Assignment 5

Game: Temple Run

by

Group 9

Course: TI2206

Software Engineering Methods

Maarten Sijm
Robin van Heukelum
Mathias Meuleman
Mitchell Dingjan
Maikel Kerkhof

TA: Jurgen van Schagen

Teacher: Alberto Bacchelli

23-10-2015

Exercise 1 – 20 Time, Revolutions

Feature: Changing the Soundtrack

Short decription: during the game, the player must be able to buy new soundtracks in the shop and set the game soundtrack to the soundtrack (that is part of his/her property) that he/she wants to listen to.

Requirements

The following requirements have been approved by our Teaching Assistant on Oct 22 2015.

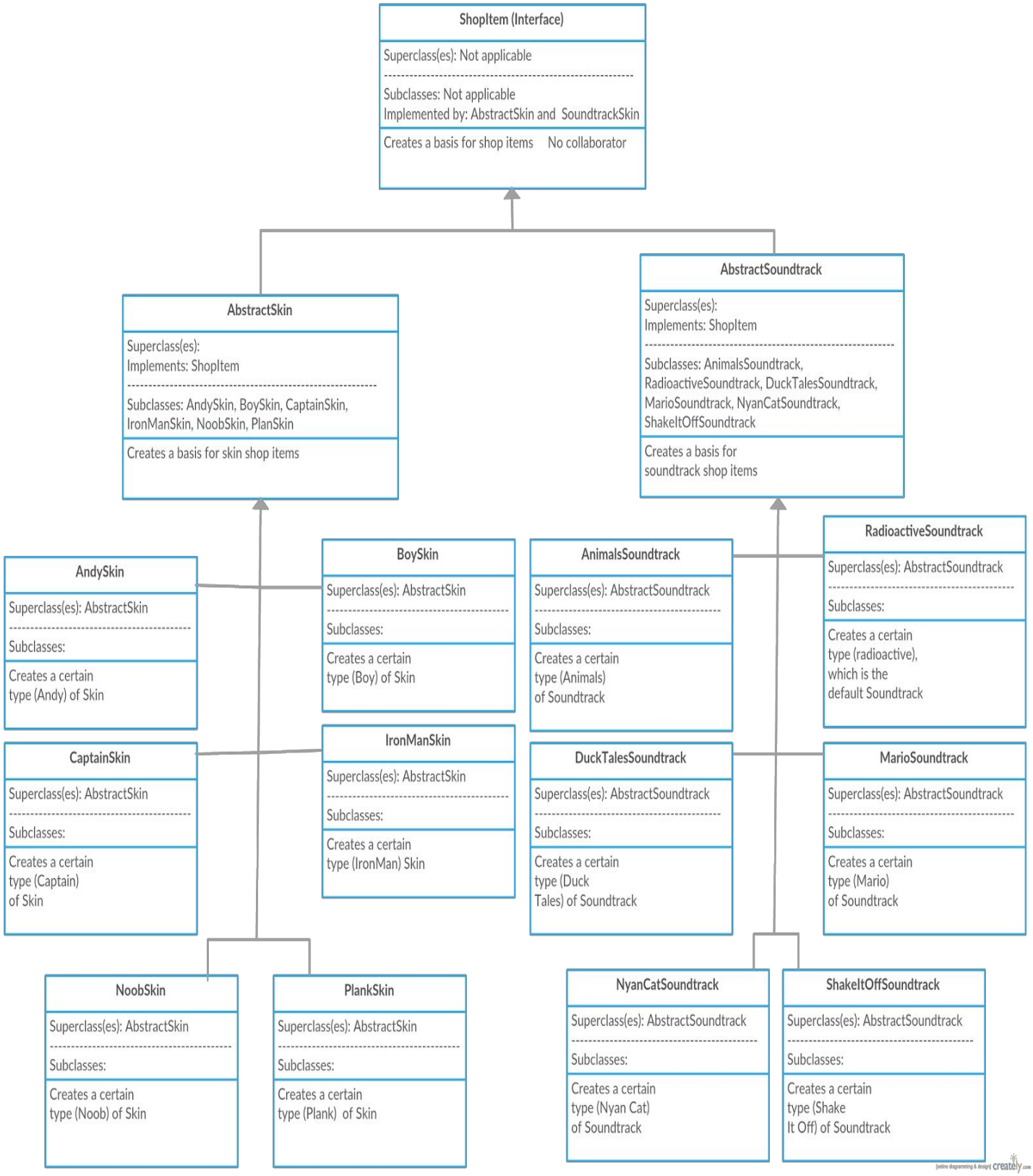
- The player can alter the soundtrack of the game.
    - The player must be able to buy a new soundtrack in the shop if the player has enough coins.
    - The user must be able to change the soundtrack by clicking on an 'Activate' button in the shop, just above the bottom of the scene.
    - The soundtrack in the Game must correspond to the soundtrack that is set to 'Current' in the shop.
    - The user should be able to see what the current soundtrack is in the shop.
    - The user should have an overview (a table or a list) in which it is clear which soundtracks are bought and which aren't and how expensive they are in the shop.
    - The user could click on a preview button, which, if is clicked on, an audioplayer is started which plays a short part of the soundtrack.

All must and should haves of the requirements described above are met. Of course the ‘must haves’ and ‘should haves’ were the minimal requirements to implement and these are indeed implemented. The last ‘could have’ requirement about the preview button is still in progress, as there was a little issue with playing only a short part of the soundtrack. Soundtracks are namely per definition infinitely long, unless stopped by an event. The could have will be completed in the next iteration.
Based on the fact that all ‘must have’ requirements and ‘should have’ requirements are met, we can confidently say that this feature has been implemented successfully.

CRC Cards

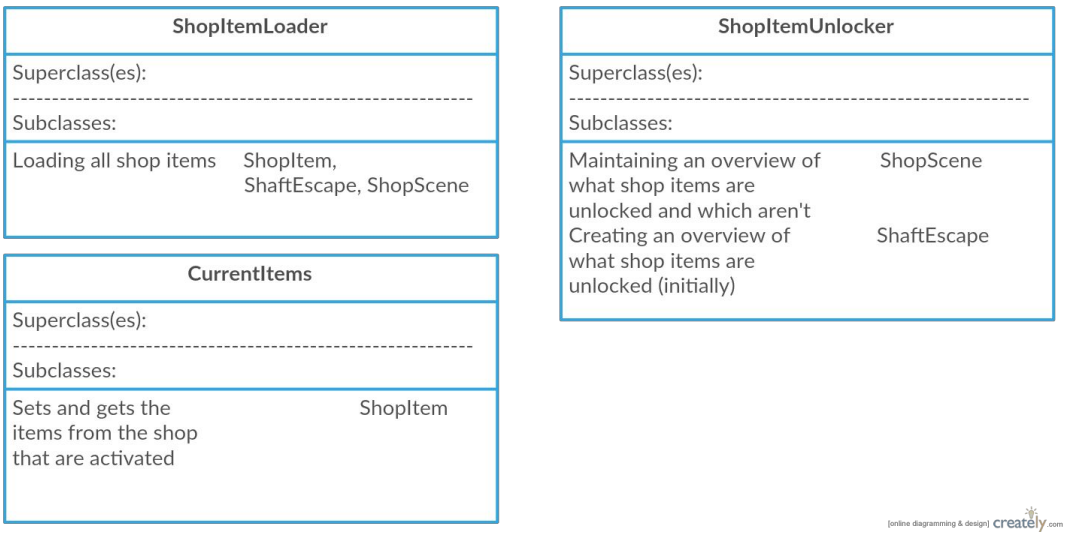
Responsibilities with respect to the shop items

(The image is as well added to git, \doc\CRC Cards, named “CRC Cards Shop Items.png”)



Responsibilities of classes that collaborate with the shop items and a utility class.

(The image is as well added to git, \doc\CRC Cards, “CRC Cards Shop Item Collaborators and Utility.png”.)



Responsibility Driven Design

This part describes how we came to the current structure for this feature, which is visualised above by the CRC cards.

First of all a short outlining
There was already an implementation for changing Skins from a previous iteration. This implementation was actually just a superclass containing a load method, an unlocker for the skins and data methods for maintaining the information of a Skin. This class acted as a superclass, with several child classes (e.g. AndySkin, BoySkin).
While creating the CRC cards, we came to the conclusion that this had to change. Skin had too many responsibilities and the whole hierarchy had to change, in order to implement the Soundtrack items.

The design with respect to the hierarchy
That’s why we came up with the interface ShopItem, which has the responsibility to give a basis for all shop items. This interface is actually at the top of the hierarchy.
Then, there is AbstractSkin and AbstractSoundtrack, of which the responsibilities are: create a basis functionality for Skins (e.g. a name, a texture) and create a basis functionality for Soundtracks (e.g. a name, a SoundtrackPlayer), respectively. For the extra responsibilities that Skin had, as described in the outlining, extra classes have been added (ShopItemLoader and ShopItemUnlocker). The described reposibility that AbstractSkin has, is the only one now, which is much better.
At the bottom of the hierarchy, there are the actual shop items. These classes have the responsibility to create a shop item. These classes contain all data in order to be a shop item and can be instantiated in order to be loaded.

The design with respect to the new classes that collaborate with the shop items
There are two main classes that are collaborating with the ShopItems in order to get the Shop working in an organised fashion: ShopItemLoader and ShopItemUnlocker.
- ShopItemLoader has the responsibility to load all shop items. This responsibility is important for the performance of the application, as shop items only have to be loaded when the ShopScene is started by the player. Besides, shop items are initialised at the start of the application, they shouldn't be initialised every time the shop is loaded or altered, etc. They just have to be initialised once. Loading both textures and audio files continuously is very inefficient.
- ShopItemUnlocker has the responsibility to create and maintain an overview of all shop items that are unlocked. Keeping this responsibility separated from the responsibility of the Shop Item classes is mainly important for testing purposes and performance. In the tests the saving and loading data about which shop items are unlocked and which aren't are tested. This data should be separated from instantiated shop item classes itself, as texture can't be loaded and AudioPlayers don't work in JUnit. Besides, altering whether a shop item is unlocked or not is performance wise not very costly by just changing a boolean in a Map.
- CurrentShopItem has the responsibility to maintain an overview of which shop items are currently activated. Besides, if the resetAll method from State is called (hard reset), the shop items will be set to their default shop items (NoobSkin and RadioactiveSoundtrack). This data is separated from the GameScene, in order to take away a responsibility from it. Besides, this responsibility should be handled in the shop package, because it involves shop items.

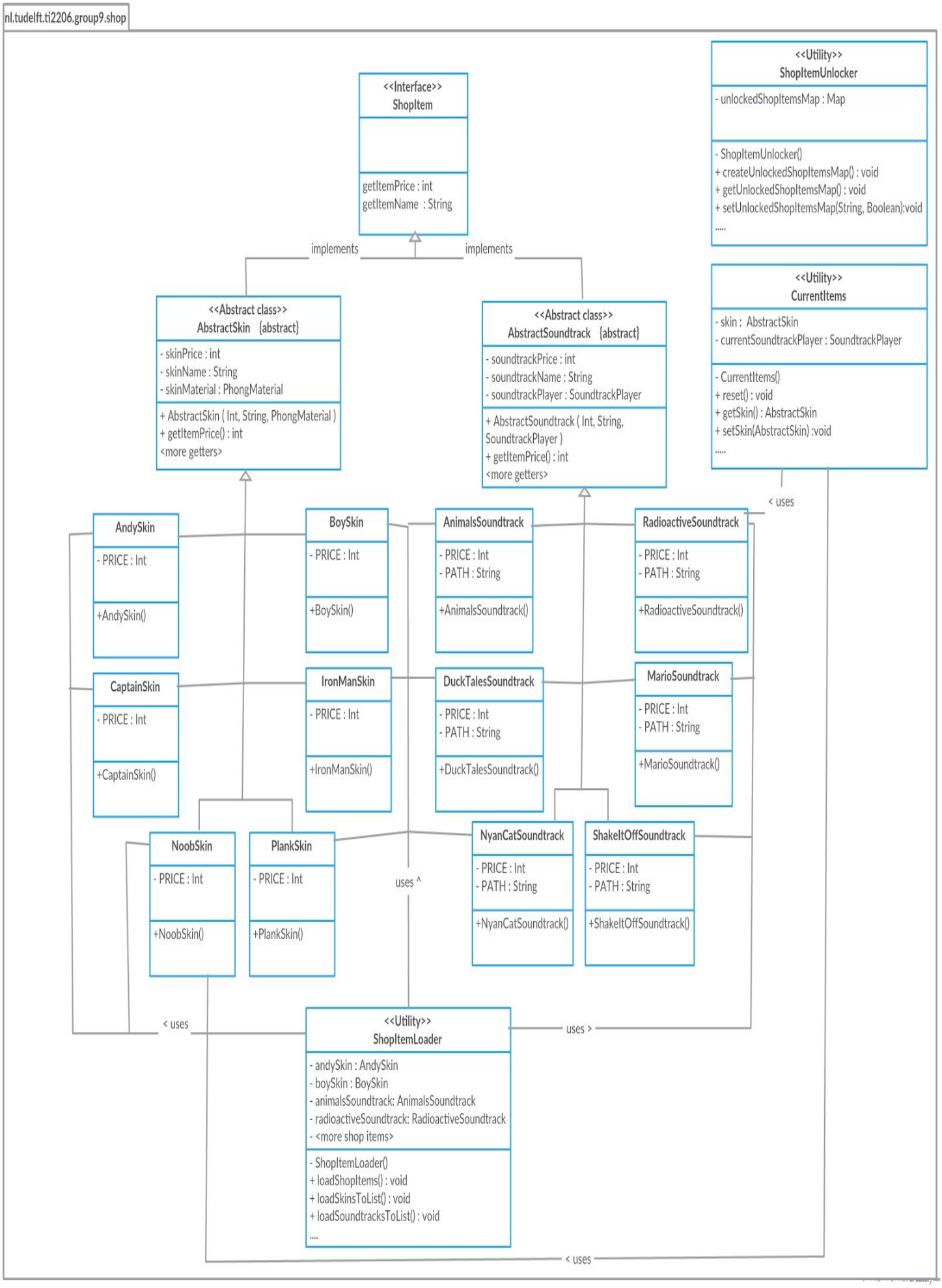
UML Class Diagram and Sequence Diagram

Class Diagram

Now that the responsibilities and structure of the new feature have been described, the structure UML of the new classes and interface shouldn't be a surprise. The UML of the shop package is given below. See the next page for the UML Class Diagram.

(The image is as well added to git, \doc\UML, "UML Shop package.png".)

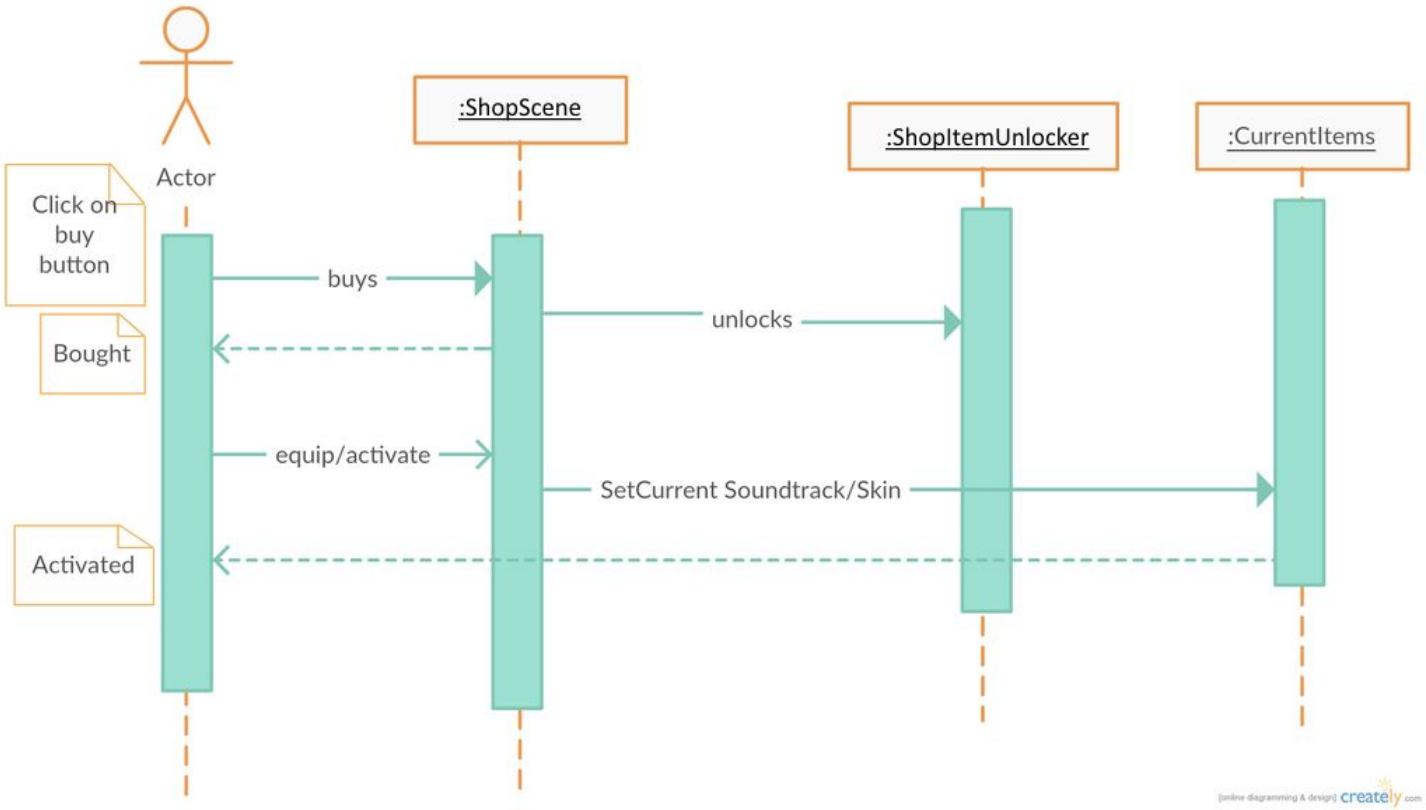
Remarks:
- Please note that the methods of ShopItemLoader are called by ShaftEscape and ShopScene. Besides, methods of ShopItemUnlocker are called as well by both ShaftEscape and the ShopScene, but also the Parser and the Writer. It will get too messy if these classes (plus package) are added as well to this overview, so these are left out.. -
- Besides, please note that in the tool that has been used, it was not possible to draw stripped lines, so the realization arrows should actually be striped (these arrows say 'implements').


nl.tudelft.ti2206.group9.shop
<<Interface>>
ShopItem
getItemPrice : int
getItemName : String
<<Utility>>
ShopItemUnlocker
- unlockedShopItemsMap : Map
- ShopItemUnlocker()
+ createUnlockedShopItemsMap() : void
+ getUnlockedShopItemsMap() : void
+ setUnlockedShopItemsMap(String, Boolean):void
.....
implements
implements
<<Abstract class>>
AbstractSkin {abstract}
- skinPrice : int
- skinName : String
- skinMaterial : PhongMaterial
+ AbstractSkin (Int, String, PhongMaterial)
+ getItemPrice() : int
<more getters>
<<Abstract class>>
AbstractSoundtrack {abstract}
- soundtrackPrice : int
- soundtrackName : String
- soundtrackPlayer : SoundtrackPlayer
+ AbstractSoundtrack (Int, String, SoundtrackPlayer)
+ getItemPrice() : int
<more getters>
<<Utility>>
CurrentItems
- skin : AbstractSkin
- currentSoundtrackPlayer : SoundtrackPlayer
- CurrentItems()
+ reset() : void
+ getSkin() : AbstractSkin
+ setSkin(AbstractSkin) :void
.....
< uses
AndySkin
- PRICE : Int
+AndySkin()
BoySkin
- PRICE : Int
+BoySkin()
AnimalsSoundtrack
- PRICE : Int
- PATH : String
+AnimalsSoundtrack()
RadioactiveSoundtrack
- PRICE : Int
- PATH : String
+RadioactiveSoundtrack()
CaptainSkin
- PRICE : Int
+CaptainSkin()
IronManSkin
- PRICE : Int
+IronManSkin()
DuckTalesSoundtrack
- PRICE : Int
- PATH : String
+DuckTalesSoundtrack()
MarioSoundtrack
- PRICE : Int
- PATH : String
+MarioSoundtrack()
NoobSkin
- PRICE : Int
+NoobSkin()
PlankSkin
- PRICE : Int
+PlankSkin()
uses ^
NyanCatSoundtrack
- PRICE : Int
- PATH : String
+NyanCatSoundtrack()
ShakeItOffSoundtrack
- PRICE : Int
- PATH : String
+ShakeItOffSoundtrack()
< uses
<<Utility>>
ShopItemLoader
- andySkin : AndySkin
- boySkin : BoySkin
- animalsSoundtrack: AnimalsSoundtrack
- radioactiveSoundtrack: RadioactiveSoundtrack
- <more shop items>
- ShopItemLoader()
+ loadShopItems() : void
+ loadSkinsToList() : void
+ loadSoundtracksToList() : void
....
uses >
< uses

Sequence Diagram

The sequence diagram below describes what methods are called when the user is performing actions (buying items and activating them) in the ShopScene of the application.

(The image is as well added to git, \doc\UML, “Sequence Diagram ShopScene.png”.)

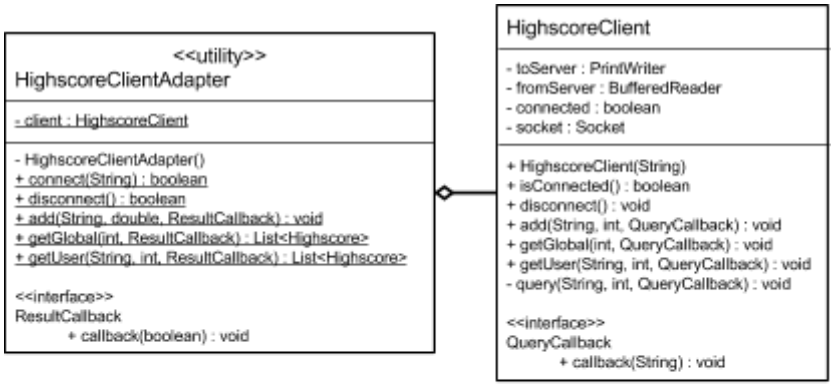


Exercise 2 – Design patterns

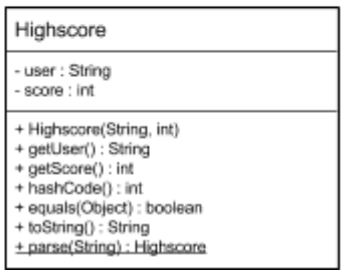
We've chosen to implement the following two patterns: the Command pattern and the Singleton pattern.

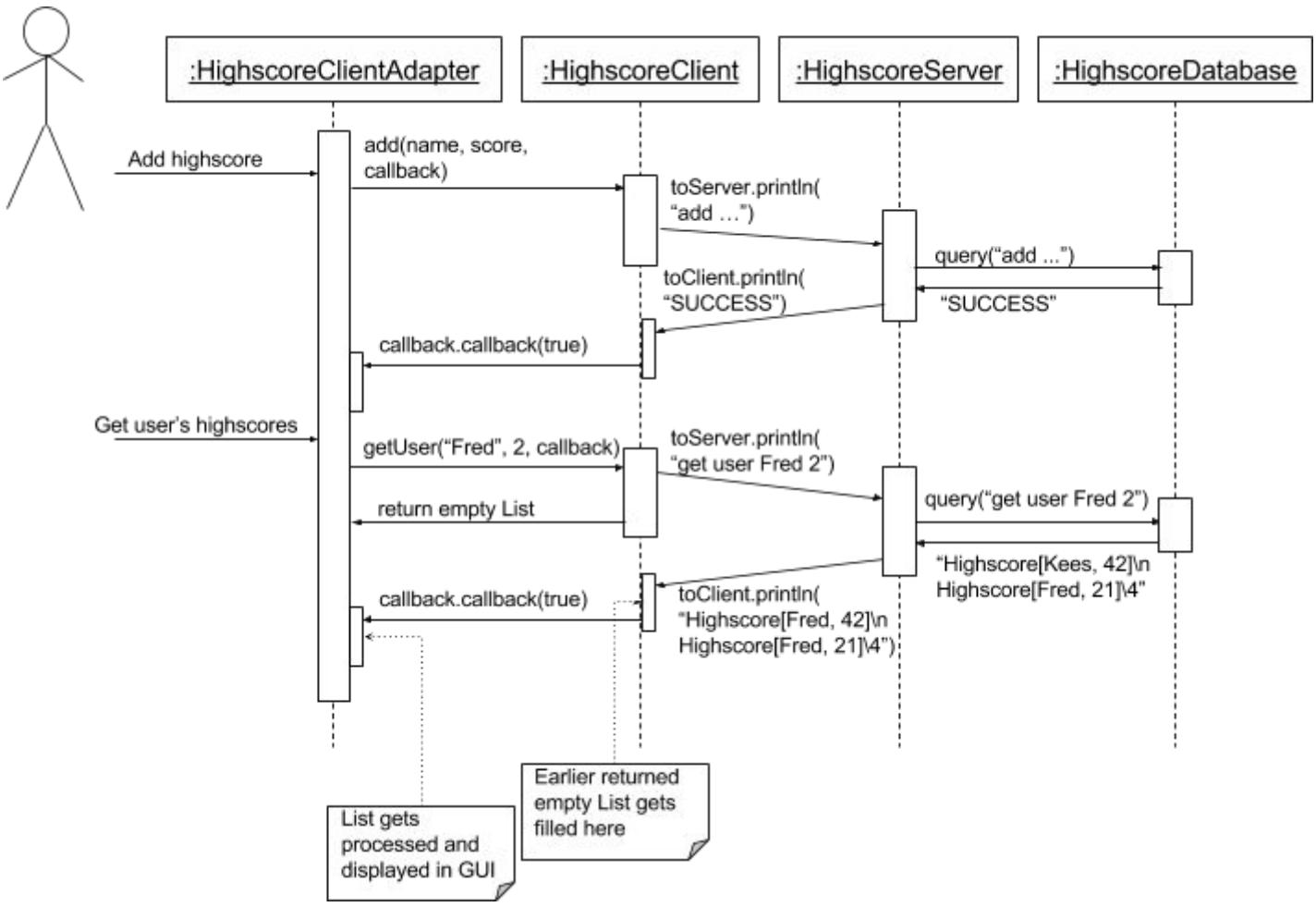
Command Pattern (highscore callback)

When a user wants to send his/her highscore to a server, the game makes sure that a connection is established with the server via the HighscoreClientAdapter. However, connecting to the server takes some time, and we do not want the game to be unresponsive while waiting for an answer from the server. This is where the Command Pattern comes in.



The HighscoreClientAdapter sends a request to the HighscoreServer, and at the same time it specifies a callback that should be called when the server has sent its response. In the callback, the caller handles the response for the server (e.g. process the list of Highscores by putting them in the GUI).



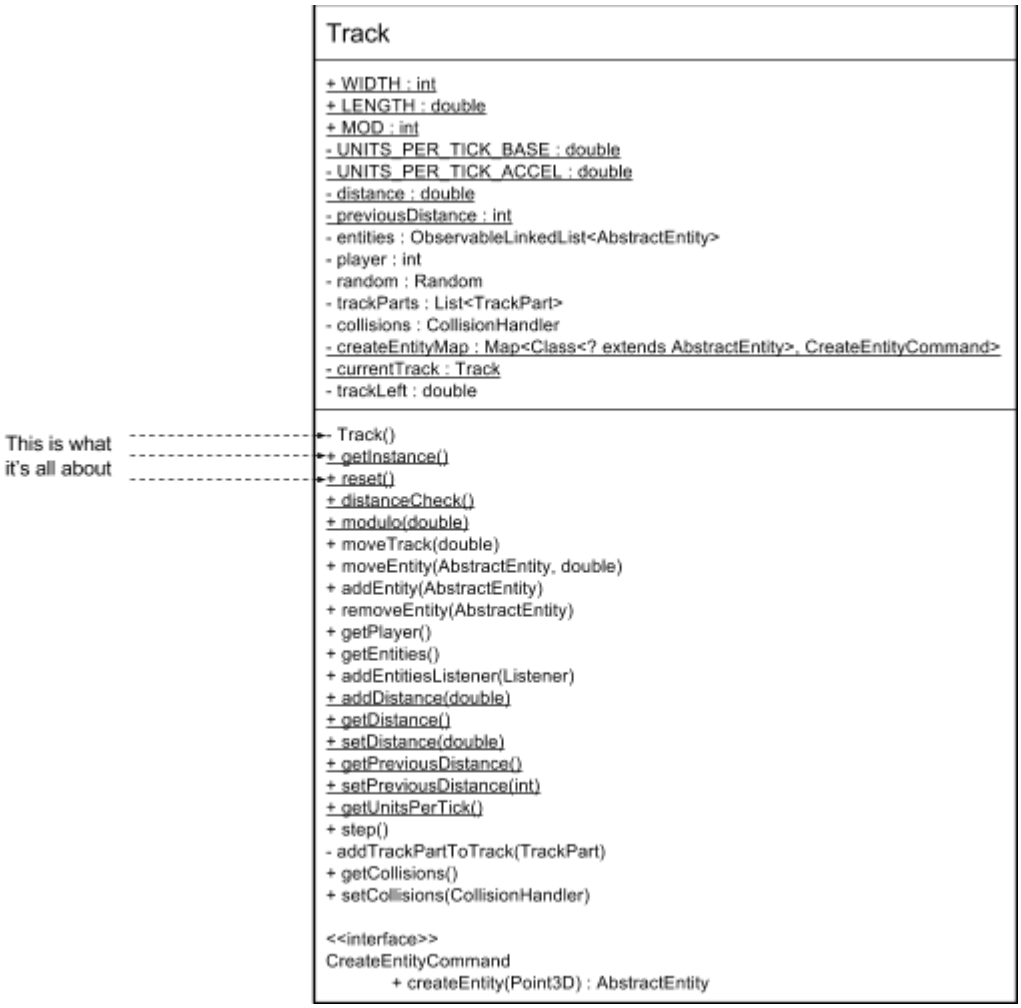


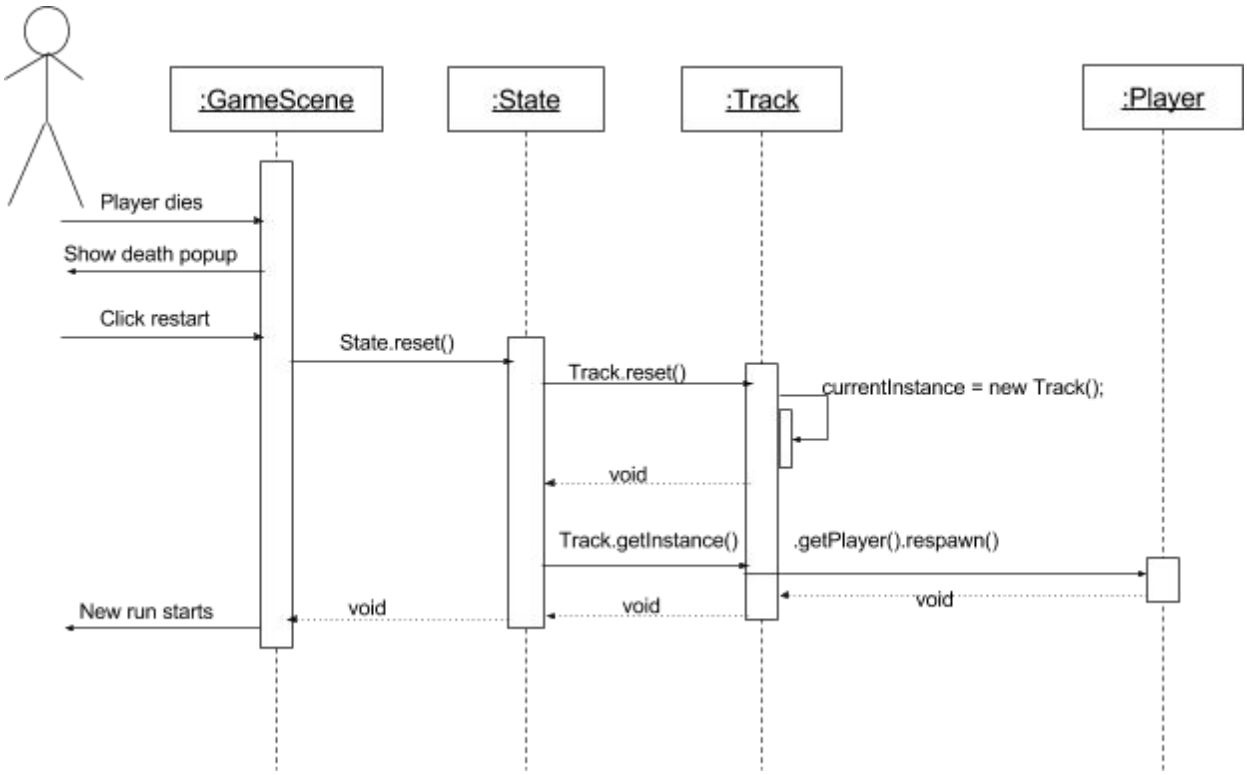
Singleton Pattern (Track)

Up until now, our State class contained a Track which could in theory be changed throughout the lifecycle of the application. This worked quite alright, but since we strive for the best code quality possible, we decided to refactor this. The idea was to refactor Track by using the Singleton pattern. If we use this pattern we can be assured that there exists one and only one instance of Track throughout the lifecycle of the application.

The class Track contains a field Track, this field will be the only instance of Track. By making the constructor of Track private, the creation of Track instances by other classes is prevented. So as long as Track itself doesn't return new Tracks, there will never be more than one Track instance in the application. The only time Track creates a new instance of Track is in the *reset()* method of Track, in which the old instance is overwritten by the new one. So we can say for sure there will never be more than one instance of Track in the application.

In order to let other classes access this single instance of Track, Track also contains a *getInstance()* method. This returns the instance of Track, at which point the class that called *getInstance()* can use this instance. This usage of the Singleton pattern is Thread safe, because the instance of Track is created in a static field, not in a callable method.




:GameScene
:State
:Track
:Player
Player dies
Show death popup
Click restart
State.reset()
Track.reset()
currentInstance = new Track();
void
Track.getInstance()
.getPlayer().respawn()
void
void
void
New run starts

Exercise 3 – Wrap up - Reflection

> *“Experience is making mistakes and learning from them.”*
>
> *Bill Ackman, American CEO* [1]

The last couple of weeks we have encountered failures and mistakes sometimes. That is why we have to reflect on the things we did, in order to learn and never make the same mistakes again. We think this quote describes one of the core features of an Agile developing style. The feedback you get after each iteration is needed in order to wipe out mistakes and learn to code in a better way.

In our project there were quite a lot of minor mistakes. Our team had to refactor quite some of code. This was, for example, due to the fact that our AudioPlayer was not working correctly. Travis did not agree with our changes, because it could not use the AudioPlayer and we had to refactor the whole code. This meant a lot of technical debt, because one branch was closed an iteration later that we had planned. It was not a big problem, but more of a frustration that was packed inside our team. However, none of this frustration was directed to a team member. We accepted the fact that the problem could not be solved and moved on. In such cases it was not our fault that we could not finish this, the tools (such as Travis) did not

Speaking of the tools we used: we really needed all of them in our process. From day 1 we have used Travis, GitHub, Slack, Waffle.io and our Google Drive. This meant we could access our data anywhere anytime. The best part of this set of tools is the integration between all the different platforms. In our case Slack was the main meeting point during the sprints outside of the meetings, with connections to all other platforms. Communication to all team members was made very easy and with lots of help of Travis and GitHub plugins in the Slack environment.

This really helped in our process, because communication is very important in such a project. Each team member is developing his feature in his own branch, not seeing what everyone else does. That is why we really valued reviewing the pull requests. We really experienced that it is valuable to see what others did and really understand what changes they made. The code quality improved as we continued reviewing. Other causes for the improvement of code quality are tools such as Checkstyle, FindBugs and PMD. They really helped us, but also frustrated us a lot. We had to make several changes in the files that keep track of the rules to check for, in order to adapt the tools to our very own project. This was rather irritating, but useful after all.

The other part of the static analysis tools is the use of Octopull. We appreciated the fact that we got the opportunity to play around with it. However the servers were down quite a lot of time and there are some issues that have to be resolved (Robin has even created an issue in the GitHub repo of the Octopull project) After all, it is useful and comes in handy while doing a review on a pull request. If this tool is developed just a bit more, we would really recommend it to our peers.

Another thing we would recommend to our peers is the use of CRC cards. It really helps figuring out what are the responsibilities of each class. It has been the cause for several refactor issues, but that is the price we were willing to pay for this code improvement. On the other hand: UML is not something we would recommend in the first place. We felt that it was useful when developing a new feature, but not to keep track of the whole project in a single diagram. In that last case it felt like a burden we had to carry and update. Therefore we would make a component UML for each new feature that we implement.

The documentation in our case mostly comes from actual Javadoc and from tests. The latter is a very good way to show a peer briefly how a system, or a unit, works in detail and what its behaviour is in a certain

---

[1] http://www.brainyquote.com/quotes/quotes/b/billackman666894.html

environment. We value the fact that our code has a high code coverage. It is useful for doing regression testing with the help of Travis. It is quite quickly able to tell if some old test has gone wrong, without you having to execute the actual test yourself. Travis helped us to keep our tests up to date and correct according to the architecture we had in mind.

The architecture itself is quite remarkable. The core has been unchanged since the start of the project. The notions of an Internal en an External Ticker have been around for a long time. Of course they have been modified, but the real architecture has not changed. Over time we have added features such as the shop and save games (which were based on initial requirements) as well as a server. The last one in particular was requested by our TA, Jurgen, he proposed to implement this rather than using a third party API. This meant quite a challenge, but we managed to implement this into our architecture as well.

Overall we are enlightened and proud to say that: we have implemented, by the time this project comes to an end, all of the features we have described in our initial requirements document! We have even taken in a step further if needed or requested. It shows that our team is capable of planning and looking ahead in the future. It is a skill we hope to develop even more as a team and this project has given it a boost already.

The most important lesson we have learned during the project is this: using the right tools can give your project a head start as well as a better progress. This choice of tools can make or break both communication and code quality. Luckily we have made the right decisions in the beginning, however it cause some errors and time to adjust the tools, but we managed to use them correctly. Therefore we can proudly present you our product: ShaftEscape!