AssetManager (singleton)
handles memory management
interface:

load[Model, Image, Sound,...] (String path) : (void *)
looks up if the resource has been already loaded, if not it returns it loads it, either way it returns reference for asset
optionally: for each load call it increments its reference counter. Then:
free[Model, Image, Sound,...] (Int reference)
decrement counter
flushUnused
could be called after loading all assets for race so we can free unused assets for ie new environment

TrackGenerator (static)
generates/load track and creates new Track instance
for detail on generation look to dedicated document
interface:

generate(int seed) : Track
generate new track based on seeded random generator
load(???): Track
loads track from description (todo: it could be some sort of string - like XML file or binary encoded?)

Track
describes track and provides some utility/statistic functions over it
interface:
//todo

Engine
not a singleton (on server there will be more than one game running at once)
should probably send messages in case of someone winning etc
should propably hold instructions for AI as singleton for engine
interface:

step(int toProccess): int
runs simulation for given time in fixed timesteps and return whats left, ie

```
while(toProcess > STEP){
doStuff()
fireEvent(stepping) OR processAI //whichever we found more suitable
toProcess -= STEP
}
return toProcess
```

it seems complicated but its really just to avoid nasty issues considering network and physics simulation, you can search it up on google or I (Aleš) can describe it in person during meeting
serialize(): ???
turn gamestate (excluding track, that would be handled separatly) into something that can be send over the network
maybe generate delta snapshots for extra reduce of lag
only changes > no creation of objects based on this (even the missiles/traps, there is fixed amount of them, so we can allocate them in advance)
loadState(???):
inverse process to upper on

setTrack(Track):

set track for current game, so we can handle boundaries, collisions etc
addVehicle():Vehicle
returns reference for vehicle that will be added in game

getVehicle(Int):Vehicle
returns vehicle reference based on its id (0-3)

Vehicle
describe position, state (breaking, turning left/right, firing?…), physical constrains (speed) and parameters (acceleration, turning rate, …)
should be also serializable

SessionsManager(singleton)
for **dedicated server**

GameSession
for **dedicated server** only, handles flow of game (lobby > game > after lobby, maybe by FS automata), holds connections, handles chats, running engine instance and recieving events from it

GameManager (singleton)
final state automata for **clients** to handle transitions between menu(s) / lobbies / game itself

Scene (or state)
abstract class for describing interaction (rendering, sound, input) between local user and game
implements:
onCreate
onChange
odExit
(self explanatory)

Menu(could be singleton, extends scene)
handles things like options and switching to appropriate state afterward

PracticeGameLobby(extends scene)
…
PracticeGame(extends scene)
simmilar to multigame

MultiGameLobbies(extends scene)
handle list of lobbies

MultiGameLobby(extends scene)
handle lobby itself

MultiGame(extends scene)
handles creating and drawing sprites on screen

GameRenderer

takes engine and renders it to scene

Sprite(abstract)

graphical representation of object, vehicle, obstacles, background, lines,...

Connection

(maybe we should split it to server / client connection?)

handles connection to server, listing lobbies, receiving updates, sending changes and so on

Controller (abstract)

describes how vehicle is controlled

implements:

connect(Vehicle, Engine):

self explanatory, engine is needed for getting track for AI

disconnect/destructor

KeyboardController

connect to keyboard events and change the state of player vehicle

AIController

AI controler, as part of connection it starts to listening to engine events, so it can change they reaction accordingly