsible for handling the configuration of the emulator, logging of the execution status during an emulation process and exception handling.
The actual implementation is not known yet, but package `config` should be based on an XML-document (Emulator Specification Document as denoted in the overall conceptual model for this project) which has to be parsed by this package and the results should be set to the emulator.
Package `logging` can be based on the standard logging libraries of Java 1.4 or higher.
Package `exception` should offer several exception classes to handle module-specific exceptions.

2.2 Modular emulator

The part of the emulator that is responsible for emulating hardware is composed of modules as defined in the initial design. Each module represents a single hardware component in software and tries to mimic the exact functionality of it. In table 2.1 all modules and subsequent packages are shown that are assumed to be required for building an x86 computer emulator.

Table 2.1: module packages

Package	Functionality
`emulator.module.cpu`	Central Processing Unit
`emulator.module.memory`	Random Access Memory
`emulator.module.ata`	ATA / IDE controller to support hard disks and CD-ROM drives
`emulator.module.fdc`	Floppy Disk Controller to support floppy disks
`emulator.module.clock`	System clock
`emulator.module.bios`	BIOS for initialising and booting the system
`emulator.module.dma`	Direct Memory Access for fast transfer of data between memory and device
`emulator.module.keyboard`	Keyboard support
`emulator.module.mouse`	Mouse support
`emulator.module.motherboard`	Motherboard functionality like I/O address space control
`emulator.module.pic`	Programmable Interrupt Controller for handling interrupts of devices
`emulator.module.rtc`	Real-time clock for CMOS registers and storing date and time information
`emulator.module.screen`	Virtual computer screen
`emulator.module.video`	Video adapter support

2.2.1 Uniqueness

Each module represents a unique implementation of a hardware component. Therefore, modules should have a unique identifier and type classification. The type defines which kind of computer component it emulates while the ID points out which implementation it is. Examples of types could be “cpu”, “memory”, “bios”, etc. Type definitions should be used in the name of the package, whereas the ID should included in the source code. Also, both type and ID should be part of the metadata when forming a module library (see initial design).

2.2.2 Interoperability

As most modules do not operate standalone, a communication mechanism is required to let each individual module communicate with other modules. Therefore, each module should implement a standard set of methods to support interoperability between other modules. In table 2.2 a list of predefined functions is given which is likely to be useful.

Table 2.2: List of standard functions for a module

Function	Description
getType()	Returns the type of the module
getID()	Returns the ID of the module
getName()	Returns the name of the emulated component
getConnection()	Returns a list of module types for which this module requires a connection to
setConnection(Module)	Create a relation to given module reference

isConnected()	Check if all required relations are set for this module
Start()	Start emulation process for this module
Stop()	Stop emulation process for this module
Reset()	Reset emulation process for this module
isObserved()	Check if this module is in an observable mode (to allow closer examination of its operation)
setObserved()	Put this module in observable mode
getDebugMode()	Returns the status of the modules debug mode
setDebugMode()	Set the status of debug mode for this module
getData(Module)	General method to send data to the calling module
setData(data, Module)	General method to retrieve data from sending module
getDump()	Returns a status dump of this module

Based on the kind of hardware component, a module will contain more specific functions to emulate the functionality of the hardware. As these functions may be invoked by other modules, it is necessary to define more type-specific functionality. To do so, each module not only has to implement the general functions, but also needs to be extended with type-specific functions. For example, module memory could be extended with functions for reading and writing bytes from and to RAM. Which functions are required, is uncertain until it is known which functionality of a component has to be emulated and should be shared with other modules.

2.2.3 Modular organisation

To make sure each module implements the general and type-specific functionality, modules should implement a standard abstract class. Each module should have an equivalent abstract class. For example, module Memory should have an abstract class ModuleMemory. This abstract class will serve as a blueprint for the module memory and forces each module that emulates memory to implement all required general and type-specific functions.

Figure 2.3 depicts the general class organisation for the modules mentioned earlier.

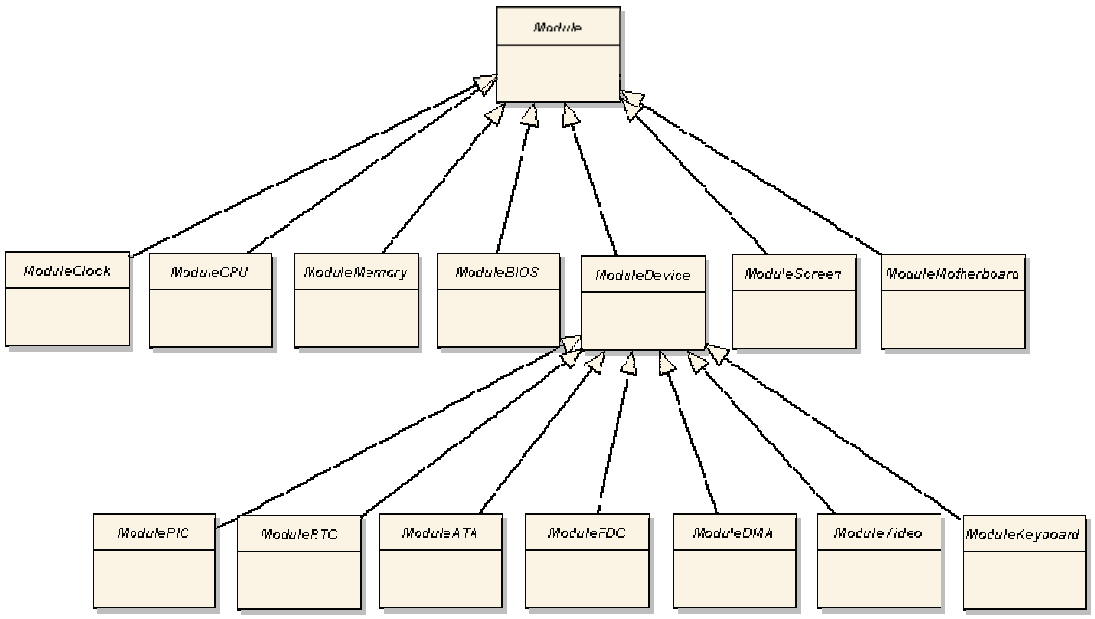

Figure 2.3: class diagram of abstract module classes

In figure 2.3, modules clock, CPU, memory, BIOS, screen and motherboard directly inherit the general methods from abstract class Module. The Each abstract type-specific module class extends this general set of methods with its own type-specific methods.

Modules PIC, RTC, ATA, FDC, DMA, video and keyboard do not directly inherit from abstract class Module. The difference is based on the way their real life hardware equivalents communicate. Each of these components are called devices and use a centrally organised I/O address space for communication. By introducing an intermediate abstract class ModuleDevice, all modules that represent a device are able to use a similar mechanism like I/O address space. Implementing methods from abstract ModuleDevice allows a device to register a particular range of I/O address space and transfer data between it.

2.2.4 Reusability

The big advantage of the modular organisation of the emulator is that models can be reused to operate in any emulation process. Reconnecting them to the required surrounding modules is sufficient to let the module operate correctly. Also, various software implementations of a hardware component can be created and used while the communication between other modules will stay the same.

Another benefit which will be useful during development, is that individual modules can be implemented and refined, while leaving the emulator itself untouched. Developers can work on different modules simultaneously without interfering with each others implementation.