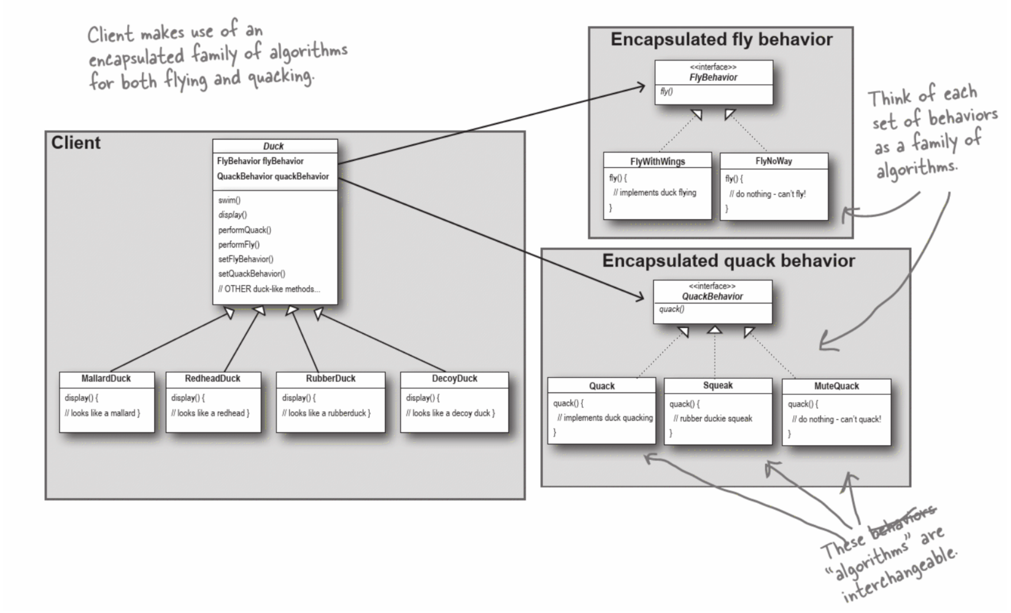
UML Basics and Designs

Part 1 Guild to UML

A Unified Modeling Language (UML) model is a simplified representation of a complex system that you can create to help understand and design the system, its components, and their relationships (reference: IBM documentation of Relational Software Modeler). It is a standard notation for modeling real-world objects, which is the first step of object-oriented design (OOD).

Classes

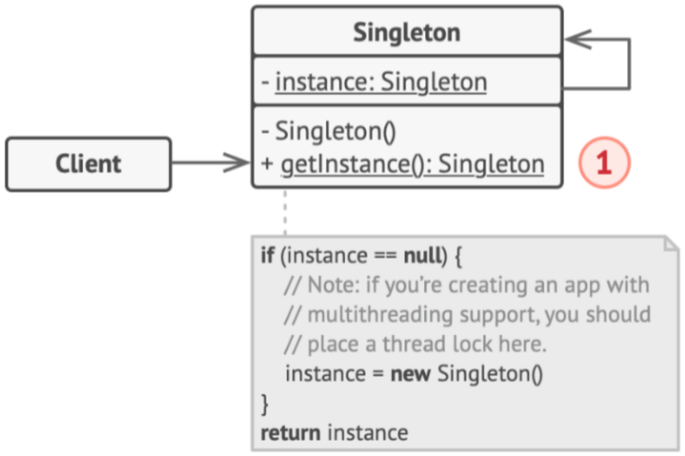
As mentioned in IBM's documentation for class diagram, in UML, "a class represents an object or a set of objects that share a common structure and behavior". The classes are represented as boxes with three parts inside: the name of the class, the attributes of the class, and the operations of the class. The first letter of the name of the class should be capitalized. The first letter of the name of the attributes and operations should be lowercase. Below is an example of a simple class diagram which shows a Duck class and its relations with FlyBehavior and QuackBehavior.


Client makes use of an encapsulated family of algorithms for both flying and quacking.
Client
Duck
FlyBehavior flyBehavior
QuackBehavior quackBehavior
swim()
display()
performQuack()
performFly()
setFlyBehavior()
setQuackBehavior()
// OTHER duck-like methods...
MallardDuck
display() { // looks like a mallard }
RedheadDuck
display() { // looks like a redhead }
RubberDuck
display() { // looks like a rubberduck }
DecoyDuck
display() { // looks like a decoy duck }
Encapsulated fly behavior
<<interface>> FlyBehavior
fly()
FlyWithWings
fly() { // implements duck flying }
FlyNoWay
fly() { // do nothing - can't fly! }
Think of each set of behaviors as a family of algorithms.
Encapsulated quack behavior
<<interface>> QuackBehavior
quack()
Quack
quack() { // implements duck quacking }
Squeak
quack() { // rubber duckie squeak }
MuteQuack
quack() { // do nothing - can't quack! }
These ~~behaviors~~ "algorithms" are interchangeable.

Duck Simulator Design, from Page 22 of *Head First Design Pattern*

Annotations

Annotations are the explanatory parts of UML. Annotations are used to describe any parts of the UML diagram. The annotations in the UML are also called notes or comments. We can create a note linking to an object or a class in the diagram. For example, the code snippet below is an annotation.

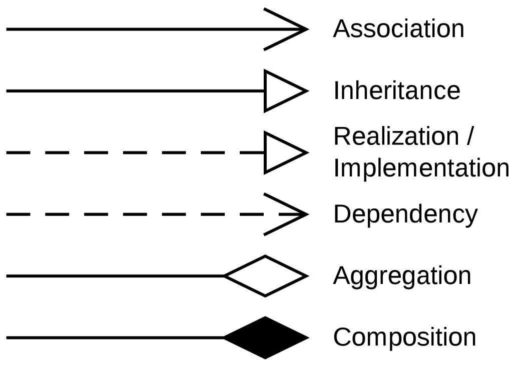


```
if (instance == null) {
   // Note: if you're creating an app with
   // multithreading support, you should
   // place a thread lock here.
   instance = new Singleton()
}
return instance
```

From Page 26 of Slide 08 Factory Method, Creational Design Pattern, 22 Fall CSci3081w, UMN

Relationships

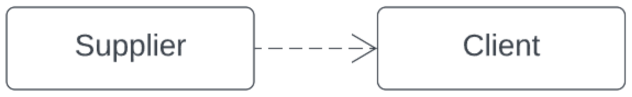
In UML, a relationship is a connection between model elements. It defines relationships and behaviors between different model elements. In this article, we are going to mainly focus on some important relationships including Dependency, Association, Realization/Implementation, Generalization/Inheritance, Aggregation, and Composition.



By Yanpas - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=48015014

Dependency

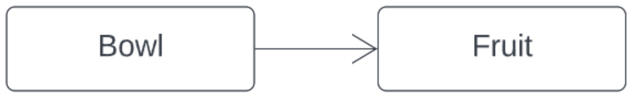
A dependency relationship is a type of association where there is a semantic connection between dependent and independent model elements. It shows a supplier-client type of relationship. Changes to one object may cause changes to the other. For example, the client is dependent on the supplier in the diagram below.



Dependency

Association

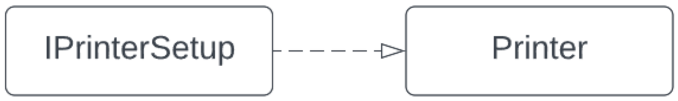
Association is the most basic relationship, meaning there is a strong relationship between classes. For example, the bowl is a container of the fruit, so we can connect Bowl class and Fruit class with an association arrow pointing from Bowl to Fruit.



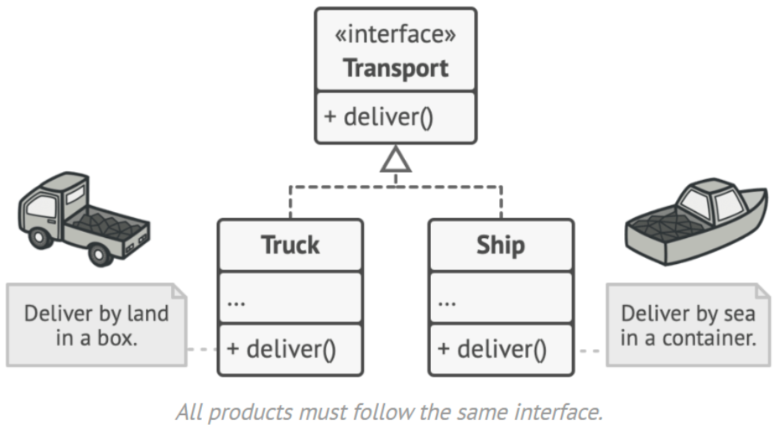
Association

Realization/Implementation

We use realization or implementation to indicate a relationship where one class implements the functions defined in another class. For example, Printer is a realization or implementation of the IPrinterSetup class. Another example, Truck and Ship are both implemented from Transport.



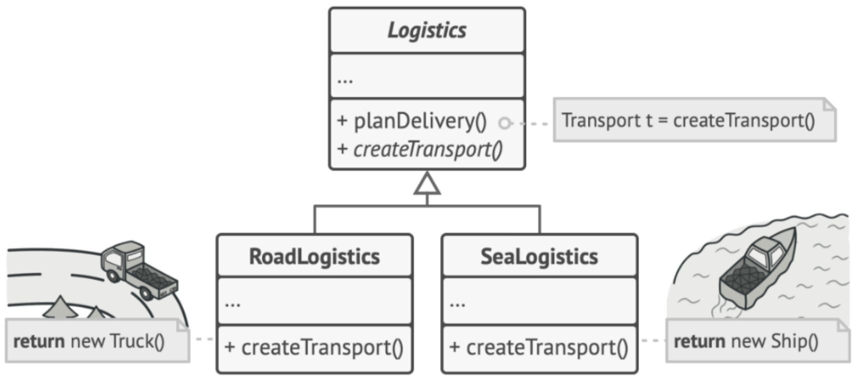
Realization/Implementation


«interface»
Transport
+ deliver()
Truck
...
+ deliver()
Ship
...
+ deliver()
Deliver by land in a box.
Deliver by sea in a container.
All products must follow the same interface.

From Page 13 of Slide 08 Factory Method, Creational Design Pattern, 22 Fall CSci3081w, UMN

Generalization/Inheritance

We use generalization or inheritance to show a child class is inherited from the parent class. The inheritance relationship is also known as IS-A relationship; subclass B is a superclass A. For example, RoadLogistics and SeaLogistics are inherited from Logistics.



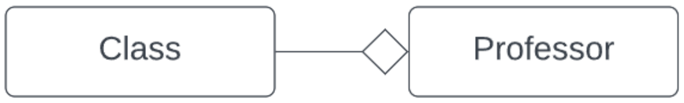
Subclasses can alter the class of objects being returned by the factory method.

From Page 12 of Slide 08 Factory Method, Creational Design Pattern, 22 Fall CSci3081w, UMN

Aggregation

Aggregation is more specific than association. Similar to the HAS-A relationship, it represents a "part of" relationship. The child class can exist independent of the parent element. They have separate lifetimes. The child class still exists when the parent class is destroyed.

For example, a professor has a class to teach, but the class and the professor are independent of each other.

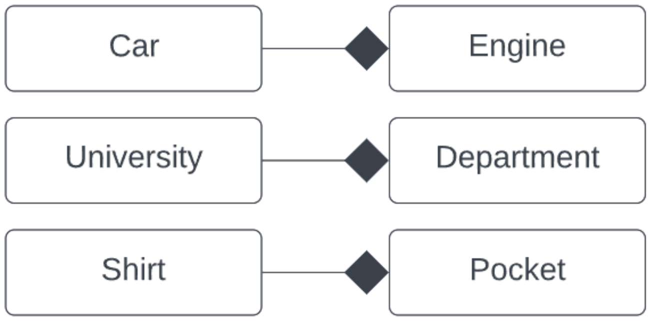


Aggregation

Composition

In composition relationships, the sub-object exists only as long as the container class exists. The two elements are not independent of each.

For example: an engine is a part of a car, and an engineer can not exist without a car; a department is a part of a university, and a department can not exist without a university; a pocket is a part of a shirt, and the pocket can not exist without a shirt.



Composition

Reference

https://www.ibm.com/docs/en/rsm/

https://www.gleek.io/blog/class-diagram-arrows.html

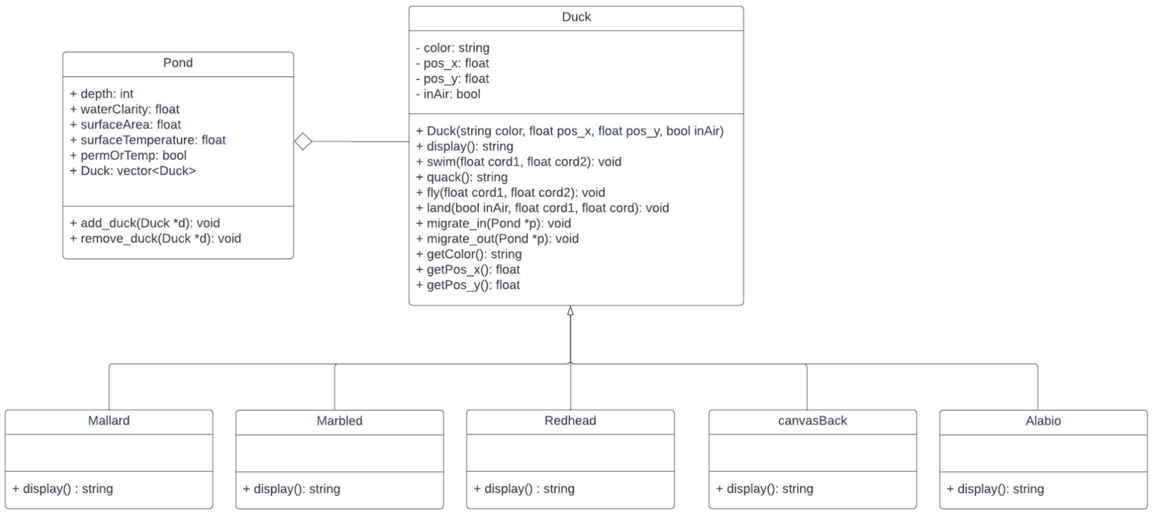
https://en.wikipedia.org/wiki/Class_diagram

Lecture notes from 22 Fall CSci3081w, UMN

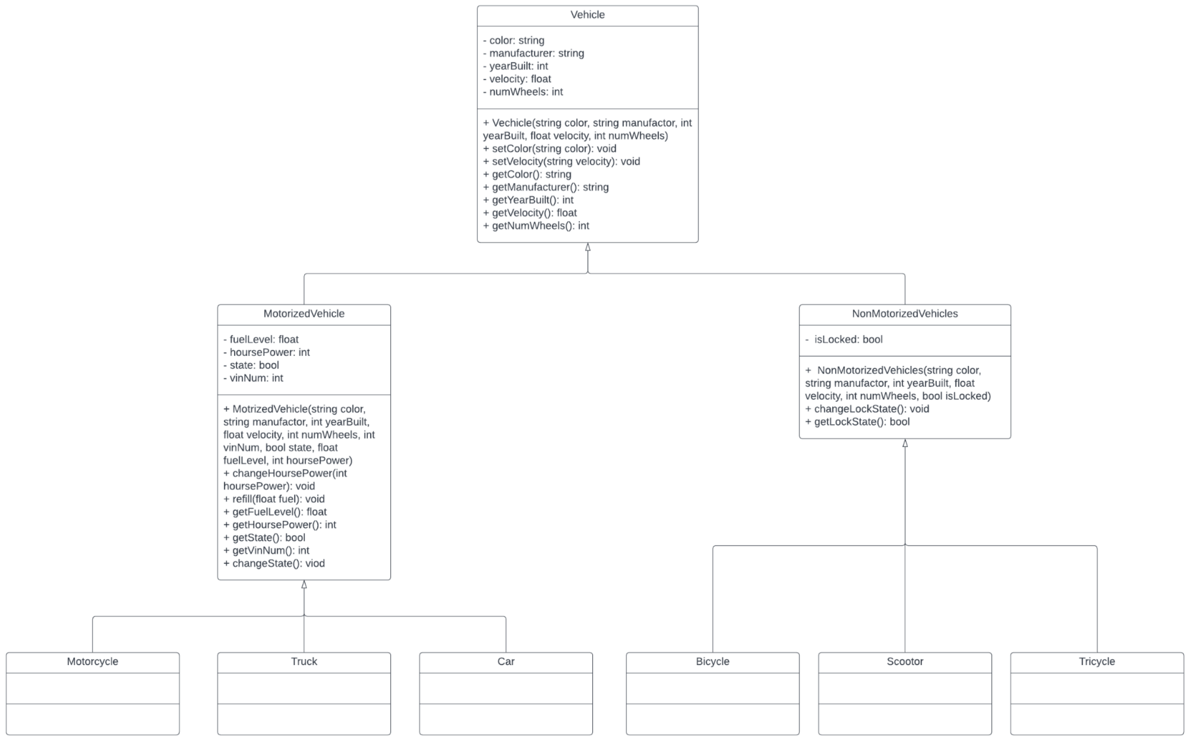
Head First Design Pattern

Part 2

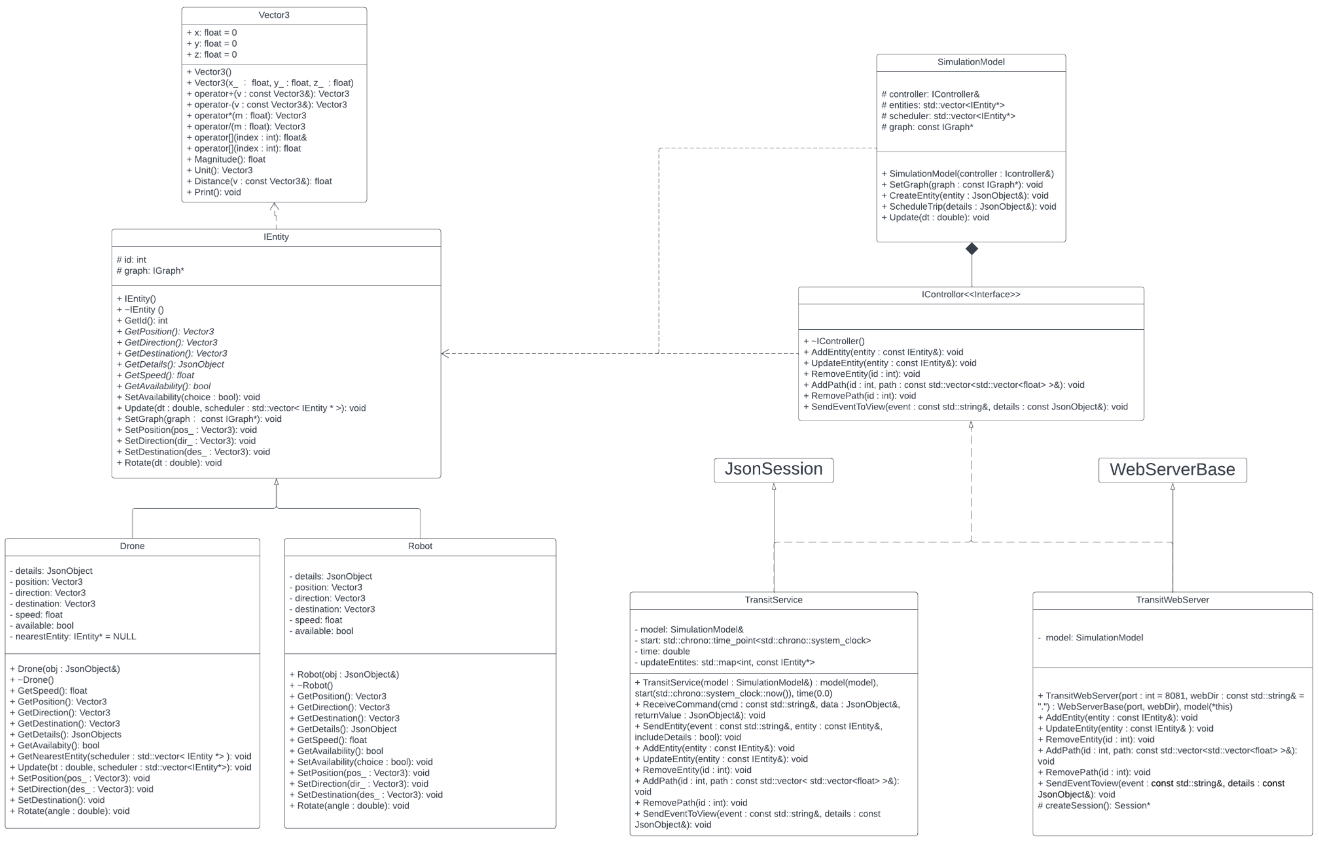
A. Ducks



B. Vehicles


Vehicle
- color: string
- manufacturer: string
- yearBuilt: int
- velocity: float
- numWheels: int
+ Vechicle(string color, string manufactor, int yearBuilt, float velocity, int numWheels)
+ setColor(string color): void
+ setVelocity(string velocity): void
+ getColor(): string
+ getManufacturer(): string
+ getYearBuilt(): int
+ getVelocity(): float
+ getNumWheels(): int
MotorizedVehicle
- fuelLevel: float
- hoursePower: int
- state: bool
- vinNum: int
+ MotrizedVehicle(string color, string manufactor, int yearBuilt, float velocity, int numWheels, int vinNum, bool state, float fuelLevel, int hoursePower)
+ changeHoursePower(int hoursePower): void
+ refill(float fuel): void
+ getFuelLevel(): float
+ getHoursePower(): int
+ getState(): bool
+ getVinNum(): int
+ changeState(): viod
NonMotorizedVehicles
- isLocked: bool
+ NonMotorizedVehicles(string color, string manufactor, int yearBuilt, float velocity, int numWheels, bool isLocked)
+ changeLockState(): void
+ getLockState(): bool
Motorcycle
Truck
Car
Bicycle
Scootor
Tricycle

C. Drone/Uber Simulation System


Vector3
+ x: float = 0
+ y: float = 0
+ z: float = 0
+ Vector3()
+ Vector3(x_ : float, y_ : float, z_ : float)
+ operator+(v : const Vector3&): Vector3
+ operator-(v : const Vector3&): Vector3
+ operator*(m : float): Vector3
+ operator/(m : float): Vector3
+ operator[](index : int): float&
+ operator[](index : int): float
+ Magnitude(): float
+ Unit(): Vector3
+ Distance(v : const Vector3&): float
+ Print(): void
IEntity
# id: int
# graph: IGraph*
+ IEntity()
+ ~IEntity ()
+ GetId(): int
+ *GetPosition(): Vector3*
+ *GetDirection(): Vector3*
+ *GetDestination(): Vector3*
+ *GetDetails(): JsonObject*
+ *GetSpeed(): float*
+ *GetAvailability(): bool*
+ SetAvailability(choice : bool): void
+ Update(dt : double, scheduler : std::vector< IEntity * >): void
+ SetGraph(graph : const IGraph*): void
+ SetPosition(pos_ : Vector3): void
+ SetDirection(dir_ : Vector3): void
+ SetDestination(des_ : Vector3): void
+ Rotate(dt : double): void
Drone
- details: JsonObject
- position: Vector3
- direction: Vector3
- destination: Vector3
- speed: float
- available: bool
- nearestEntity: IEntity* = NULL
+ Drone(obj : JsonObject&)
+ ~Drone()
+ GetSpeed(): float
+ GetPosition(): Vector3
+ GetDirection(): Vector3
+ GetDestination(): Vector3
+ GetDetails(): JsonObjects
+ GetAvailabity(): bool
+ GetNearestEntity(scheduler : std::vector< IEntity *>): void
+ Update(bt : double, scheduler : std::vector<IEntity*>): void
+ SetPosition(pos_ : Vector3): void
+ SetDirection(des_ : Vector3): void
+ SetDestination(): void
+ Rotate(angle : double): void
Robot
- details: JsonObject
- position: Vector3
- direction: Vector3
- destination: Vector3
- speed: float
- available: bool
+ Robot(obj : JsonObject&)
+ ~Robot()
+ GetPosition(): Vector3
+ GetDirection(): Vector3
+ GetDestination(): Vector3
+ GetDetails(): JsonObject
+ GetSpeed(): float
+ GetAvailability(): bool
+ SetAvailability(choice : bool): void
+ SetPosition(pos_ : Vector3): void
+ SetDirection(dir_ : Vector3): void
+ SetDestination(des_ : Vector3): void
+ Rotate(angle : double): void
SimulationModel
# controller: IController&
# entities: std::vector<IEntity*>
# scheduler: std::vector<IEntity*>
# graph: const IGraph*
+ SimulationModel(controller : Icontroller&)
+ SetGraph(graph : const IGraph*): void
+ CreateEntity(entity : JsonObject&): void
+ ScheduleTrip(details : JsonObject&): void
+ Update(dt : double): void
IControllor<<Interface>>
+ ~IController()
+ AddEntity(entity : const IEntity&): void
+ UpdateEntity(entity : const IEntity&): void
+ RemoveEntity(id : int): void
+ AddPath(id : int, path : const std::vector<std::vector<float> >&): void
+ RemovePath(id : int): void
+ SendEventToView(event : const std::string&, details : const JsonObject&): void
JsonSession
WebServerBase
TransitService
- model: SimulationModel&
- start: std::chrono::time_point<std::chrono::system_clock>
- time: double
- updateEntites: std::map<int, const IEntity*>
+ TransitService(model : SimulationModel&) : model(model), start(std::chrono::system_clock::now()), time(0.0)
+ ReceiveCommand(cmd : const std::string&, data : JsonObject&, returnValue : JsonObject&): void
+ SendEntity(event : const std::string&, entity : const IEntity&, includeDetails : bool): void
+ AddEntity(entity : const IEntity&): void
+ UpdateEntity(entity : const IEntity&): void
+ RemoveEntity(id : int): void
+ AddPath(id : int, path : const std::vector< std::vector<float> >&): void
+ RemovePath(id : int): void
+ SendEventToView(event : const std::string&, details : const JsonObject&): void
TransitWebServer
- model: SimulationModel
+ TransitWebServer(port : int = 8081, webDir : const std::string& = ".") : WebServerBase(port, webDir), model(*this)
+ AddEntity(entity : const IEntity&): void
+ UpdateEntity(entity : const IEntity&): void
+ RemoveEntity(id : int): void
+ AddPath(id : int, path: const std::vector<std::vector<float> >&): void
+ RemovePath(id : int): void
+ SendEventToview(event : const std::string&, details : const JsonObject&): void
# createSession(): Session*