Corso di Programmazione ad Oggetti - a.a. 2013/2014
Corso di laurea in Ingegneria e Scienze Informatiche - Università di Bologna

Development of War Games Army Builder project

The universal army builder for wargames

Group members: Luca Longobardi, Matteo Orzes.

1) Problem Analysis

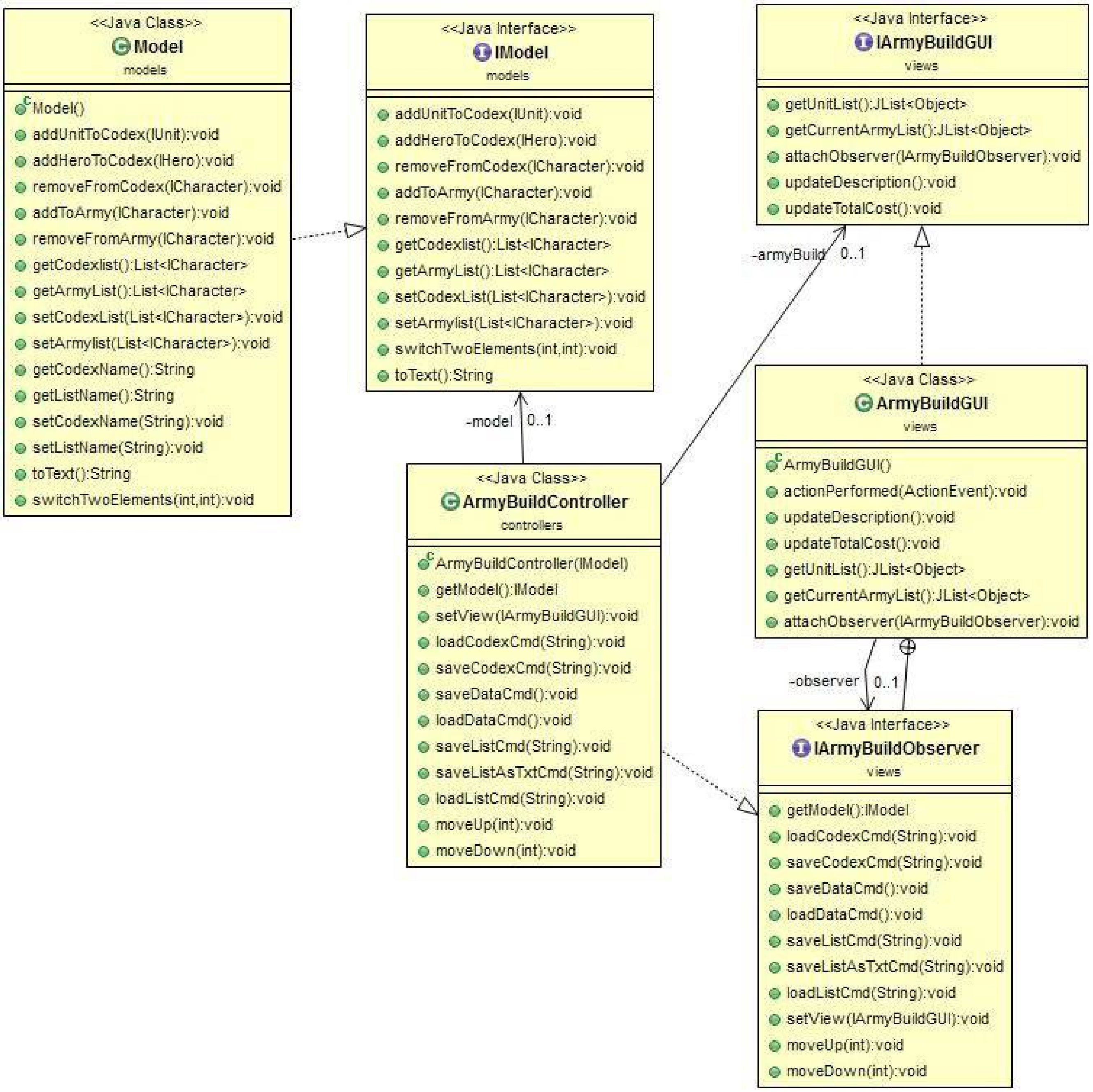
The WarGames Italian community committed an universal software for Army editing due to an increasing request by the Users. Requested functions are various.
The user must be able to:

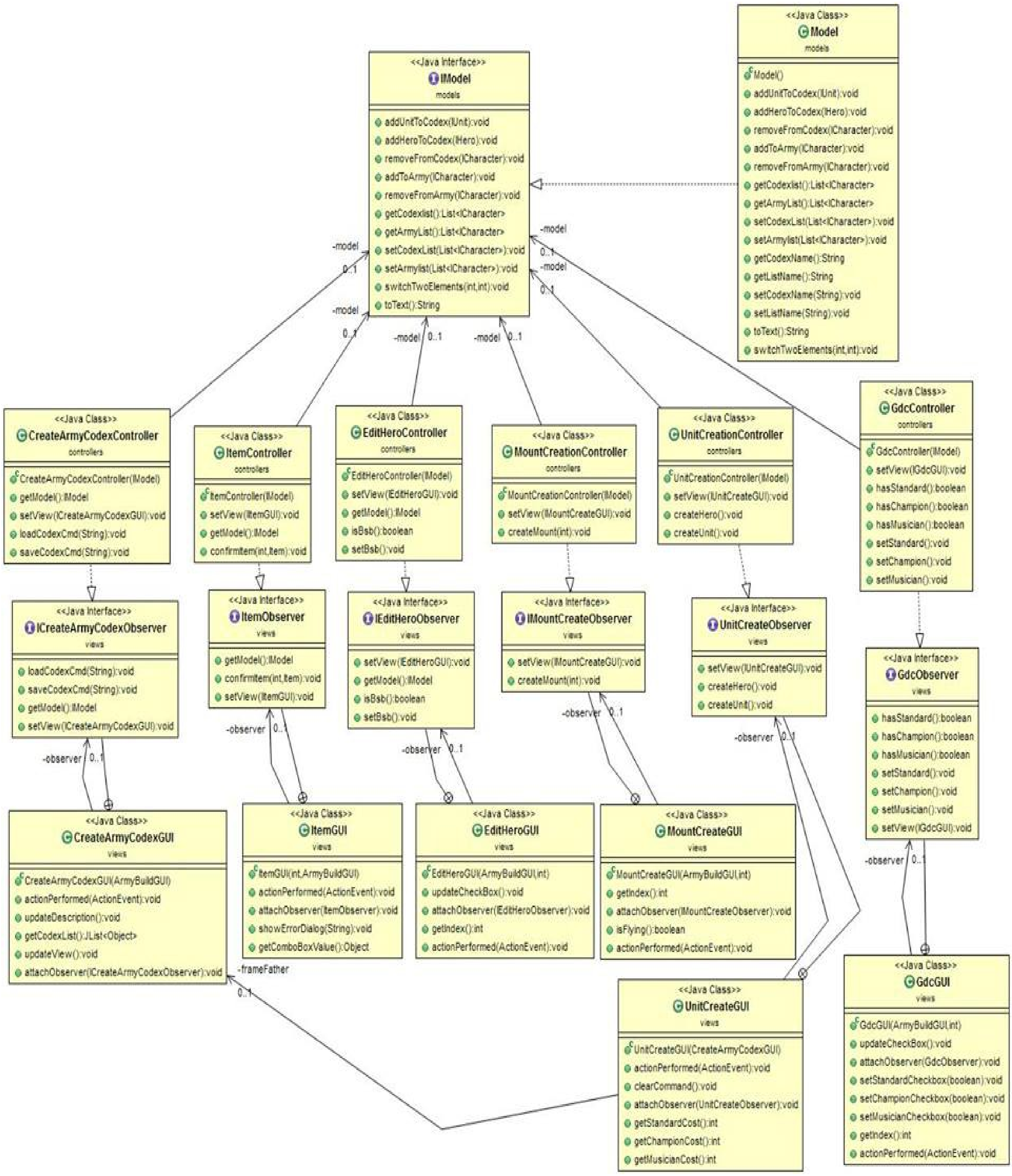
- Create a new Codex list (a list of possible character choices).
- Implement any number of Army lists for every instance of a single Codex.
- Create and delete two kind of characters, Units and Heroes(These two different types of characters must be handled in different ways).
- An unit can modify its occurrence(number of models the unit is made of), while the Hero is a single model by statement.
- Create and add any number of common items to the character. Every common item is assigned to a single model, so it multiplies its cost for the occurrence.
- Create and add magic items to the character. A single Unit can have only one magic item(defined as magic standard), indeed a hero can have any number of magic items.
- Create and add a mount to the character. Both Unit and Hero can own a single mount, which multiplies its cost for the occurrence.
- Manage the order of characters in the Army list.
- See the description of the last selected Unit.
- Edit particular features of a character(GDC for Units, BSB for Hero).
- Save and load Army lists and Codex list on disk with different extension(.CO for Codex, .AR for Army).
- Save on disk the “.txt” format for an Army list.

2) Architectural Analysis

During this phase of the project we observed and studied the application, its features and its functionalities. In particular we made a massive use of the Model_View_Controller pattern (MVC). The most important idea of this pattern is to operate a clear distinction between architectural model and the graphical user interface. The separation between these two components allow the reutilization of the view, controller and model in different applications.

We provide now a presentation about the main aspects of the application.


<<Java Class>>
Model
models
Model()
addUnitToCodex(IUnit):void
addHeroToCodex(IHero):void
removeFromCodex(ICharacter):void
addToArmy(ICharacter):void
removeFromArmy(ICharacter):void
getCodexlist():List<ICharacter>
getArmyList():List<ICharacter>
setCodexList(List<ICharacter>):void
setArmylist(List<ICharacter>):void
getCodexName():String
getListName():String
setCodexName(String):void
setListName(String):void
toText():String
switchTwoElements(int,int):void
<<Java Interface>>
IModel
models
addUnitToCodex(IUnit):void
addHeroToCodex(IHero):void
removeFromCodex(ICharacter):void
addToArmy(ICharacter):void
removeFromArmy(ICharacter):void
getCodexlist():List<ICharacter>
getArmyList():List<ICharacter>
setCodexList(List<ICharacter>):void
setArmylist(List<ICharacter>):void
switchTwoElements(int,int):void
toText():String
<<Java Interface>>
IArmyBuildGUI
views
getUnitList():JList<Object>
getCurrentArmyList():JList<Object>
attachObserver(IArmyBuildObserver):void
updateDescription():void
updateTotalCost():void
-armyBuild
0..1
-model
0..1
<<Java Class>>
ArmyBuildController
controllers
ArmyBuildController(IModel)
getModel():IModel
setView(IArmyBuildGUI):void
loadCodexCmd(String):void
saveCodexCmd(String):void
saveDataCmd():void
loadDataCmd():void
saveListCmd(String):void
saveListAsTxtCmd(String):void
loadListCmd(String):void
moveUp(int):void
moveDown(int):void
<<Java Class>>
ArmyBuildGUI
views
ArmyBuildGUI()
actionPerformed(ActionEvent):void
updateDescription():void
updateTotalCost():void
getUnitList():JList<Object>
getCurrentArmyList():JList<Object>
attachObserver(IArmyBuildObserver):void
-observer
0..1
<<Java Interface>>
IArmyBuildObserver
views
getModel():IModel
loadCodexCmd(String):void
saveCodexCmd(String):void
saveDataCmd():void
loadDataCmd():void
saveListCmd(String):void
saveListAsTxtCmd(String):void
loadListCmd(String):void
setView(IArmyBuildGUI):void
moveUp(int):void
moveDown(int):void



<<Java Interface>>
IModel
models
addUnitToCodex(IUnit):void
addHeroToCodex(IHero):void
removeFromCodex(ICharacter):void
addToArmy(ICharacter):void
removeFromArmy(ICharacter):void
getCodexlist():List<ICharacter>
getArmyList():List<ICharacter>
setCodexList(List<ICharacter>):void
setArmylist(List<ICharacter>):void
switchTwoElements(int,int):void
toText():String
<<Java Class>>
Model
models
Model()
addUnitToCodex(IUnit):void
addHeroToCodex(IHero):void
removeFromCodex(ICharacter):void
addToArmy(ICharacter):void
removeFromArmy(ICharacter):void
getCodexlist():List<ICharacter>
getArmyList():List<ICharacter>
setCodexList(List<ICharacter>):void
setArmylist(List<ICharacter>):void
getCodexName():String
getListName():String
setCodexName(String):void
setListName(String):void
toText():String
switchTwoElements(int,int):void
-model
0..1
<<Java Class>>
CreateArmyCodexController
controllers
CreateArmyCodexController(IModel)
getModel():IModel
setView(ICreateArmyCodexGUI):void
loadCodexCmd(String):void
saveCodexCmd(String):void
<<Java Class>>
ItemController
controllers
ItemController(IModel)
setView(ItemGUI):void
getModel():IModel
confirmItem(int,Item):void
<<Java Class>>
EditHeroController
controllers
EditHeroController(IModel)
setView(EditHeroGUI):void
getModel():IModel
isBsb():boolean
setBsb():void
<<Java Class>>
MountCreationController
controllers
MountCreationController(IModel)
setView(IMountCreateGUI):void
createMount(int):void
<<Java Class>>
UnitCreationController
controllers
UnitCreationController(IModel)
setView(IUnitCreateGUI):void
createHero():void
createUnit():void
<<Java Class>>
GdcController
controllers
GdcController(IModel)
setView(IGdcGUI):void
hasStandard():boolean
hasChampion():boolean
hasMusician():boolean
setStandard():void
setChampion():void
setMusician():void
<<Java Interface>>
ICreateArmyCodexObserver
views
loadCodexCmd(String):void
saveCodexCmd(String):void
getModel():IModel
setView(ICreateArmyCodexGUI):void
<<Java Interface>>
ItemObserver
views
getModel():IModel
confirmItem(int,Item):void
setView(ItemGUI):void
<<Java Interface>>
IEditHeroObserver
views
setView(EditHeroGUI):void
getModel():IModel
isBsb():boolean
setBsb():void
<<Java Interface>>
IMountCreateObserver
views
setView(IMountCreateGUI):void
createMount(int):void
<<Java Interface>>
UnitCreateObserver
views
setView(IUnitCreateGUI):void
createHero():void
createUnit():void
<<Java Interface>>
GdcObserver
views
hasStandard():boolean
hasChampion():boolean
hasMusician():boolean
setStandard():void
setChampion():void
setMusician():void
setView(IGdcGUI):void
-observer
0..1
<<Java Class>>
CreateArmyCodexGUI
views
CreateArmyCodexGUI(ArmyBuildGUI)
actionPerformed(ActionEvent):void
updateDescription():void
getCodexList():JList<Object>
updateView():void
attachObserver(ICreateArmyCodexObserver):void
<<Java Class>>
ItemGUI
views
ItemGUI(int,ArmyBuildGUI)
actionPerformed(ActionEvent):void
attachObserver(ItemObserver):void
showErrorDialog(String):void
getComboBoxValue():Object
<<Java Class>>
EditHeroGUI
views
EditHeroGUI(ArmyBuildGUI,int)
updateCheckBox():void
attachObserver(IEditHeroObserver):void
getIndex():int
actionPerformed(ActionEvent):void
<<Java Class>>
MountCreateGUI
views
MountCreateGUI(ArmyBuildGUI,int)
getIndex():int
attachObserver(IMountCreateObserver):void
isFlying():boolean
actionPerformed(ActionEvent):void
-frameFather
0..1
<<Java Class>>
UnitCreateGUI
views
UnitCreateGUI(CreateArmyCodexGUI)
actionPerformed(ActionEvent):void
clearCommand():void
attachObserver(UnitCreateObserver):void
getStandardCost():int
getChampionCost():int
getMusicianCost():int
<<Java Class>>
GdcGUI
views
GdcGUI(ArmyBuildGUI,int)
updateCheckBox():void
attachObserver(GdcObserver):void
setStandardCheckbox(boolean):void
setChampionCheckbox(boolean):void
setMusicianCheckbox(boolean):void
getIndex():int
actionPerformed(ActionEvent):void

3) Package Organization

controllers: this package contains the source codes about controllers related to ArmyBuildGUI and all the others GUIs.

models: this package contains the source codes used to implement the model of the application.

views: this package contains the source codes used to implement all the GUIs used in the application.

test: this package contains the source codes (Junit) used for the application testing.

modifiedpanels: this package contains two source codes used to implement two JPanels frequently used in the view.

exceptions: this package contains the exceptions used in the application context.

main: this package contains only the main class of the application.

4) Working Division

In the context of the project realization the working procedure has been divided following this scheme.

- Luca Longobardi developed the model of the application.
- Matteo Orzes developed the view of the application.

Some parts of the project have been realized in common:

- The controller of the application has been developed together.
- The view appearance/design has been decided together before the start of the project.

Detailed Planning: Luca Longobardi

This single part of the Project Relation is responsibility of the single member of the group: Luca Longobardi.

Detailed description of the Model

The model represents the whole data management by the Army Builder application.

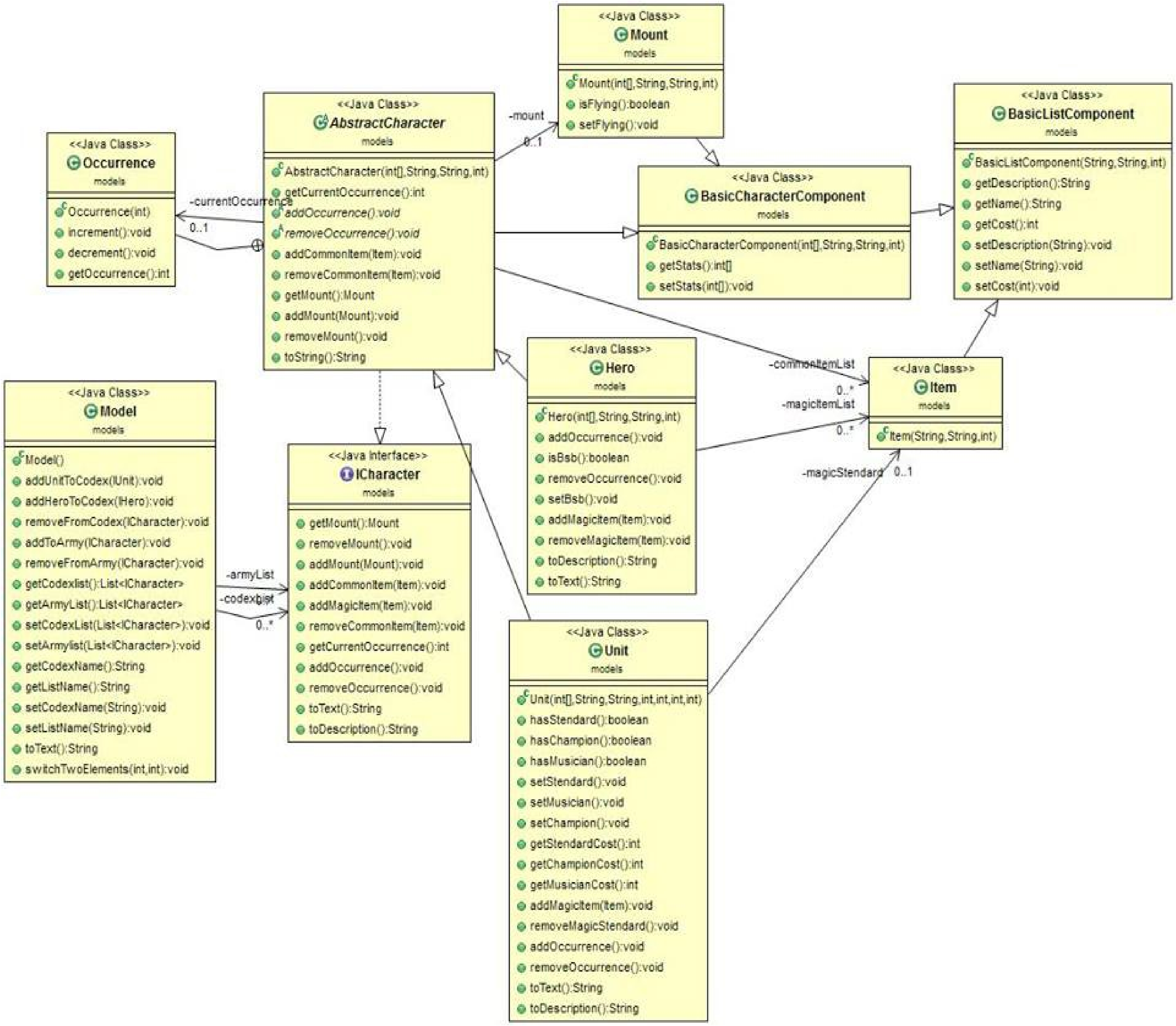
- BasicListComponent class models the basic characteristics that a single element must have to be part of an Army List or a Codex List. In our case, a list component can be a hero, an unit, an item, a mount or any object that extends this class. It is composed by a name, a description and a cost, with their accessor methods.
- Item directly extends BasicListComponent, simply because it does not add any further information. There are two kinds of items: magic and common. We decided not to implement this feature in this class, since the instance "magic" of "common" does not add any important information about the item. It is just managed differently when assigned to a character. I have chosen to represent the concept of item with a different class so that in the future, if needed, we will add more features. NOTE: wargames change their rules approximately every four years. This kind of hierarchy helps code retention.
- BasicCharacterComponent class extends the concept of basic list component, modelling the frame for the real concept of Character(described below). It adds the statistics concept, which is implemented by an array with a certain default length(in our case 9). Each stat cannot exceed the upper bound 10 or the lower bound 0, and an exception is thrown if the vector does not match the defined length or one or more stat is not legit.
- Mount class directly extends BasicCharacterComponent. It only adds the concept of flying or not flying to the class. A mount, exactly as an item, can be bought by an unit or a hero.
- AbstractCharacter is the core of the entire model. It represents the Pattern Template for classes Hero and Unit described below. This class manages:
  1. ownership of common item and mount. Since a common item or a mount adds a certain cost to every single occurrence of the character(depending on the cost of the object itself), implements a set of private methods for cost management(which is automatically calculated for every occurrence change). Common items are managed with a linked list, since a single character can have more than one of them, where a mount is implemented with a single field, representing the fact that a mount is unique for every instance of a character.
  2. ad-hoc toString that perfectly represents the character in the JList view. A factory method(getAlignedString) provides the string alignment to avoid string disalignement, caused by different lengths of characters names.
  3. Abstract methods addOccurrence and removeOccurrence, which are implemented in classes Hero and Unit. We have chosen this pattern to represent that only certain characters can increment of decrement their occurrence(for example in our case only an unit, which is composed by more characters, can increment his occurrence, where a hero is always a single element).

4. a nested class which provides the necessary primitives to organize occurrence management. An instance of this class composes a field of AbstractCharacter.

- Hero class represents the concept of hero. By extending AbstractCharacter, this class only implements the management of Magic item and Battle Standard Bearer(BSB). A hero can have any number of magic items, so I decided to implement this concept with a linked list. Battle standard bearer is a value that can assume true or false, so it is implemented with a boolean and his accessor methods. When an Hero is set BSB, it adds to his cost a certain quantity defined by default. NOTE: as I said before, a hero cannot modify his occurrence. In other words when the user tries to modify it, an exception is thrown.
- Unit class represents the concept of unit. By extending AbstractCharacter, this class implements the management of Command group and Magic Standard.
  Command group is composed by three booleans: Standard, Champion and Musician. Each of these fields have accessor methods, and every time they are set true, a cost is added to the total of the unit. This cost is not set by default, but is decided when the class is instanced.
  On the contrary of an Hero, an Unit can have a single magic item, represented by its magic standard(implemented with a single field and his accessor methods). Obviously when a magic item is assigned, if the Standard field is false, is automatically set true.
- Model class representing the whole data managed by the Army Builder application. This class provides:
  1. Two linked lists, one for the Codex List and one for the Army list.
  2. Primitives to add and remove characters from Codex List. In this list characters cannot be duplicated, or an exception Is thrown.
  3. Primitives to add and remove characters from Army List. In this list characters can be duplicated, but you cannot add a character that is not included in the Codex List.
  4. Primitives to get and set Codex and Army Lists, which are used during the load/save operations.
  5. Primitives to set Codex and Army Lists names.

Final Notes: All classes that descend from AbstractCharacter have a toDescription and a toText method. The first is used in the view classes to print information about the character in a square box, and the second is used when the user saves his current Army list in TXT format.
All classes have private methods to check null pointers and other inconsistency problems. In those cases NullFieldException and Others are thrown.

Following an UML about the model organization.


<<Java Class>>
Mount
models
Mount(int[],String,String,int)
isFlying():boolean
setFlying():void
<<Java Class>>
AbstractCharacter
models
AbstractCharacter(int[],String,String,int)
getCurrentOccurrence():int
addOccurrence():void
removeOccurrence():void
addCommonItem(Item):void
removeCommonItem(Item):void
getMount():Mount
addMount(Mount):void
removeMount():void
toString():String
<<Java Class>>
Occurrence
models
Occurrence(int)
increment():void
decrement():void
getOccurrence():int
-currentOccurrence
0..1
-mount
0..1
<<Java Class>>
BasicCharacterComponent
models
BasicCharacterComponent(int[],String,String,int)
getStats():int[]
setStats(int[]):void
<<Java Class>>
BasicListComponent
models
BasicListComponent(String,String,int)
getDescription():String
getName():String
getCost():int
setDescription(String):void
setName(String):void
setCost(int):void
<<Java Class>>
Hero
models
Hero(int[],String,String,int)
addOccurrence():void
isBsb():boolean
removeOccurrence():void
setBsb():void
addMagicItem(Item):void
removeMagicItem(Item):void
toDescription():String
toText():String
-commonItemList
0..*
-magicItemList
0..*
<<Java Class>>
Item
models
Item(String,String,int)
-magicStendard
0..1
<<Java Class>>
Model
models
Model()
addUnitToCodex(IUnit):void
addHeroToCodex(IHero):void
removeFromCodex(ICharacter):void
addToArmy(ICharacter):void
removeFromArmy(ICharacter):void
getCodexlist():List<ICharacter>
getArmyList():List<ICharacter>
setCodexList(List<ICharacter>):void
setArmylist(List<ICharacter>):void
getCodexName():String
getListName():String
setCodexName(String):void
setListName(String):void
toText():String
switchTwoElements(int,int):void
-armyList
-codexList
0..*
<<Java Interface>>
ICharacter
models
getMount():Mount
removeMount():void
addMount(Mount):void
addCommonItem(Item):void
addMagicItem(Item):void
removeCommonItem(Item):void
getCurrentOccurrence():int
addOccurrence():void
removeOccurrence():void
toText():String
toDescription():String
<<Java Class>>
Unit
models
Unit(int[],String,String,int,int,int,int)
hasStendard():boolean
hasChampion():boolean
hasMusician():boolean
setStendard():void
setMusician():void
setChampion():void
getStendardCost():int
getChampionCost():int
getMusicianCost():int
addMagicItem(Item):void
removeMagicStendard():void
addOccurrence():void
removeOccurrence():void
toText():String
toDescription():String

Detailed Planning: Matteo Orzes

This single part of the Project Relation is responsibility of the single member of the group: Matteo Orzes.

Detailed Description of the View.

The View shows the data stored in the model and offers the user multiple ways to interact with them.

- **BasicGUI** is a simple class that is extended by almost all the others classes, it implements the method showErrorDialog that is used by all the GUIs in order to show an error dialog when some exceptions have been generated because of an error in the application (wrong input of stats, trying to operate without selecting a unit etc.).
- **ArmyBuildGUI** is the main GUI and the most important one. This GUI allows the user to first load an existing codex and then use it to create various army lists. The functionalities offered by this GUI are the following ones:
    - From the MainMenu the user can load and save both codex and armylist and also Txt format of the current army list.
    - After having selected a unit from the codex its possible to add it to the current army list using the AddButton, its not possible to Delete a unit from the codex (if the user need to do this operation he is allowed to operate this way using the CreateArmyCodexGUI), indeed its possible to delete a unit from the current army list .
    - After having selected a unit from the current army list its possible to modify its occurrence (the number of models it contains) using Increment and decrement number Buttons, this is possible only if the unit selected is not an hero.
    - Using the UnitUp and UnitDown Buttons the user is able to move the units in the current army list in order to create the desired order.
    - If the user wants to add an item to a unit he must select first a unit from the current army list and then he is allowed to create the new Item in ItemGUI. Remember that a unit can have a single magic object while an hero can own more than one.
    - Using the button Edit the user can manage important information about the state of the last selected unit in current army list. (Gdc management for a unit and Bsb for an Hero).
    - The AddMount button is logically used to add a mount to the selected unit. In order to create the new mount the user will operate in a Mount Create GUI.
    - Removing a mount is a very important feature in a war game because a unit can have only a single type of mount. This operation is performed by remove mount Button.
- **CreateArmyCodexGUI** is the GUI that offers the user the possibility to create a new codex or modify an existing one. From this GUI its possible to delete units from the codex or create new ones using the AddNew button. In order to create a new unit a UnitCreateGUI will be used. From the Menu of this GUI the user can save and load codex.
- **AbstractUnitCreateGUI** is an important abstract class that is extended by both **UnitCreateGUI** and **MountCreateGUI,** merely looking at design and appearance these two class create really

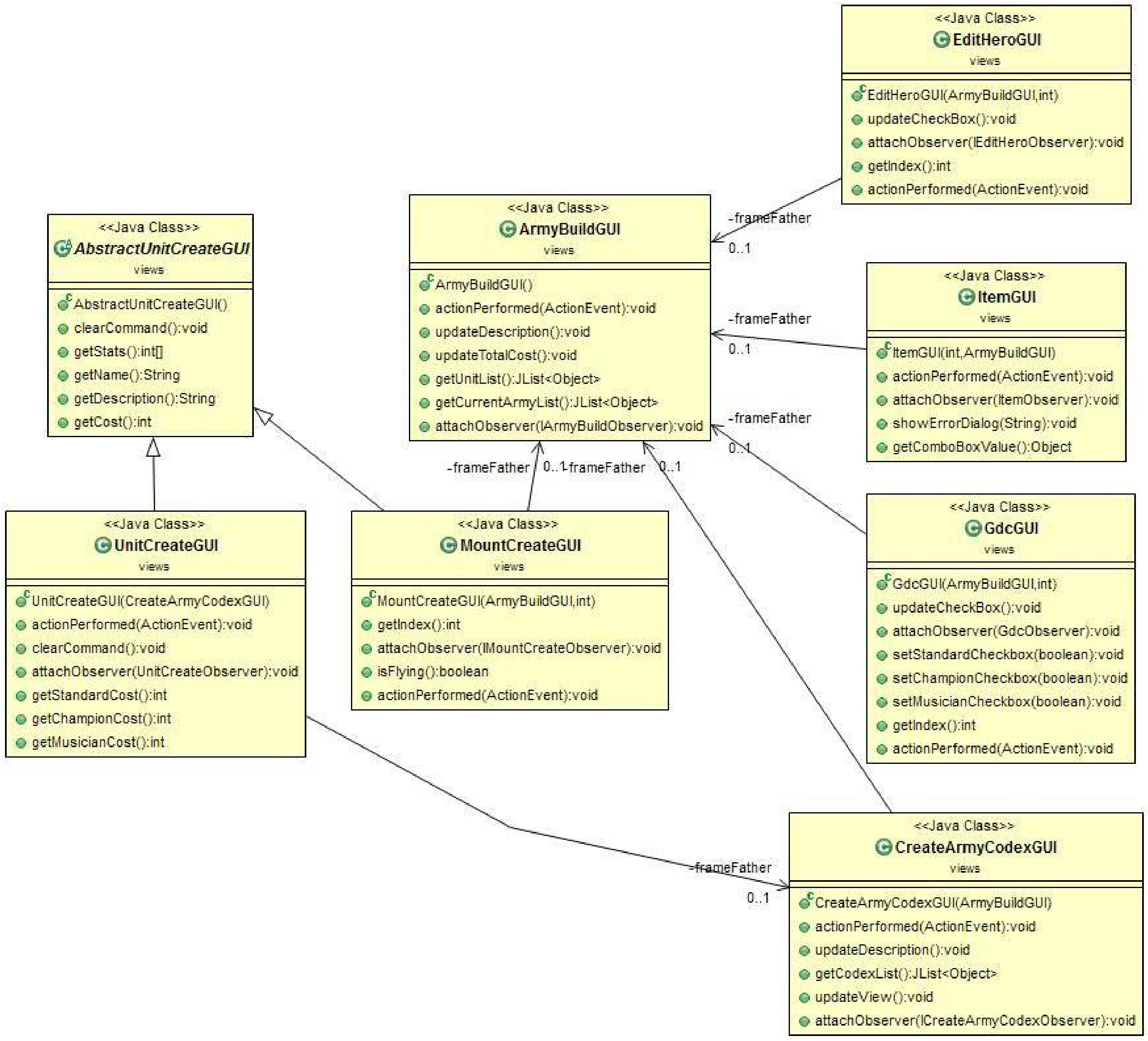
similar frames so I avoided code duplication inserting the common part in this abstract class. This class uses a boxLayout for the INSERTSTATSPANEL that allows the user to insert a numeric stat in the textfields, than the user will insert both name and cost of the unit in the INSERTNAMENCOSTPANEL. The remaining part of the GUI is the one that differs and its created in **UnitCreateGUI** and **MountCreateGUI.**

- **UnitCreateGUI** is the GUI that the user will use after having pressed the addNew button in CreateArmyCodexGUI. This GUI that extends AbstractUnitCreateGUI adds a panel to manage the Gdc Cost of the unit. This GUI is also used to create new Heroes, if the ComboBox is set on "Hero" the Gdc Panel will be disabled. Pressing Confirm button the information are taken from all the textfields and a new unit/hero is created.
- Really similar to UnitCreateGUI is **MountCreateGUI**. This frame allows the user to create a new mount for the last selected unit/hero from current army list in ArmyBuildGUI. This class is really similar to UnitCreateGUI and it extends the same abstract class, the Gdc management panel in UnitCreateGUI is replaced by a panel where the user can state if the new mount its or not flying. Pressing Confirm button the information are taken from all the textfields and a new mount is assign to the last selected unit/hero.
- **EditHeroGUI** is a rather simple frame, the only thing that the user can do with this frame is to state if the hero is or not the Bsb. Using a frame for only this can be judged a wrong choice but the basic rules for WarGames are changed every four years and the next major change will be the next year so looking at future I decided to create a separate frame because I am pretty confident that the EditHero options in future will grow significantly .
- **GdcGUI,** this simple frame just offers the user the power to manage the unit Gdc (this frame is opened by pressing the edit button while a unit is selected, if an hero is selected the editHero frame will be opened instead of this one). The user using three checkboxes is able to state if the last selected Unit in current army list (from ArmyBuildGUI) has or not a specified member of the Gdc (command group).
- **ItemGUI** is the frame that is used in order to create and assign a new item to the unit selected before opening this GUI. The user must insert name cost description and set the comboBox in order to specify if the item its or not a magical one. When the confirm button is pressed the item is created and assigned.

Notes:

All the GUIs own a field frameFather of the type of the GUI from which they has been generated. This field is used in order to re-enable the frame that has been disabled from the creation of this ones when the current frame get closed. Some other frames has as second parameter an int Index that is used in order to know the precise position of the last selected unit/hero in the proper list in frameFather. I decided to use the Observer pattern so every GUI has a proper nested interface **observer** that is then used to implement the controller.

Following an UML showing view's classes and their interactions.

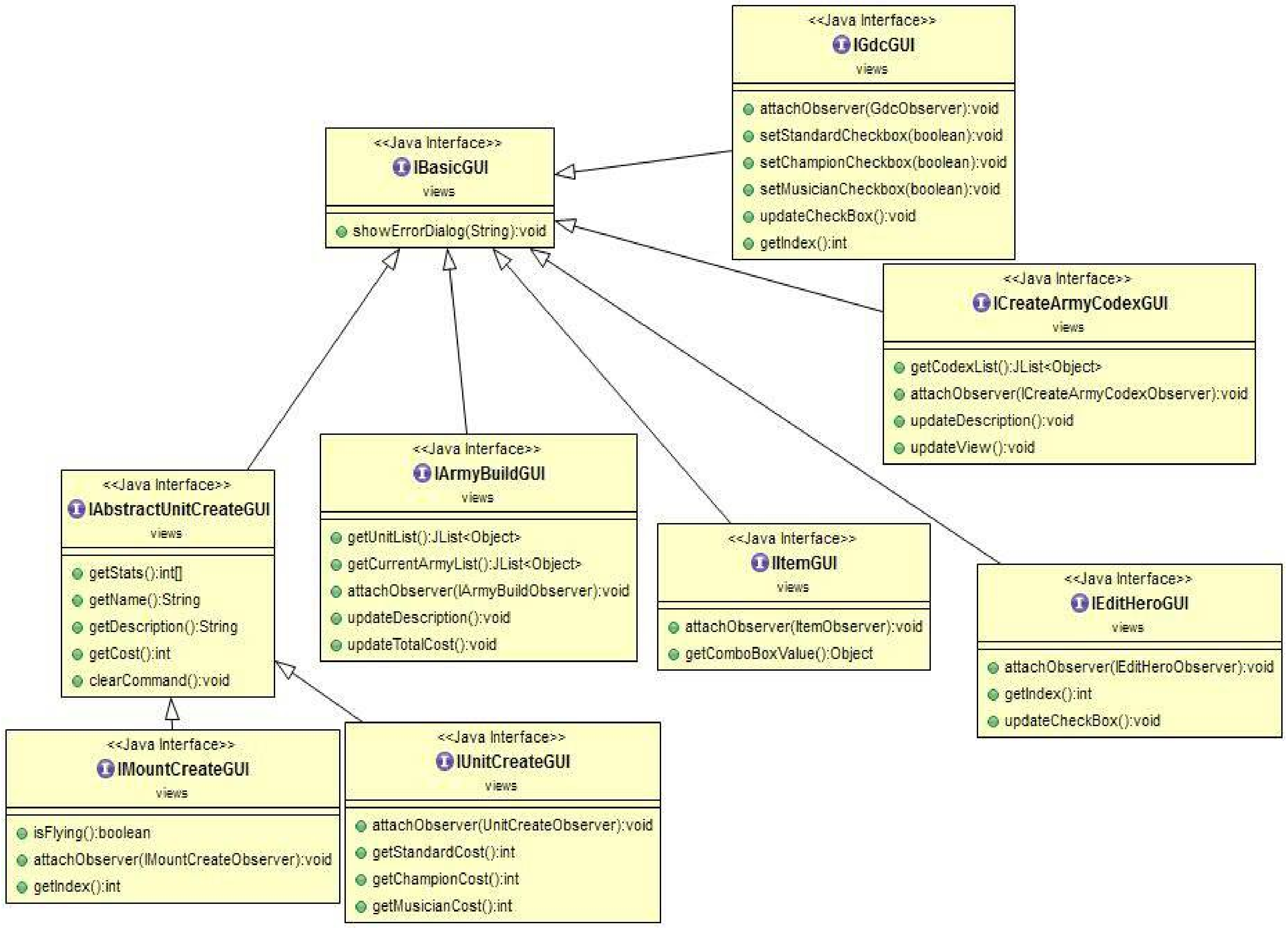


<<Java Class>>
EditHeroGUI
views
EditHeroGUI(ArmyBuildGUI,int)
updateCheckBox():void
attachObserver(IEditHeroObserver):void
getIndex():int
actionPerformed(ActionEvent):void
<<Java Class>>
AbstractUnitCreateGUI
views
AbstractUnitCreateGUI()
clearCommand():void
getStats():int[]
getName():String
getDescription():String
getCost():int
<<Java Class>>
ArmyBuildGUI
views
ArmyBuildGUI()
actionPerformed(ActionEvent):void
updateDescription():void
updateTotalCost():void
getUnitList():JList<Object>
getCurrentArmyList():JList<Object>
attachObserver(IArmyBuildObserver):void
-frameFather
0..1
-frameFather
0..1
-frameFather
0..1
-frameFather
0..1
-frameFather
0..1
<<Java Class>>
ItemGUI
views
ItemGUI(int,ArmyBuildGUI)
actionPerformed(ActionEvent):void
attachObserver(ItemObserver):void
showErrorDialog(String):void
getComboBoxValue():Object
<<Java Class>>
UnitCreateGUI
views
UnitCreateGUI(CreateArmyCodexGUI)
actionPerformed(ActionEvent):void
clearCommand():void
attachObserver(UnitCreateObserver):void
getStandardCost():int
getChampionCost():int
getMusicianCost():int
<<Java Class>>
MountCreateGUI
views
MountCreateGUI(ArmyBuildGUI,int)
getIndex():int
attachObserver(IMountCreateObserver):void
isFlying():boolean
actionPerformed(ActionEvent):void
<<Java Class>>
GdcGUI
views
GdcGUI(ArmyBuildGUI,int)
updateCheckBox():void
attachObserver(GdcObserver):void
setStandardCheckbox(boolean):void
setChampionCheckbox(boolean):void
setMusicianCheckbox(boolean):void
getIndex():int
actionPerformed(ActionEvent):void
-frameFather
0..1
<<Java Class>>
CreateArmyCodexGUI
views
CreateArmyCodexGUI(ArmyBuildGUI)
actionPerformed(ActionEvent):void
updateDescription():void
getCodexList():JList<Object>
updateView():void
attachObserver(ICreateArmyCodexObserver):void

Its important to remember that every single view 's class extends (directly or not in the specific case of UnitCreateGUI and MountCreateGUI) the abstract class AbstractBasicGUI. This important class is not in this UML because is simply extended by every other class (UnitCreateGUI and MountCreateGUI indirectly extends it by extending AbstractUnitCreateGUI) and its presence in the UML make the whole image really complex.

I have not included every nested interface observer for order purposes and also because this aspect is properly shown in the second UML in point 2 of this relation.

Following an UML showing view's interfaces and their interactions. (Putting both Interfaces and classes in a single UML seemed to me a bit chaotic.)



Its also important to remember that every single view 's interface extends (directly or not in the specific case of IUnitCreateGUI and IMountCreateGUI) the interface IBasicGUI.

Detailed Description of the Modified Panels.

These two Panels are used multiple times in the view so I thought about inserting them in a new package in order to avoid code duplication.

- **ImagePanel** work is to set an image in background following the specified dimension in the constructor. In order to paint the image resized its used an override of the paintComponent method of Jpanel. The Constructor require as parameters a path and a dimension.
- **StatsPanel** is a simple Panel that using Flowlayout create some labels that will work as a sort of index/legenda for the stats of all the units in the list. This panel will be placed right up the list of units. This Labels are ordered to align with the 'to string' of the units.

Detailed description of the Common part: Controllers.

ArmyBuildController: This controller operates as an observer for the GUI, implementing and executing save/load operations on disk. It also implements functions to move up and down Characters in the current army list.

CreateArmyCodexController: Class that models the controller for the view CreateArmyCodexGUI. Implements functions for save/load data on disk.

EditHeroController: Class that models the controller for the view EditHeroGUI. Implements only two functions. One gets the boolean that states if the selected Hero is BSB. The other sets the BSB status to the selected Hero.

GdcController: Class that models the controller for the view GdcGUI. Implements methods to check if the unit has a champion, a standard and a musician. It also provides methods to set the state for the command group of the unit.

ItemController: Class that implements the controller for ItemGUI. Provides only one method: confirmItem. It is used when the button confirm item is pressed, and assigned the new item with his inserted values to the last selected unit/hero from ArmyBuildGUI.

MountCreationController: Class that models the controller for the view MountCreateGUI. Only one method is provided by this controller. CreateMount method is used in order to create a new mount with the input values and assign it to the unit/hero at the position specified by index given as input (the index of the last selected unit in ArmyBuildGUI).

UnitCreationController: Class that implements the controller for UnitCreateGUI. This controller provides two methods really similar: createUnit and createHero. These two methods add an unit or an hero (depending from the state of the combobox of UnitCreateGUI) to the current codex.

Notes:

Every controller implements the proper nested interface 'Observer' of his related GUI. This way the controller effectively becomes the GUI observer. Every controller has in is constructor a IModel as parameter. Obviously the controller will perform all his operation only on this model. The controllers provide a setView method that sets the view on which it will work.

5)Testing

tests.HeroTest: this class provides some assertions to ensure that all operations which could be called on a Hero don't cause an Inconsistency problem both in the cost and in the hero's fields.

tests.UnitTest: this class provides some assertions to ensure that all operations which could be called on a Unit don't cause an Inconsistency problem both in the cost and in the unit's fields.

tests.ModelTest: this class provides assertions to ensure the correct execution of the operations on the Model. The first group of assertions tests add/remove operations. The second group is composed by try/catch blocks that test the launch of a NullFieldException if an input parameter of a method is NULL. The last block of try/catch tests the add fail to the army list. An Exception is thrown when the user tries to add an unit or hero that is not listed in the codex.

Final Notes

During the application development we encountered only minor problems. That lead us to operate minor changes and only some little code refactoring (PMD and Checkstyle plugins provided a huge help in this case). So we can easily state that we never changed our starting idea about the application and its deployment choices.

The members of the group never worked together before the development of this application. We didn't encounter communication problem of any kind. We developed our assigned parts in parallel using bitbucket and the support of tortoiseHg. Then we worked together in order to implement the common part of the project, because the concurrent modification of the same file using bitbucket is discouraged.

We encountered some little problems working with file extension and loading of resources from the runnable jar file.

We achieved our expectations about the application features and code maintainability, but in the future we will implement new features to upgrade the application (example: a GUI totally dedicated to item management, generalization for every single wargame, custom restriction for the current army list.....).