# CS 441 Software Engineering
# Assignment 3

Due on Thursday, March 5, 11:59PM.

Gustavo Hammerschmidt[*]
California State University San Marcos

February 25, 2020

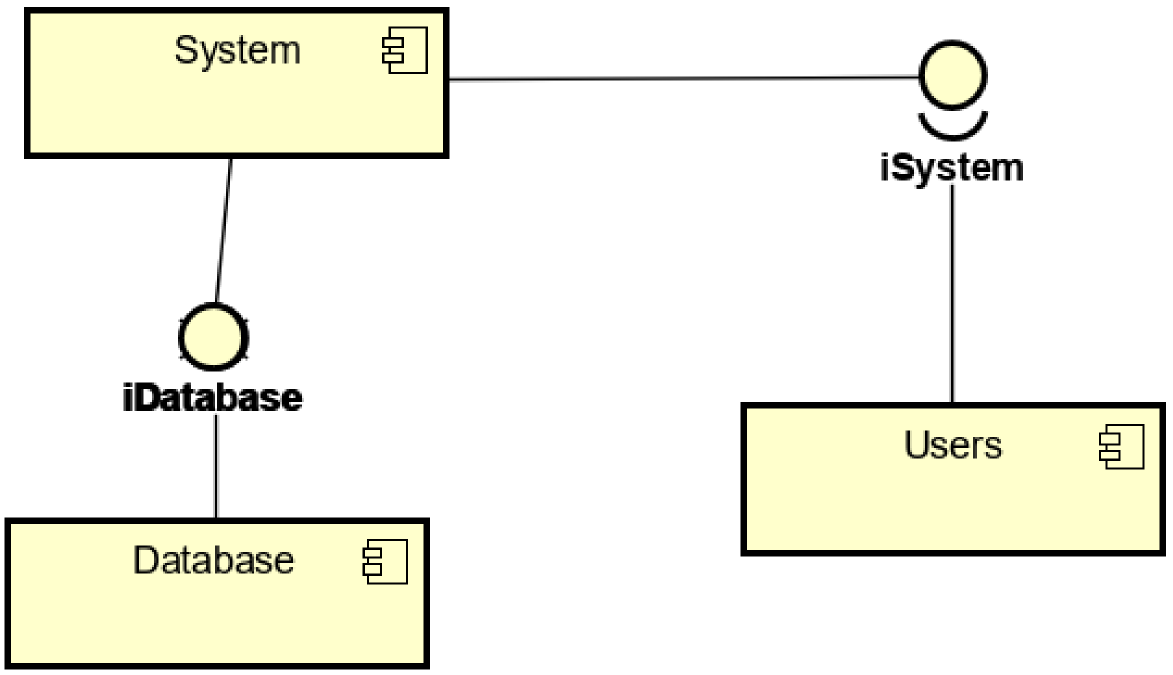
# 1 Individual Portion

## 1.1 First question

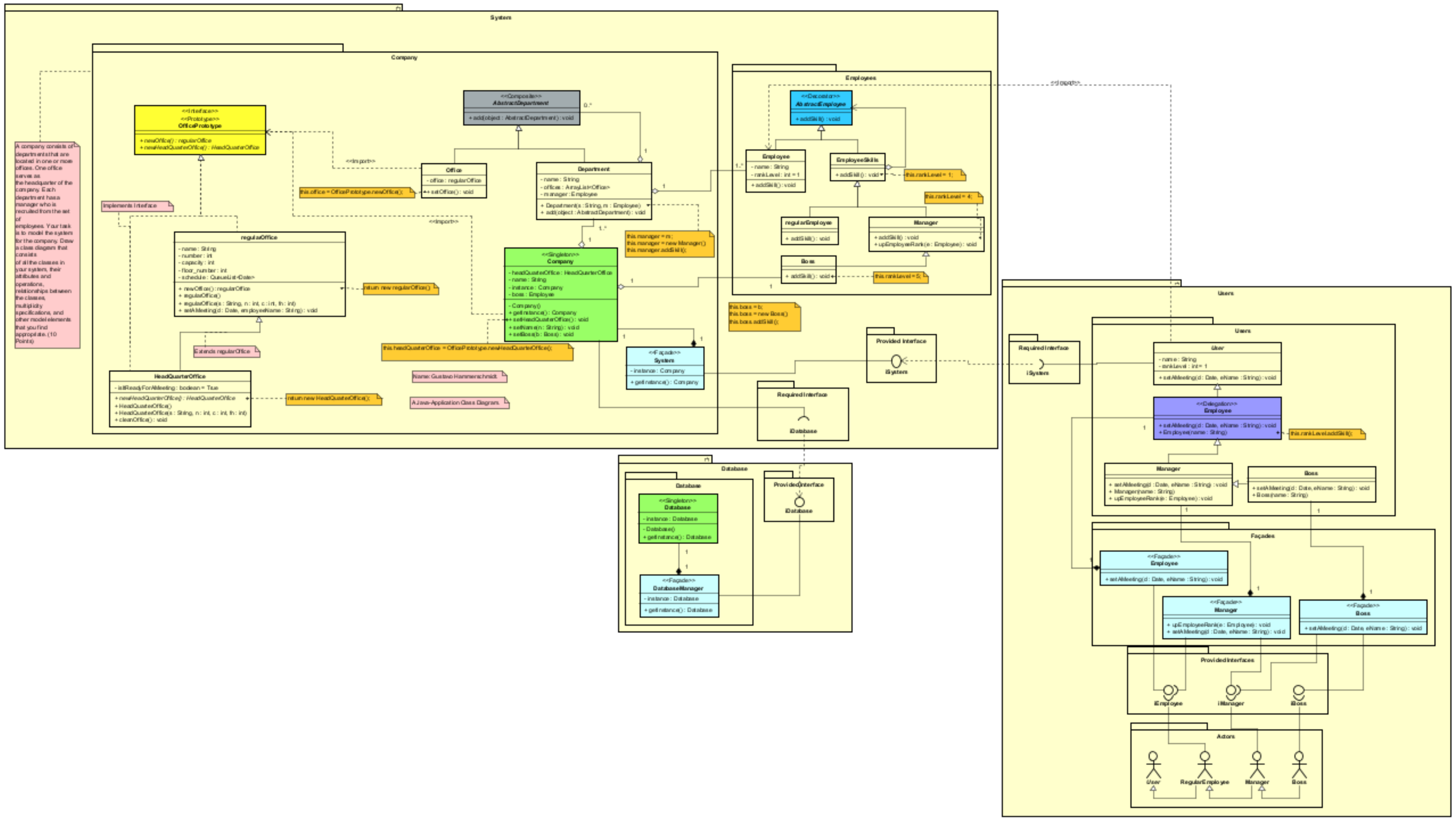
1. A company consists of departments that are located in one or more offices. One office serves as the headquarter of the company. Each department has a manager who is recruited from the set of employees. Your task is to model the system for the company. Draw a class diagram that consists of all the classes in your system, their attributes and operations, relationships between the classes, multiplicity specifications, and other model elements that you find appropriate. (10 Points)

My answer to this question is an UML diagram that does not fit into this paper, therefore, I will show a component diagram of it to clarify the space position of each component in the class diagram. I will show the whole-picture of the class diagram and separately will show each component in the diagram. Follow the caption of the images. The UML file will be posted too. I use the astah tool to create my UML files.

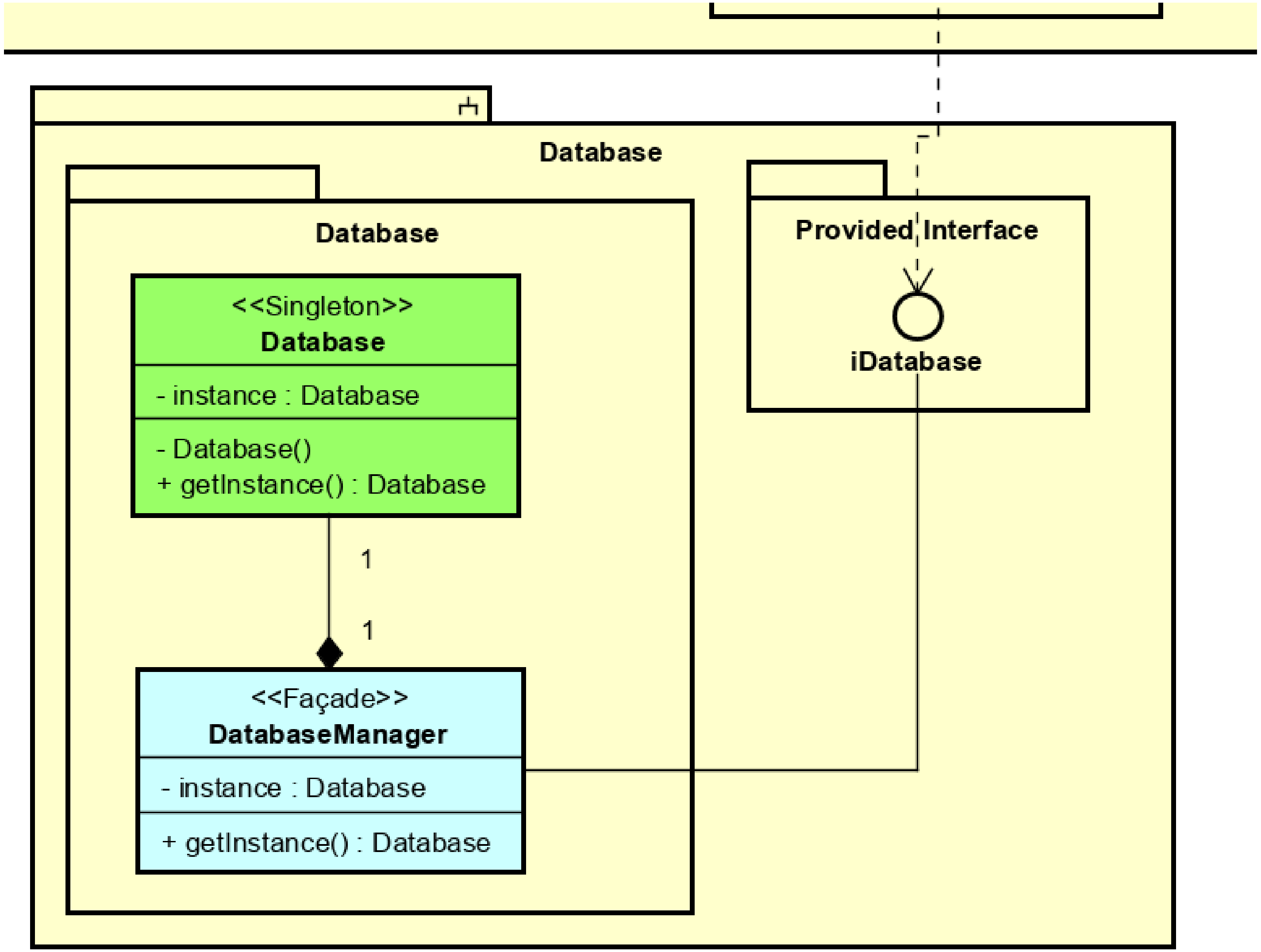
[*] hamme032@cougars.csusm.edu



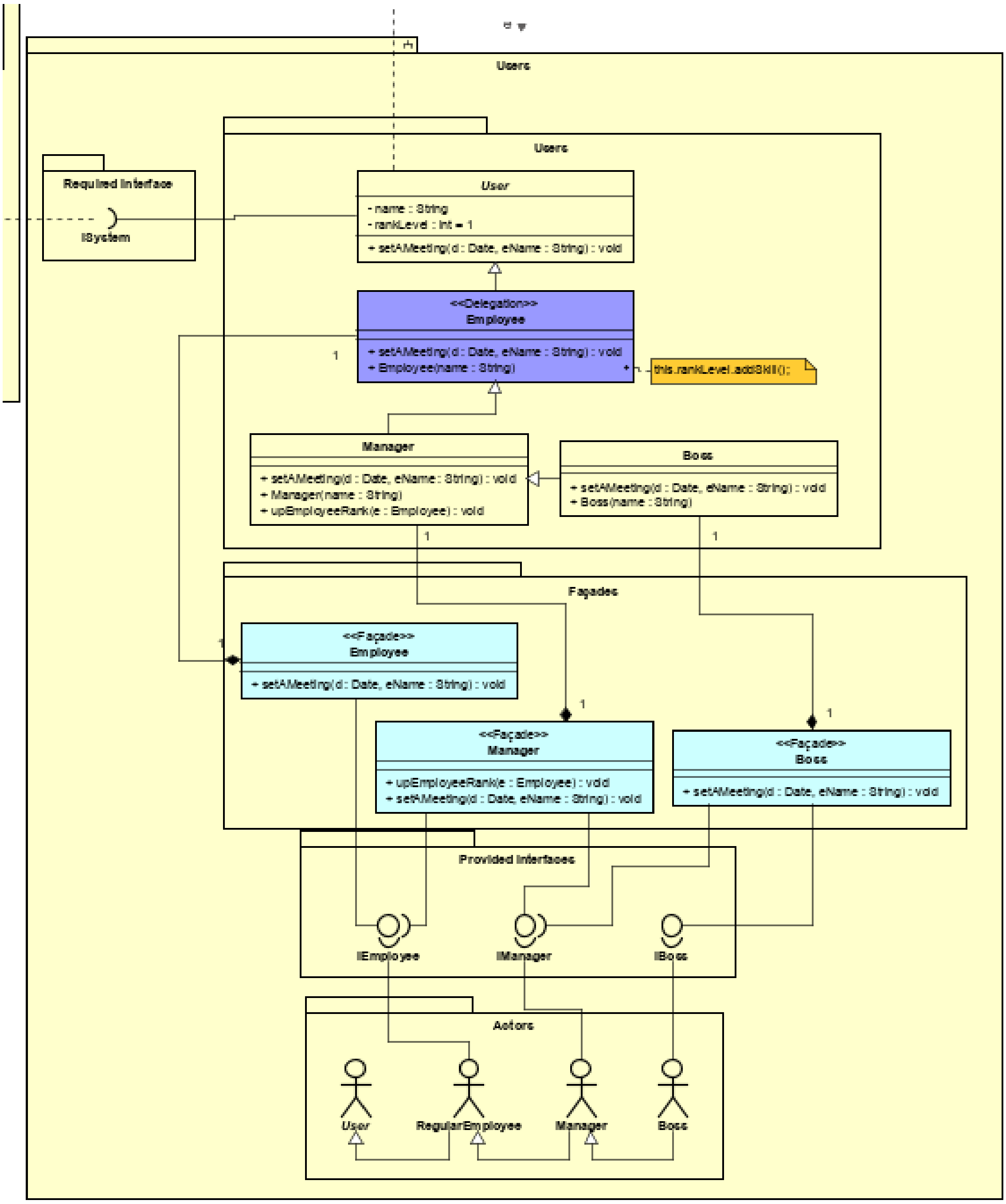
(a) Component diagram of the class diagram


System
Company
<<Interface>>
<<Prototype>>
OfficePrototype
+ newOffice() : regularOffice
+ newHeadQuarterOffice() : HeadQuarterOffice
A company consists of departments that are located in one or more offices. One office serves as the headquarter of the company. Each department has a manager who is recruited from the set of employees. Your task is to model the system for the company. Draw a class diagram that consists of all the classes in your system, their attributes and operations, relationships between the classes, multiplicity specifications, and other model elements that you find appropriate. (10 Points)
Implements Interface
<<import>>
this.office = OfficePrototype.newOffice();
<<import>>
regularOffice
- name : String
- number : int
- capacity : int
- floor_number : int
- schedule : QueueList<Date>
+ newOffice() : regularOffice
+ regularOffice()
+ regularOffice(s : String, n : int, c : int, fn : int)
+ setAMeeting(d : Date, employeeName : String) : void
return new regularOffice()
Extends regularOffice
HeadQuarterOffice
- isItReadyForAMeeting : boolean = True
+ newHeadQuarterOffice() : HeadQuarterOffice
+ HeadQuarterOffice()
+ HeadQuarterOffice(s : String, n : int, c : int, fn : int)
+ cleanOffice() : void
return new HeadQuarterOffice();
<<Composite>>
AbstractDepartment
+ add(object : AbstractDepartment) : void
0..*
1
Office
- office : regularOffice
+ setOffice() : void
Department
- name : String
- offices : ArrayList<Office>
- manager : Employee
+ Department(s : String, m : Employee)
+ add(object : AbstractDepartment) : void
1..*
this.manager = m;
this.manager = new Manager()
this.manager.addSkill();
<<Singleton>>
Company
- headQuarterOffice : HeadQuarterOffice
- name : String
- instance : Company
- boss : Employee
- Company()
+ getInstance() : Company
+ setHeadQuarterOffice() : void
+ setName(n : String) : void
+ setBoss(b : Boss) : void
this.headQuarterOffice = OfficePrototype.newHeadQuarterOffice();
Name: Gustavo Hammerschmidt.
A Java-Application Class Diagram.
<<Façade>>
System
- instance : Company
+ getInstance() : Company
Employees
<<Decorator>>
AbstractEmployee
+ addSkill() : void
Employee
- name : String
- rankLevel : int = 1
+ addSkill() : void
EmployeeSkills
+ addSkill () : void
this.rankLevel = 1;
this.rankLevel = 4;
regularEmployee
+ addSkill() : void
Manager
+ addSkill() : void
+ upEmployeeRank(e : Employee) : void
Boss
+ addSkill() : void
this.rankLevel = 5;
this.boss = b;
this.boss = new Boss()
this.boss.addSkill();
Provided Interface
iSystem
Required Interface
iDatabase
<<import>>
Database
Database
<<Singleton>>
Database
- instance : Database
- Database()
+ getInstance() : Database
<<Façade>>
DatabaseManager
- instance : Database
+ getInstance() : Database
ProvidedInterface
iDatabase
Users
Required Interface
iSystem
Users
User
- name : String
- rankLevel : int = 1
+ setAMeeting(d : Date, eName : String) : void
<<Delegation>>
Employee
+ setAMeeting(d : Date, eName : String) : void
+ Employee(name : String)
this.rankLevel.addSkill();
Manager
+ setAMeeting(d : Date, eName : String) : void
+ Manager(name : String)
+ upEmployeeRank(e : Employee) : void
Boss
+ setAMeeting(d : Date, eName : String) : void
+ Boss(name : String)
Façades
<<Façade>>
Employee
+ setAMeeting(d : Date, eName : String) : void
<<Façade>>
Manager
+ upEmployeeRank(e : Employee) : void
+ setAMeeting(d : Date, eName : String) : void
<<Façade>>
Boss
+ setAMeeting(d : Date, eName : String) : void
Provided Interfaces
iEmployee
iManager
iBoss
Actors
User
RegularEmployee
Manager
Boss

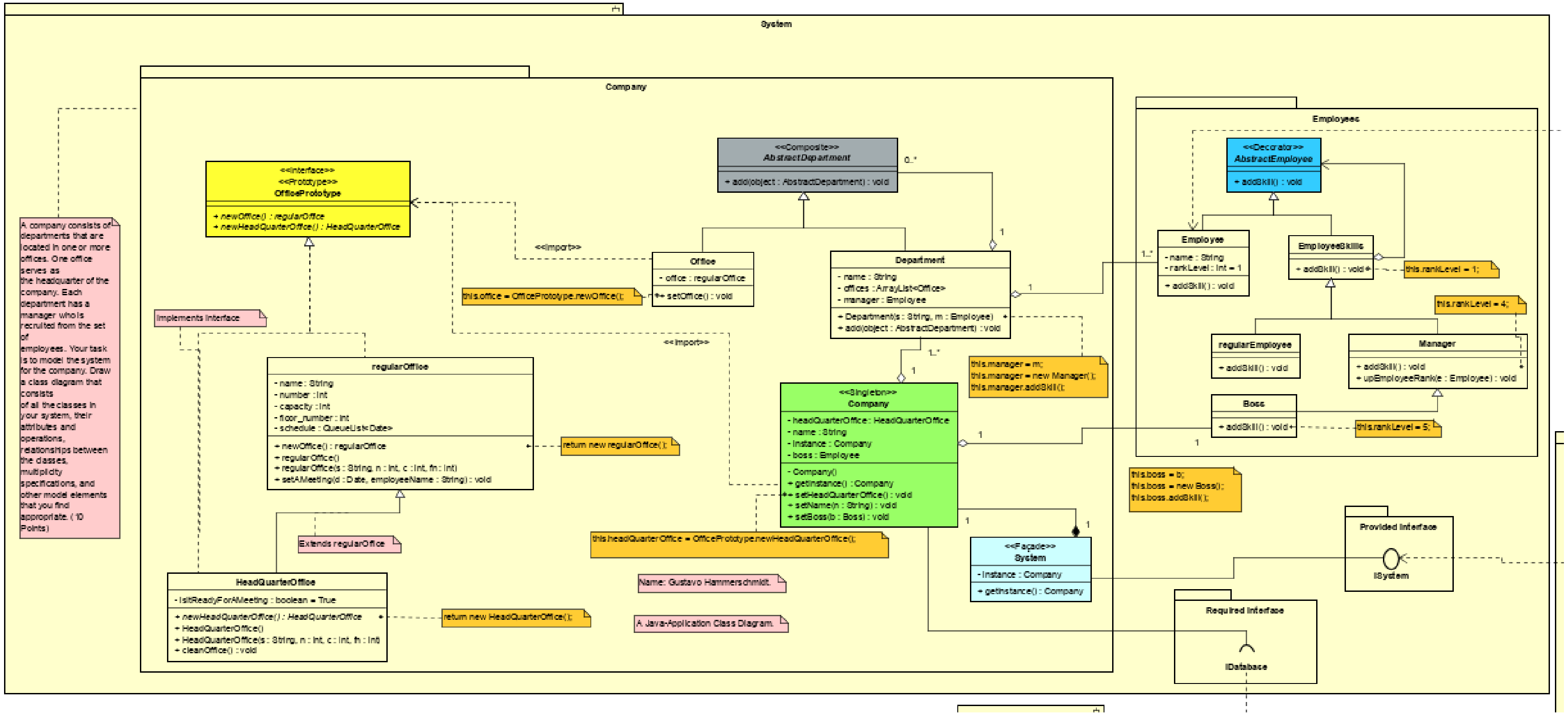
(a) The big picture of the class diagram



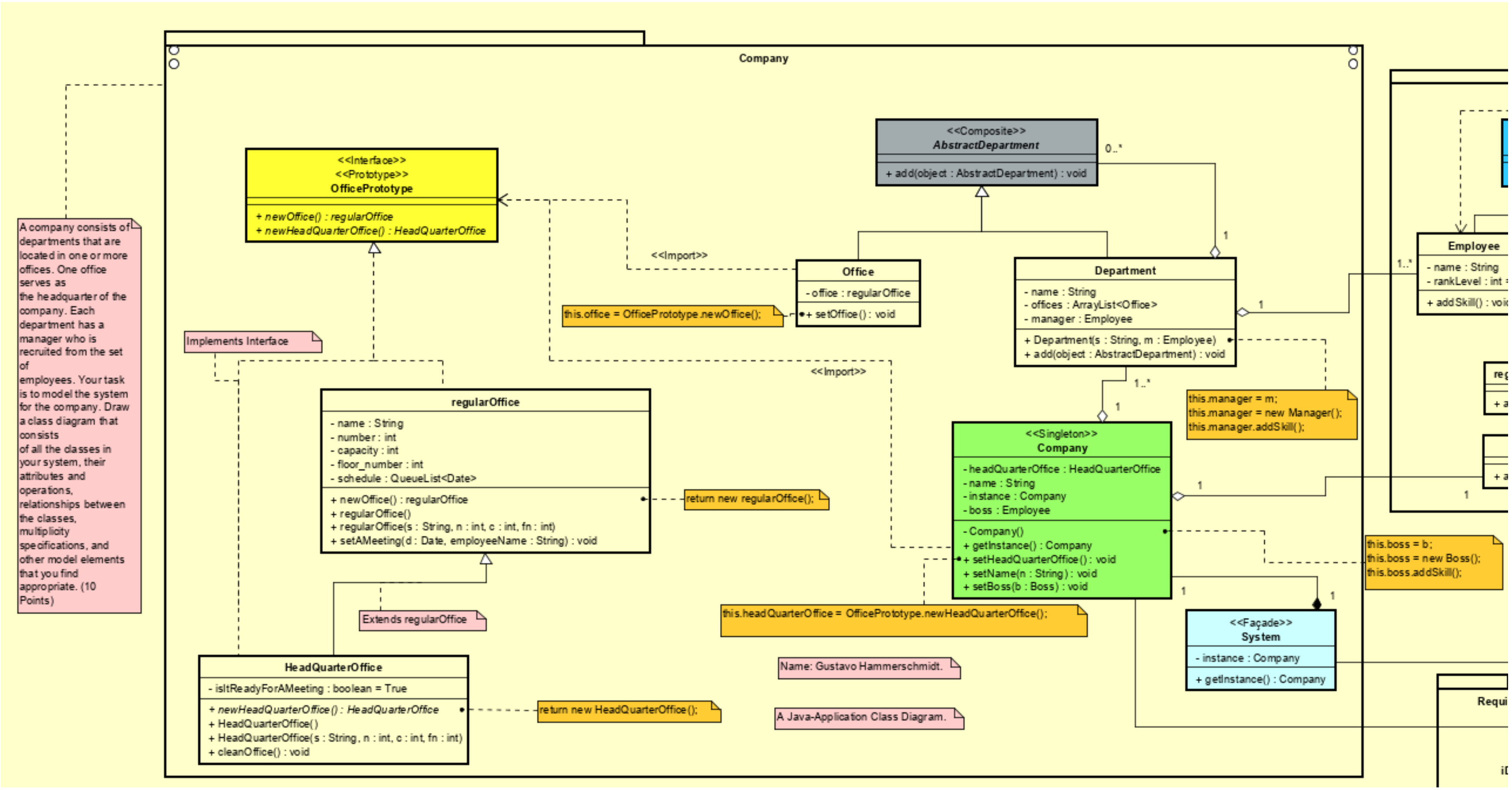
(a) The database component in the class diagram


Users
Users
Required Interface
ISystem
User
- name : String
- rankLevel : int = 1
+ setAMeeting(d : Date, eName : String) : void
<<Delegation>>
Employee
+ setAMeeting(d : Date, eName : String) : void
+ Employee(name : String)
this.rankLevel.addSkill();
1
Manager
+ setAMeeting(d : Date, eName : String) : void
+ Manager(name : String)
+ upEmployeeRank(e : Employee) : void
Boss
+ setAMeeting(d : Date, eName : String) : void
+ Boss(name : String)
1
1
Façades
<<Façade>>
Employee
+ setAMeeting(d : Date, eName : String) : void
1
<<Façade>>
Manager
+ upEmployeeRank(e : Employee) : void
+ setAMeeting(d : Date, eName : String) : void
1
<<Façade>>
Boss
+ setAMeeting(d : Date, eName : String) : void
1
Provided Interfaces
IEmployee
IManager
IBoss
Actors
User
RegularEmployee
Manager
Boss

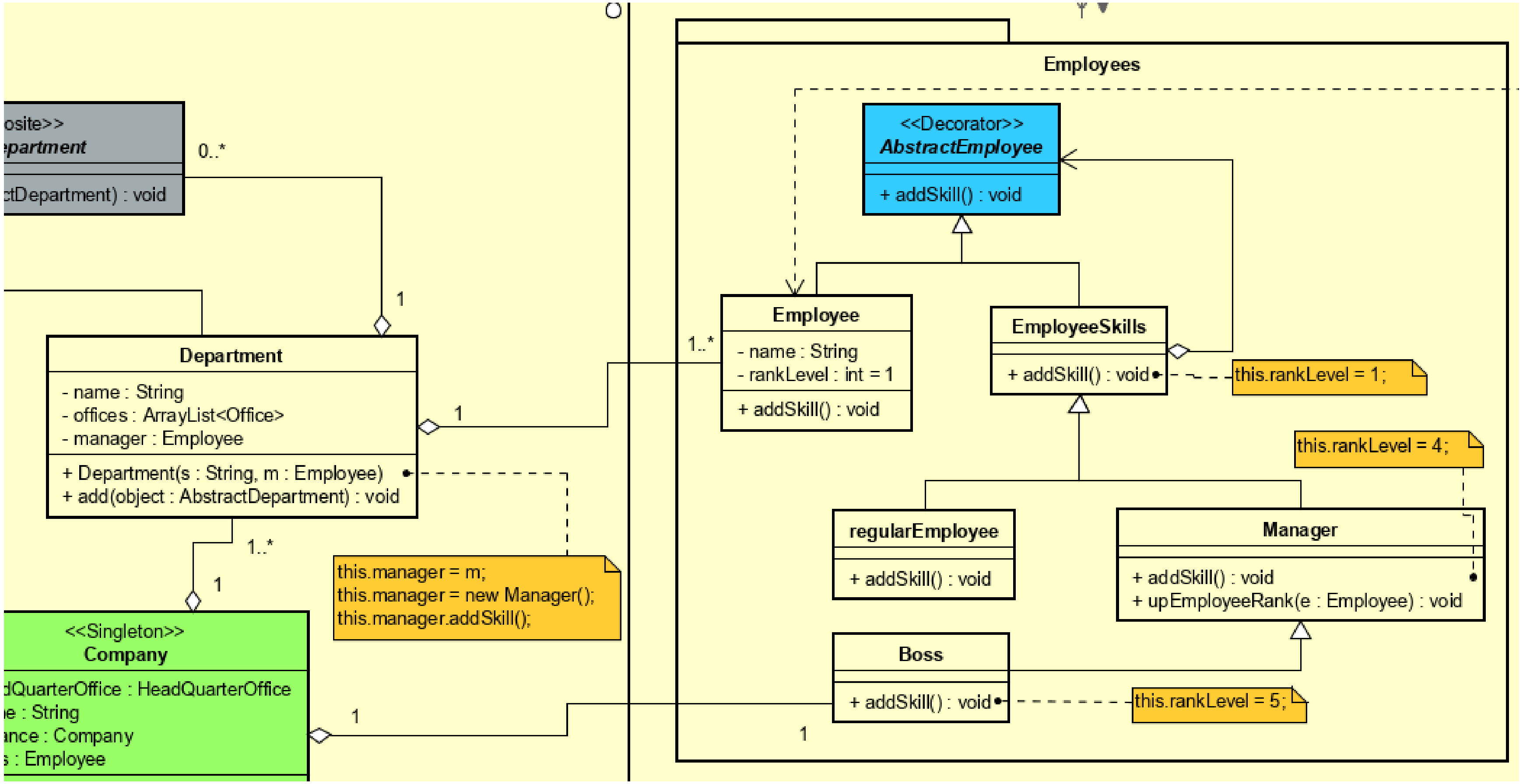
(a) The user component in the class diagram


System
Company
Employees
A company consists of departments that are located in one or more offices. One office serves as the headquarter of the company. Each department has a manager who is recruited from the set of employees.. Your task is to model the system for the company. Draw a class diagram that consists of all the classes in your system, their attributes and operations, relationships between the classes, multiplicity specifications, and other model elements that you find appropriate. ( 10 Points)
<<Interface>>
<<Prototype>>
OfficePrototype
+ newOffice() : regularOffice
+ newHeadQuarterOffice() : HeadQuarterOffice
<<import>>
<<import>>
this.office = OfficePrototype.newOffice();
Implements Interface
regularOffice
- name : String
- number : int
- capacity : int
- floor_number : int
- schedule : QueueList<Date>
+ newOffice() : regularOffice
+ regularOffice()
+ regularOffice(s : String, n : int, c : int, fn : int)
+ setAMeeting(d : Date, employeeName : String) : void
return new regularOffice();
Extends regularOffice
HeadQuarterOffice
- IsItReadyForAMeeting : boolean = True
+ newHeadQuarterOffice() : HeadQuarterOffice
+ HeadQuarterOffice()
+ HeadQuarterOffice(s : String, n : int, c : int, fn : int)
+ cleanOffice() : void
return new HeadQuarterOffice();
<<Composite>>
AbstractDepartment
+ add(object : AbstractDepartment) : void
0..*
1
Office
- office : regularOffice
+ setOffice() : void
Department
- name : String
- offices : ArrayList<Office>
- manager : Employee
+ Department(s : String, m : Employee)
+ add(object : AbstractDepartment) : void
1..*
1
this.manager = m;
this.manager = new Manager();
this.manager.addSkill();
<<Singleton>>
Company
- headQuarterOffice : HeadQuarterOffice
- name : String
- Instance : Company
- boss : Employee
- Company()
+ getInstance() : Company
+ setHeadQuarterOffice() : void
+ setName(n : String) : void
+ setBoss(b : Boss) : void
this.headQuarterOffice = OfficePrototype.newHeadQuarterOffice();
Name: Gustavo Hammerschmidt.
A Java-Application Class Diagram.
<<Façade>>
System
- Instance : Company
+ getInstance() : Company
<<Decorator>>
AbstractEmployee
+ addSkill() : void
Employee
- name : String
- rankLevel : int = 1
+ addSkill() : void
1..*
EmployeeSkills
+ addSkill() : void
this.rankLevel = 1;
this.rankLevel = 4;
regularEmployee
+ addSkill() : void
Manager
+ addSkill() : void
+ upEmployeeRank(e : Employee) : void
Boss
+ addSkill() : void
this.rankLevel = 5;
this.boss = b;
this.boss = new Boss();
this.boss.addSkill();
Provided Interface
ISystem
Required Interface
IDatabase

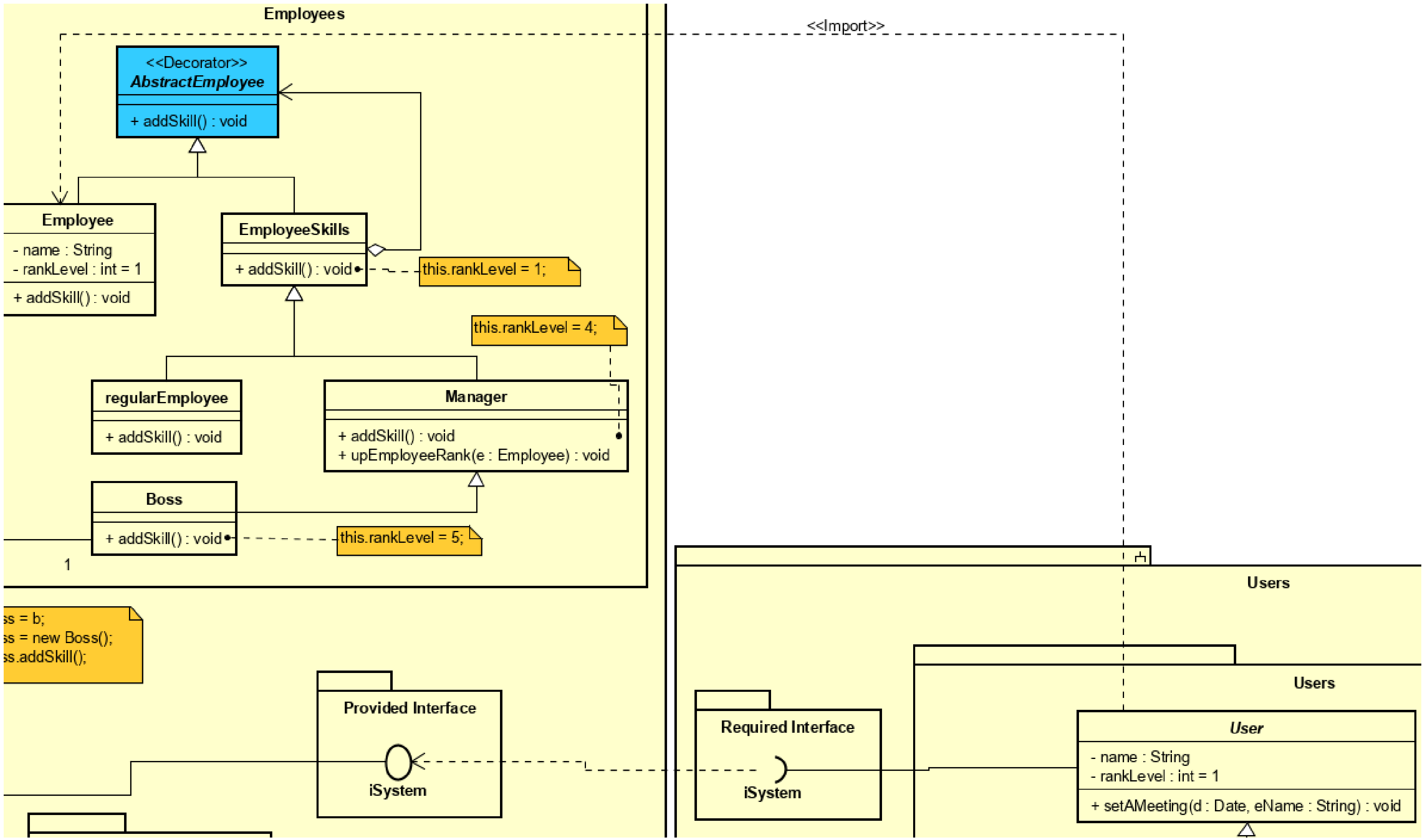
(a) the system component big picture


Company
<<Interface>>
<<Prototype>>
OfficePrototype
+ newOffice() : regularOffice
+ newHeadQuarterOffice() : HeadQuarterOffice
<<Composite>>
AbstractDepartment
+ add(object : AbstractDepartment) : void
0..*
1
<<Import>>
Office
- office : regularOffice
+ setOffice() : void
this.office = OfficePrototype.newOffice();
Department
- name : String
- offices : ArrayList<Office>
- manager : Employee
+ Department(s : String, m : Employee)
+ add(object : AbstractDepartment) : void
1..*
1
this.manager = m;
this.manager = new Manager();
this.manager.addSkill();
Employee
- name : String
- rankLevel : int
+ addSkill() : void
1..*
Implements Interface
<<Import>>
regularOffice
- name : String
- number : int
- capacity : int
- floor_number : int
- schedule : QueueList<Date>
+ newOffice() : regularOffice
+ regularOffice()
+ regularOffice(s : String, n : int, c : int, fn : int)
+ setAMeeting(d : Date, employeeName : String) : void
return new regularOffice();
<<Singleton>>
Company
- headQuarterOffice : HeadQuarterOffice
- name : String
- instance : Company
- boss : Employee
- Company()
+ getInstance() : Company
+ setHeadQuarterOffice() : void
+ setName(n : String) : void
+ setBoss(b : Boss) : void
1
1
this.boss = b;
this.boss = new Boss();
this.boss.addSkill();
Extends regularOffice
this.headQuarterOffice = OfficePrototype.newHeadQuarterOffice();
<<Façade>>
System
- instance : Company
+ getInstance() : Company
1
1
HeadQuarterOffice
- isItReadyForAMeeting : boolean = True
+ newHeadQuarterOffice() : HeadQuarterOffice
+ HeadQuarterOffice()
+ HeadQuarterOffice(s : String, n : int, c : int, fn : int)
+ cleanOffice() : void
return new HeadQuarterOffice();
Name: Gustavo Hammerschmidt.
A Java-Application Class Diagram.
A company consists of departments that are located in one or more offices. One office serves as the headquarter of the company. Each department has a manager who is recruited from the set of employees. Your task is to model the system for the company. Draw a class diagram that consists of all the classes in your system, their attributes and operations, relationships between the classes, multiplicity specifications, and other model elements that you find appropriate. (10 Points)
reg
+ a
Requi
iD

(a) the company class in the system component



(a) the employee class in the system component



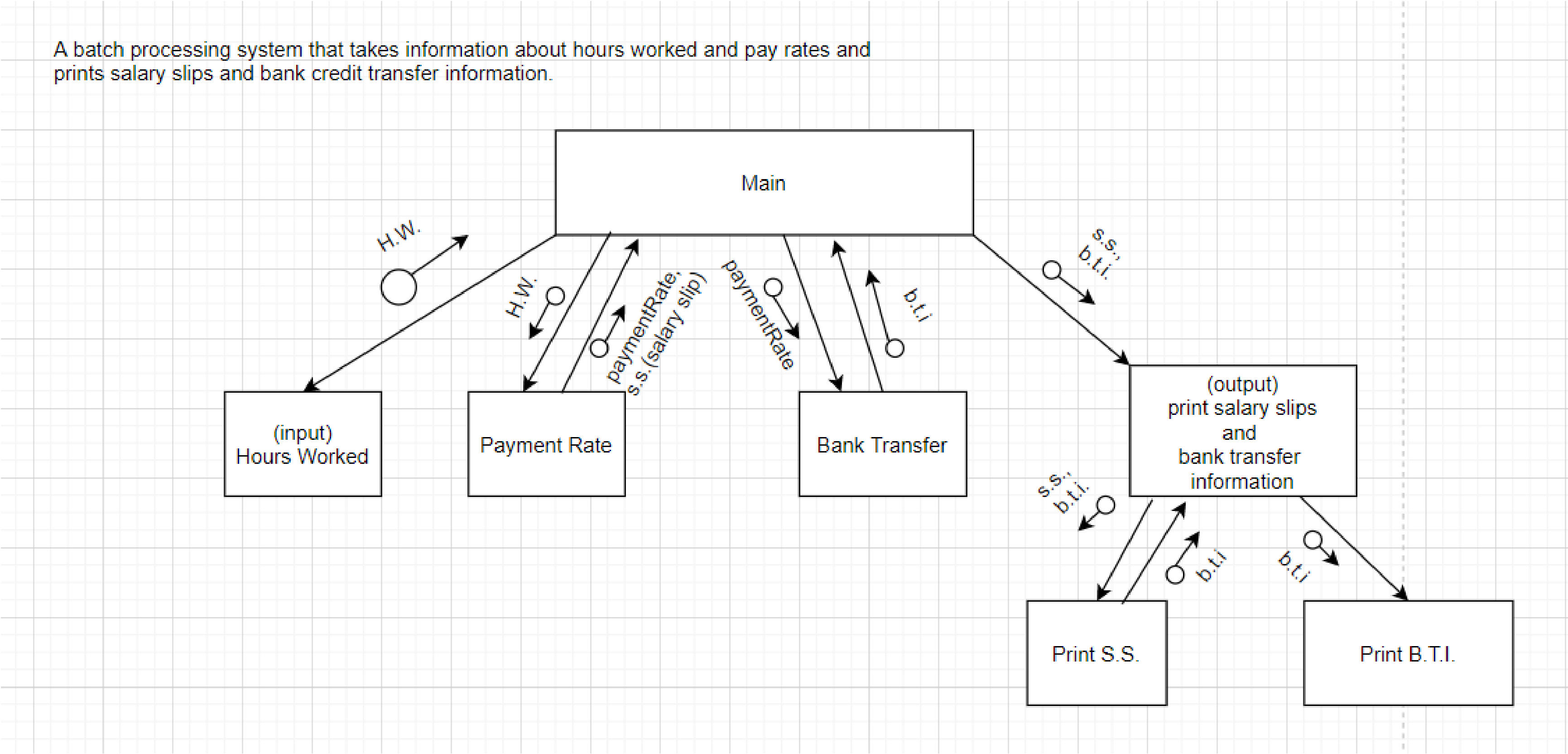
(a) A highlight to show that the user imports the employee class

## 1.2 Second Question

2. Use one of the design methods or architecture patterns and styles that we learned in this class to design each of the following systems. You need to: (1) explain the reason(s) that you choose that method/pattern/style; (2) draw the architectural diagram of the system. (30 point)

a. A batch processing system that takes information about hours worked and pay rates and prints salary slips and bank credit transfer information.

To this problem, my software design solution was based on function-oriented design because, in this problem, the data flow is more impacting on the solution, once the problem can be reduced to functions and its input is transformed on an output. To be honest, it also felt more easy to express this problem in a function-oriented design than a object-oriented one. I think that the point here is to manage the information incoming into outgoing results that will help the software user. And, by that, I mean that it is easier to do it on a more functional programming inclined environment, therefore, a function-oriented design suits better the situation. The solution diagram is given below.



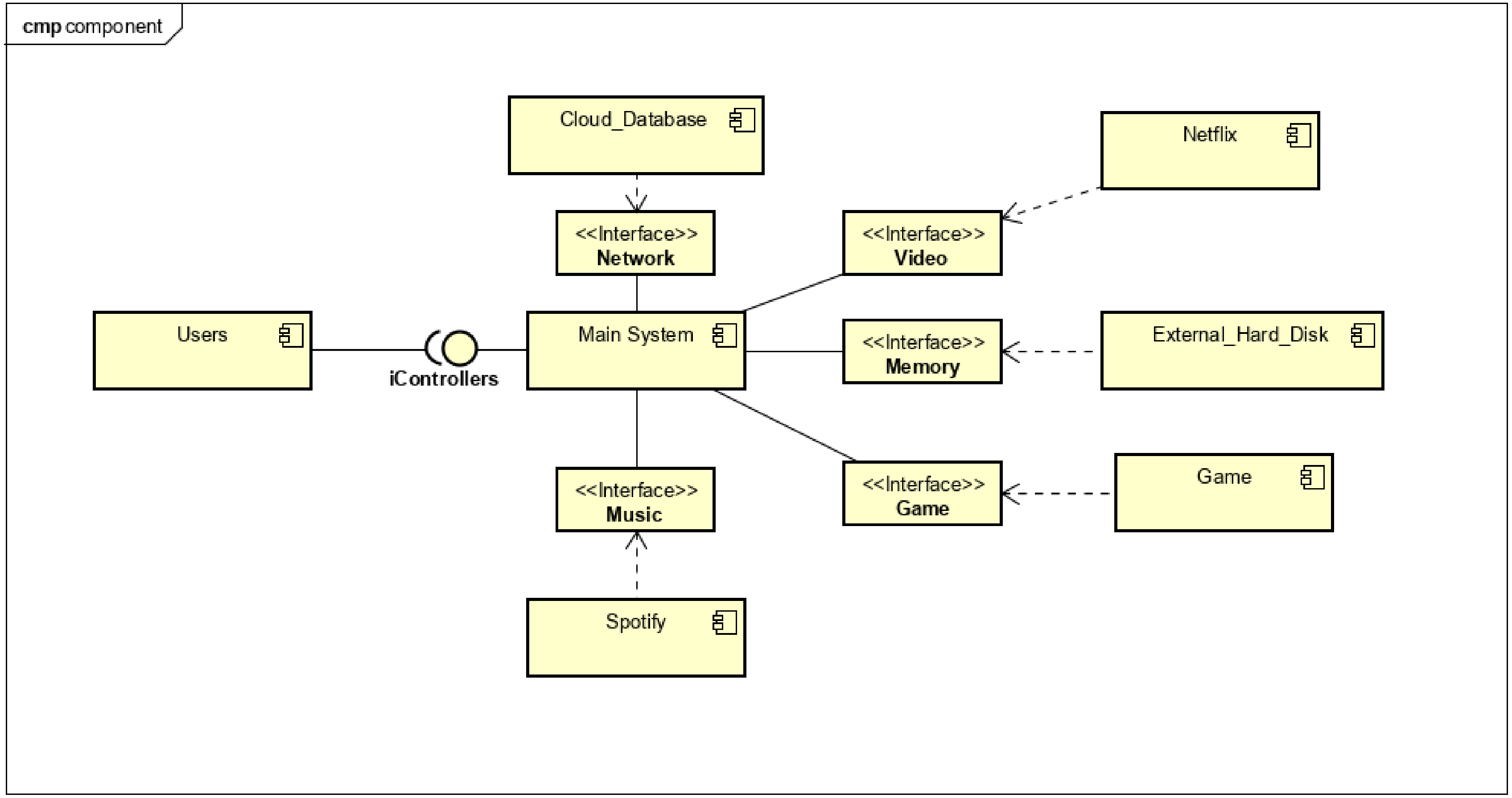
A batch processing system that takes information about hours worked and pay rates and prints salary slips and bank credit transfer information.

(a) Question 2a function-oriented diagram

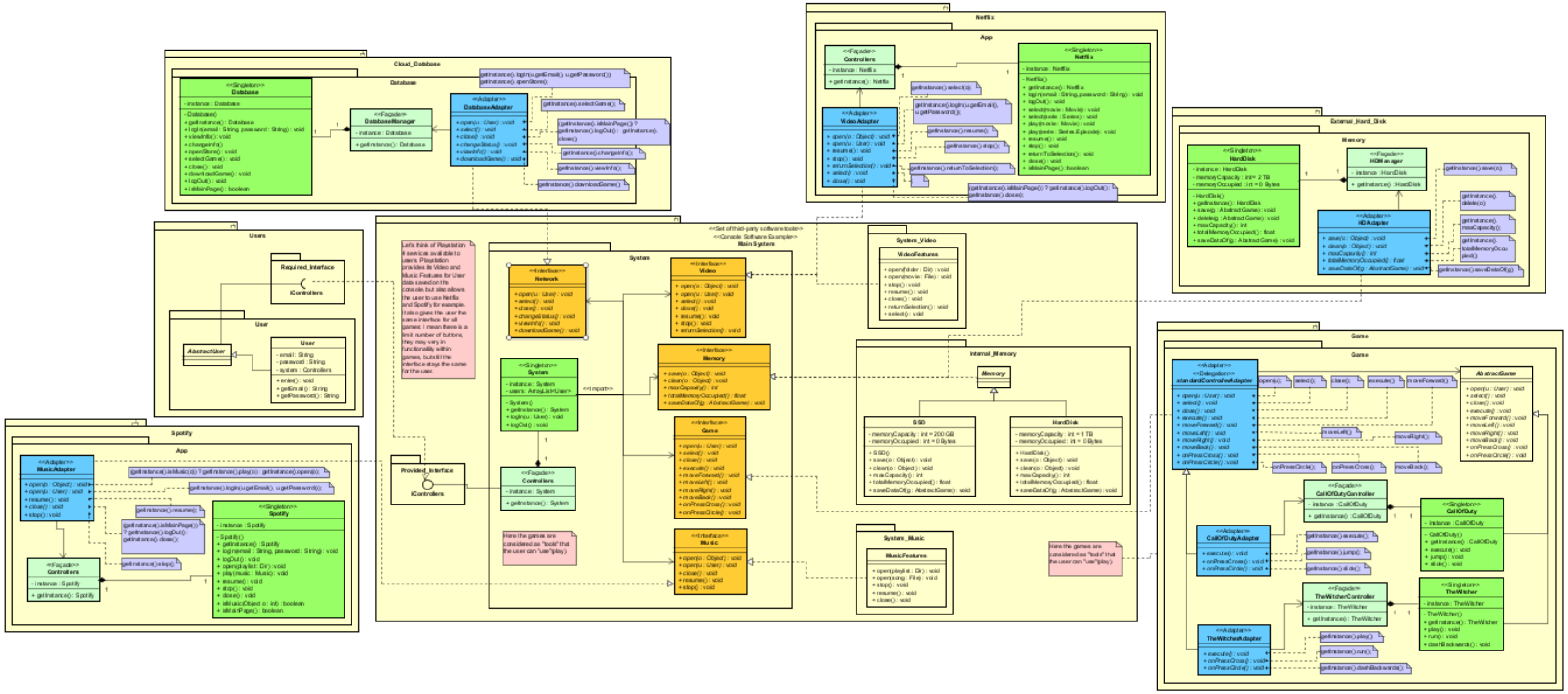
b. A set of software tools that are produced by different vendors, but which must work together.

My answer to this question is an UML diagram that does not fit into this paper, therefore, I will show a component diagram of it to clarify the space position of each component in the class diagram. I will show the whole-picture of the class diagram and separately will show each component in the diagram. Follow the caption of the images. The UML file will be posted too. I use the astah tool to create my UML files.

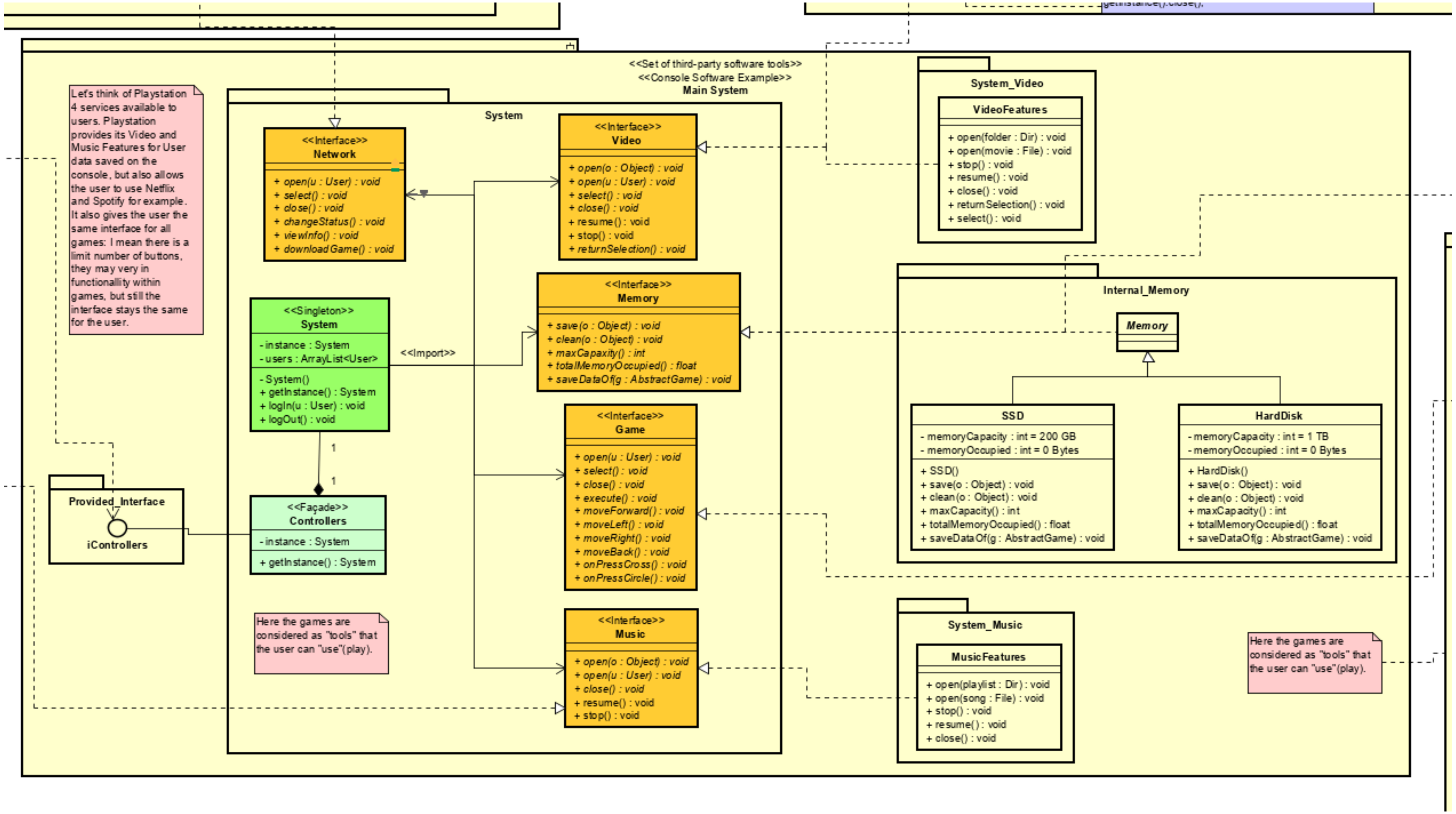
To solve this problem, one major design pattern came to head: the adapter pattern. The problem is about a system that provides compatibility to third-party software to integrate the system and provide a final product to the user that has the 3rd-party software functionalities within. To better explain my use of the adapter pattern, I used an example: the Sony console, PlayStation 4. The console provides an development tool to third-parties game companies, so they can make a version of their game to the console. The video game also has streaming services that can be connected to the console, so the user will be able to use their functionalities too. In this diagram, I've used Netflix and Spotify as examples to 3rd-party services of this console. It is worth mentioning that not only the adapter pattern was used but also singleton and façade were. The diagram is considerably large to display in this pdf file as a whole, so it was decomposed into component images. This problem is object-oriented designed software because it is easily decomposed into entities that communicate with each other. The diagram is given below.



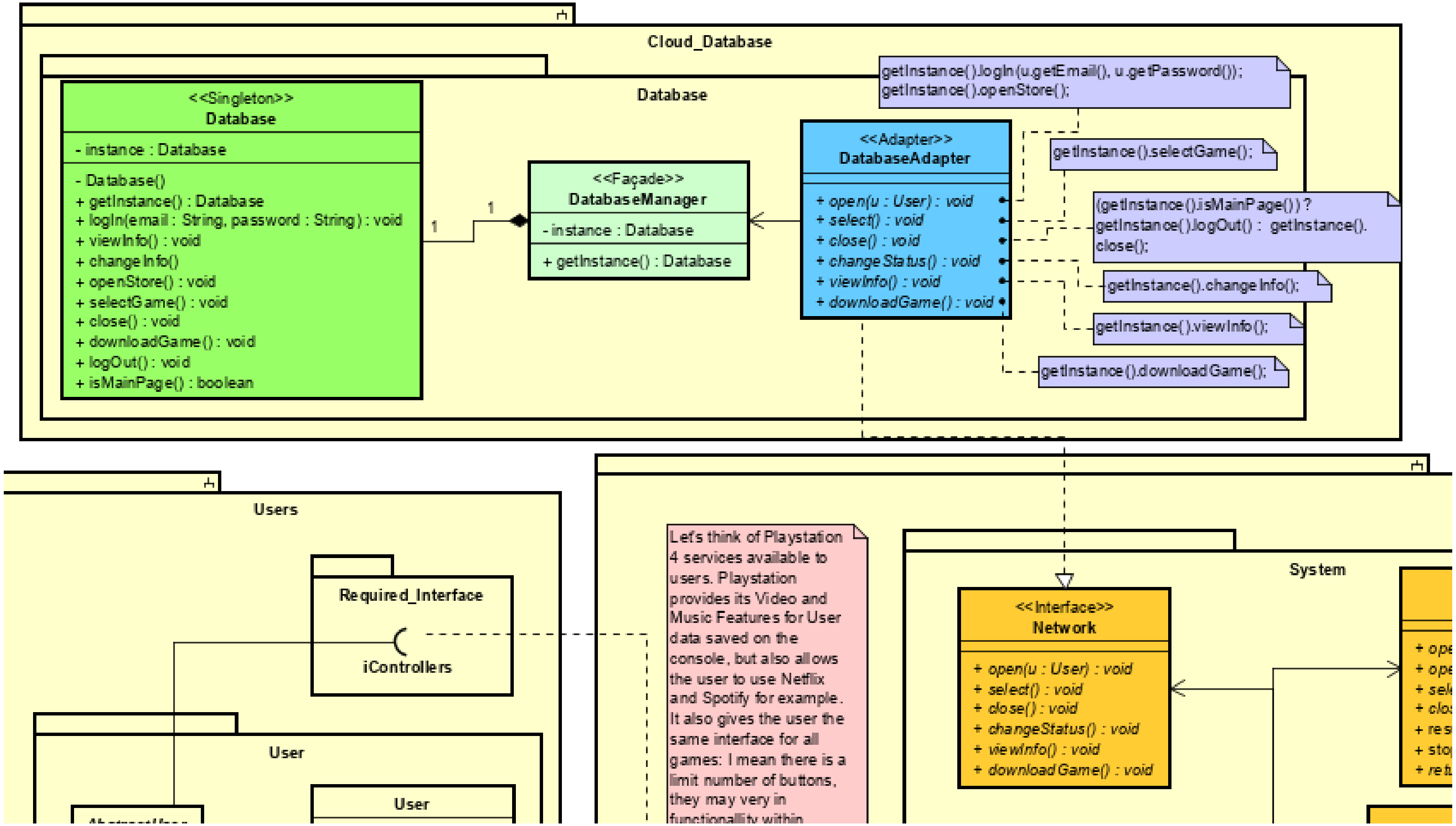
(a) The component diagram



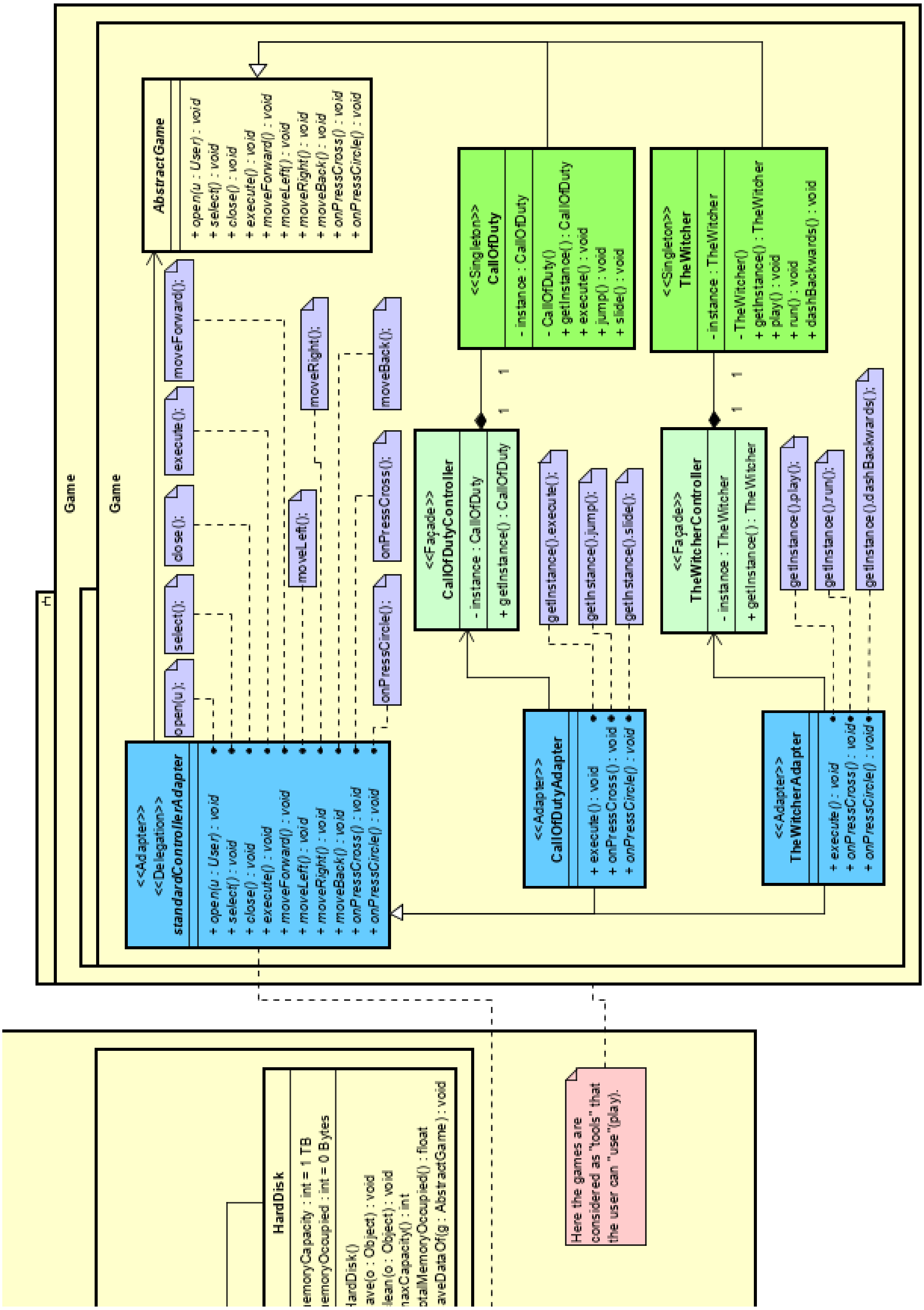
(a) The big picture of the class diagram


Let's think of Playstation 4 services available to users. Playstation provides its Video and Music Features for User data saved on the console, but also allows the user to use Netflix and Spotify for example. It also gives the user the same interface for all games: I mean there is a limit number of buttons, they may very in functionallity within games, but still the interface stays the same for the user.
<<Set of third-party software tools>>
<<Console Software Example>>
Main System
System
<<Interface>>
Network
+ open(u : User) : void
+ select() : void
+ close() : void
+ changeStatus() : void
+ viewInfo() : void
+ downloadGame() : void
<<Interface>>
Video
+ open(o : Object) : void
+ open(u : User) : void
+ select() : void
+ close() : void
+ resume() : void
+ stop() : void
+ returnSelection() : void
<<Singleton>>
System
- instance : System
- users : ArrayList<User>
- System()
+ getInstance() : System
+ logIn(u : User) : void
+ logOut() : void
<<Import>>
<<Interface>>
Memory
+ save(o : Object) : void
+ clean(o : Object) : void
+ maxCapaxity() : int
+ totalMemoryOccupied() : float
+ saveDataOf(g : AbstractGame) : void
1
1
<<Façade>>
Controllers
- instance : System
+ getInstance() : System
Provided_Interface
iControllers
<<Interface>>
Game
+ open(u : User) : void
+ select() : void
+ close() : void
+ execute() : void
+ moveForward() : void
+ moveLeft() : void
+ moveRight() : void
+ moveBack() : void
+ onPressCross() : void
+ onPressCircle() : void
Here the games are considered as "tools" that the user can "use"(play).
<<Interface>>
Music
+ open(o : Object) : void
+ open(u : User) : void
+ close() : void
+ resume() : void
+ stop() : void
System_Video
VideoFeatures
+ open(folder : Dir) : void
+ open(movie : File) : void
+ stop() : void
+ resume() : void
+ close() : void
+ returnSelection() : void
+ select() : void
Internal_Memory
Memory
SSD
- memoryCapacity : int = 200 GB
- memoryOccupied : int = 0 Bytes
+ SSD()
+ save(o : Object) : void
+ clean(o : Object) : void
+ maxCapacity() : int
+ totalMemoryOccupied() : float
+ saveDataOf(g : AbstractGame) : void
HardDisk
- memoryCapacity : int = 1 TB
- memoryOccupied : int = 0 Bytes
+ HardDisk()
+ save(o : Object) : void
+ clean(o : Object) : void
+ maxCapacity() : int
+ totalMemoryOccupied() : float
+ saveDataOf(g : AbstractGame) : void
System_Music
MusicFeatures
+ open(playlist : Dir) : void
+ open(song : File) : void
+ stop() : void
+ resume() : void
+ close() : void
Here the games are considered as "tools" that the user can "use"(play).

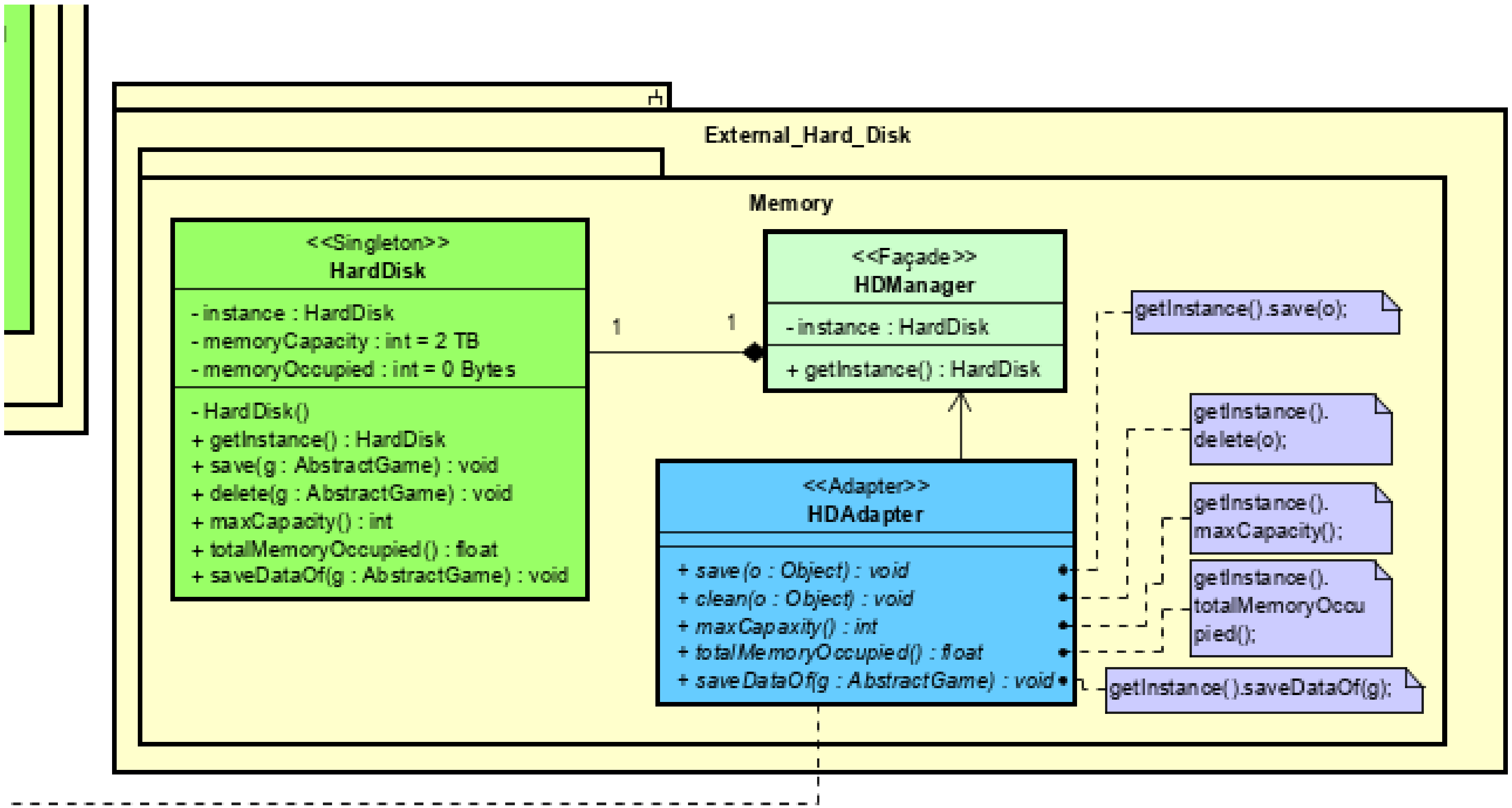
(a) The system component



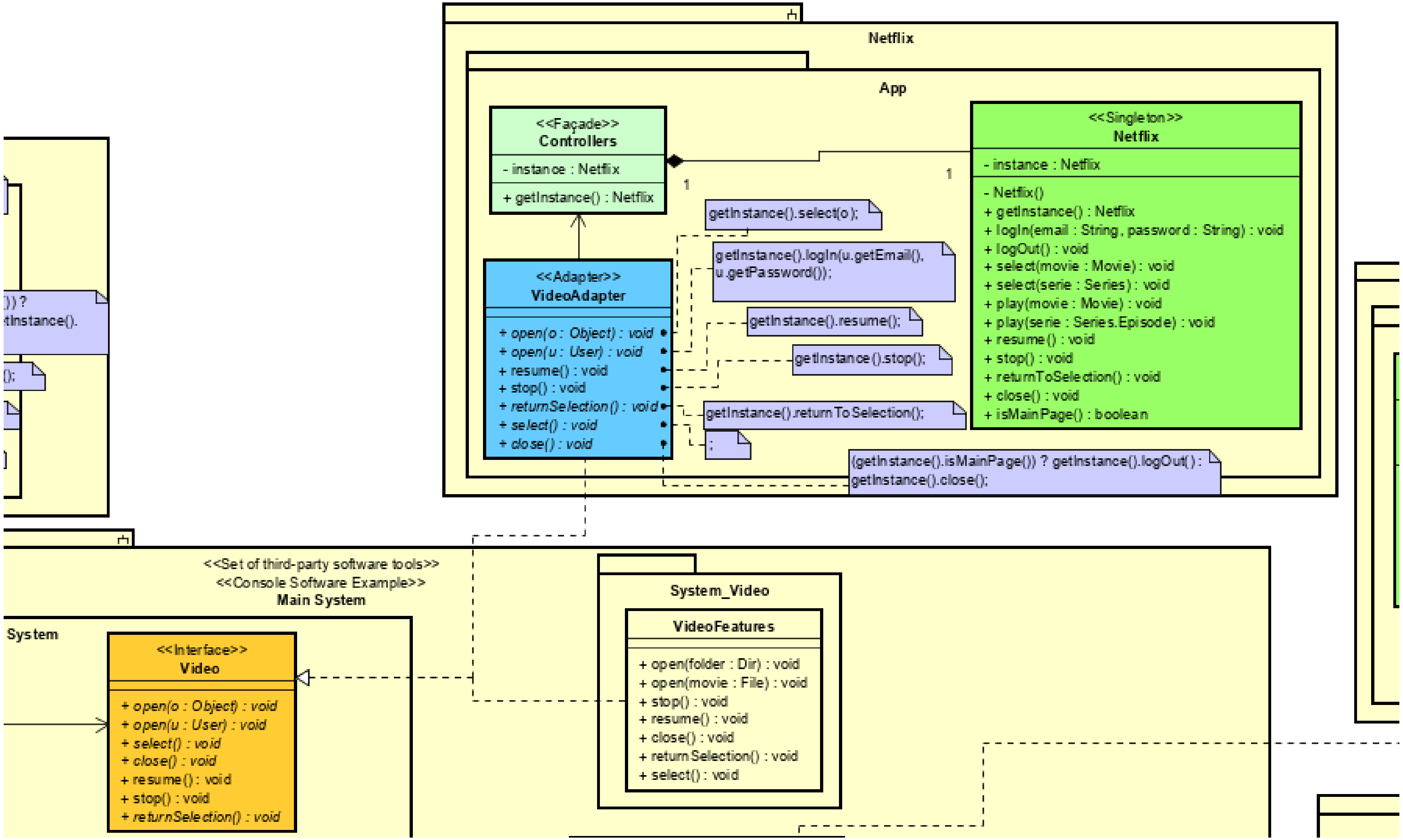
(a) The cloud component



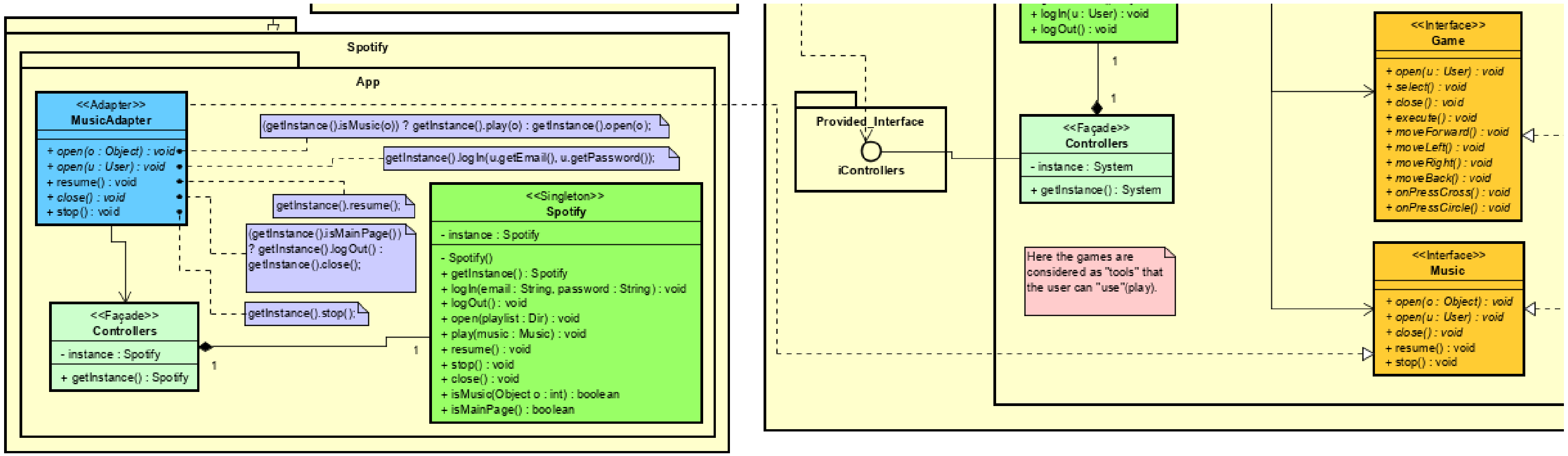
(a) The game component


External_Hard_Disk
Memory
<<Singleton>>
HardDisk
- instance : HardDisk
- memoryCapacity : int = 2 TB
- memoryOccupied : int = 0 Bytes
- HardDisk()
+ getInstance() : HardDisk
+ save(g : AbstractGame) : void
+ delete(g : AbstractGame) : void
+ maxCapacity() : int
+ totalMemoryOccupied() : float
+ saveDataOf(g : AbstractGame) : void
1
1
<<Façade>>
HDManager
- instance : HardDisk
+ getInstance() : HardDisk
<<Adapter>>
HDAdapter
+ save(o : Object) : void
+ clean(o : Object) : void
+ maxCapaxity() : int
+ totalMemoryOccupied() : float
+ saveDataOf(g : AbstractGame) : void
getInstance().save(o);
getInstance().
delete(o);
getInstance().
maxCapacity();
getInstance().
totalMemoryOccu
pied();
getInstance().saveDataOf(g);

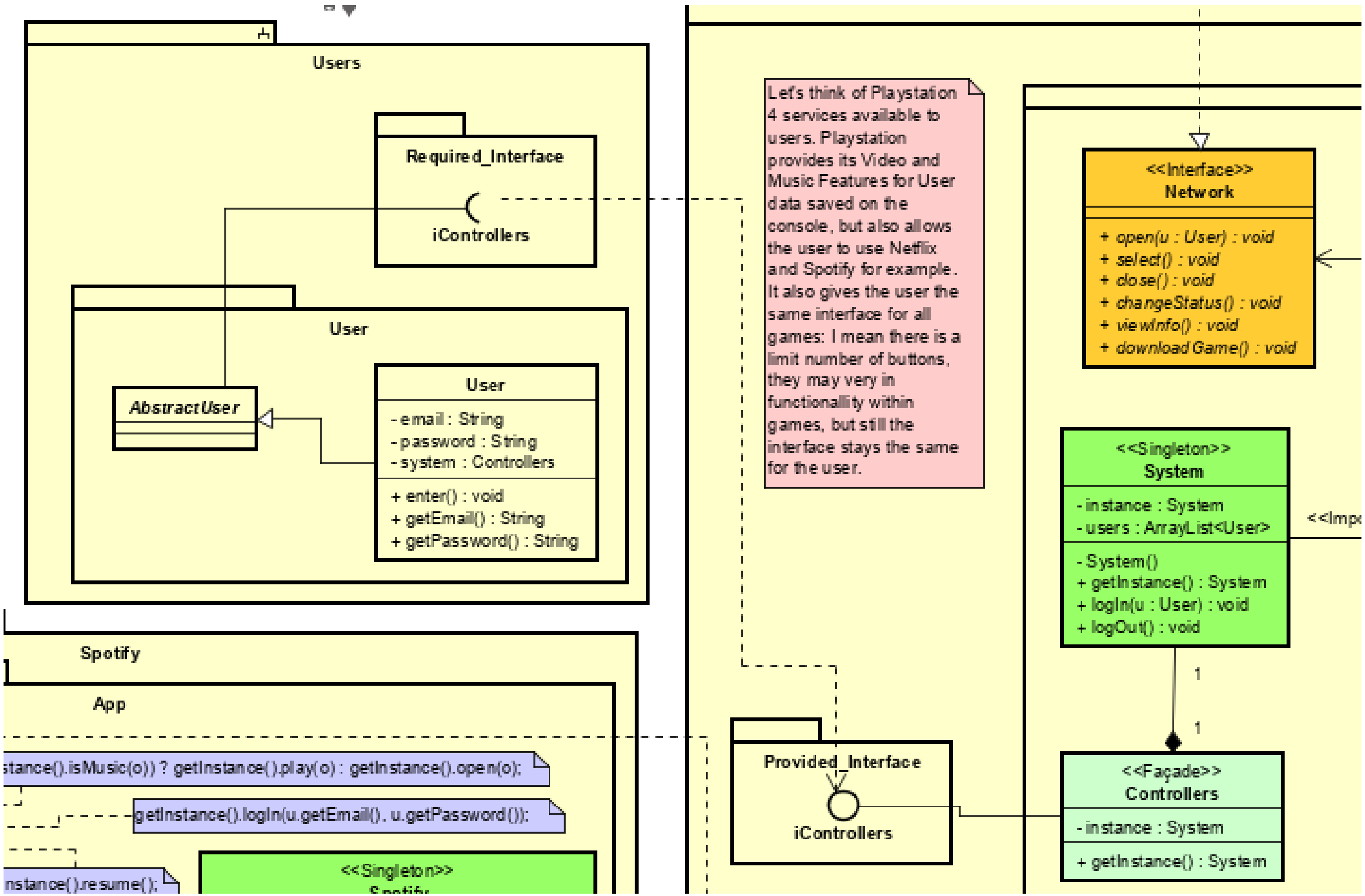
(a) The External Hard Disk component


Netflix
App
<<Façade>>
Controllers
- instance : Netflix
+ getInstance() : Netflix
<<Singleton>>
Netflix
- instance : Netflix
- Netflix()
+ getInstance() : Netflix
+ logIn(email : String, password : String) : void
+ logOut() : void
+ select(movie : Movie) : void
+ select(serie : Series) : void
+ play(movie : Movie) : void
+ play(serie : Series.Episode) : void
+ resume() : void
+ stop() : void
+ returnToSelection() : void
+ close() : void
+ isMainPage() : boolean
1
1
<<Adapter>>
VideoAdapter
+ open(o : Object) : void
+ open(u : User) : void
+ resume() : void
+ stop() : void
+ returnSelection() : void
+ select() : void
+ close() : void
getInstance().select(o);
getInstance().logIn(u.getEmail(), u.getPassword());
getInstance().resume();
getInstance().stop();
getInstance().returnToSelection();
;
(getInstance().isMainPage()) ? getInstance().logOut() : getInstance().close();
)) ?
tInstance().
();
<<Set of third-party software tools>>
<<Console Software Example>>
Main System
System
<<Interface>>
Video
+ open(o : Object) : void
+ open(u : User) : void
+ select() : void
+ close() : void
+ resume() : void
+ stop() : void
+ returnSelection() : void
System_Video
VideoFeatures
+ open(folder : Dir) : void
+ open(movie : File) : void
+ stop() : void
+ resume() : void
+ close() : void
+ returnSelection() : void
+ select() : void

(a) The Netflix component



(a) The Spotify component



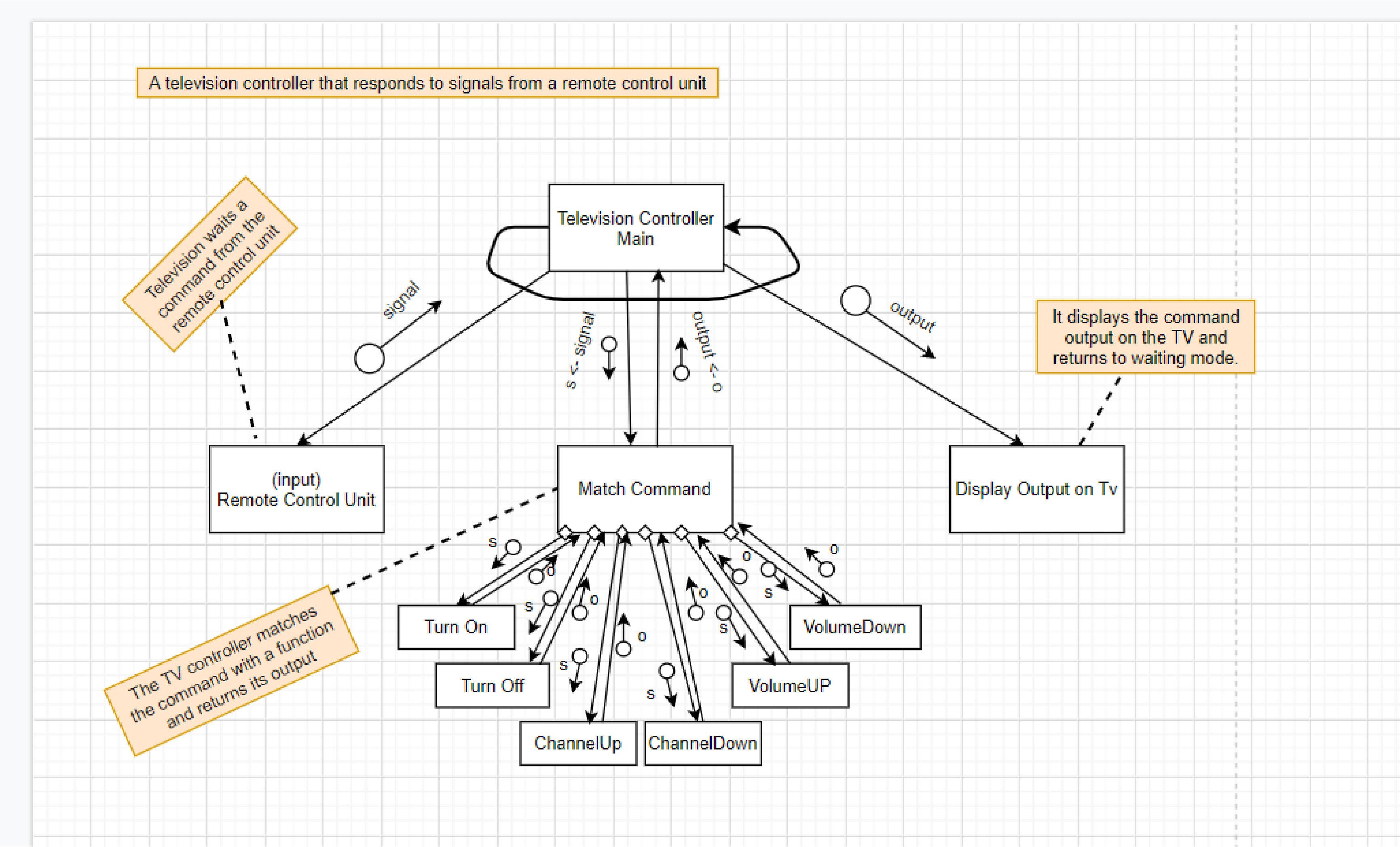
Let's think of Playstation 4 services available to users. Playstation provides its Video and Music Features for User data saved on the console, but also allows the user to use Netflix and Spotify for example. It also gives the user the same interface for all games: I mean there is a limit number of buttons, they may very in functionallity within games, but still the interface stays the same for the user.

```
stance().isMusic(o)) ? getInstance().play(o) : getInstance().open(o);
getInstance().logIn(u.getEmail(), u.getPassword());
nstance().resume();
```

(a) The user component

c. A television controller that responds to signals from a remote control unit.

To solve this problem, I thought of object-oriented design to it, but it seemed hard to define the compatibility within each part of the problem and what part exactly would communicate with another one. The function-oriented design solved the problem in a proper way, because this situation is more similar to a functional-programming problem and can be solved as one. The signal can be defined as an input that is transformed and displayed on the TV. It seems more logical to think of it as a functional procedure problem than a problem with entities communicating to respond the remote controller command. I designed it as a function-oriented software, because I couldn't identify which part of the problem would fit into an interface and would serve as a 'bridge' between the controller and the TV control system. And I think the function-oriented is the right design for this situation. The diagram is given below.


A television controller that responds to signals from a remote control unit
Television waits a command from the remote control unit
signal
Television Controller Main
s <- signal
output <- o
output
It displays the command output on the TV and returns to waiting mode.
(input) Remote Control Unit
Match Command
Display Output on Tv
The TV controller matches the command with a function and returns its output
s
o
Turn On
Turn Off
ChannelUp
ChannelDown
VolumeUP
VolumeDown

(a) Question 2c function-oriented diagram