Universidad ORT Uruguay
Facultad de Ingeniería

Diseño de Aplicaciones

Obligatorio 1

Lucía Dabezies - 200604
Sebastián Uriarte - 194973
Grupo M6A, 2017

Índice

Descripción general

La aplicación representa un sistema de gestión de flujo para empresas automovilísticas. La misma permite a los usuarios hacer un seguimiento de los vehículos desde que llegan a un puerto hasta que están guardados para luego ser vendidos.

Permite registrar distintos tipos de usuarios los cuales pueden realizar diferentes acciones dependiendo de su rol. Los roles existentes hasta la fecha son: administrador, operario de puerto, transportista y operario de patio.

Para poder implementar esta aplicación tuvimos que realizar una serie de suposiciones que detallaremos a continuación.

Supuestos

Alta, baja y modificación de usuarios

Decidimos que los únicos usuarios que tendrán permisos para realizar dichas funcionalidades serán aquellos que su rol sea de administrador.

No consideramos adecuado que el resto de los roles se puedan encargar de realizar dichas tareas.

Alta, baja y modificación de vehículos

No es parte del dominio de nuestro problema realizar el mantenimiento de los mismos. Se asume que ya se encuentran en el puerto al momento de comenzar su flujo de ejecución y que no pueden ser modificados ni eliminados.

Sin embargo, decidimos que los administradores si podrán realizar dichas acciones de modo que la aplicación sea más sencilla de probar. Si se requiere un vehículo con determinadas características para realizar alguna acción en particular, se podrá agregar el mismo de forma sencilla sin tener que modificar el código o agregarlo manualmente en la base de datos.

Alta, baja y modificación de lotes

Los lotes serán creados en el puerto con el fin de simplificar el transporte de los vehículos. Luego que se inicia el transporte de los mismos, ya no podrán ser modificados ni eliminados.

Por esta razón es que decidimos que quienes podrán crear, modificar y eliminar los lotes serán únicamente los usuarios que tengan rol de administrador o de operario de patio.

Alta, baja y modificación de inspecciones

Todos los roles de usuarios, con la excepción de los usuarios transportistas, tendrán permisos para registrar inspecciones. Esto se debe a que se tendrá que registrar una inspección cuando el vehículo está en el puerto y otra cuando está en el patio.

Al momento de crear una inspección se crearán tanto los daños introducidos en la misma como la ubicación en la cual se está realizando.

En el alcance de nuestro dominio las mismas no se podrán modificar ni eliminar.

Alta, baja y modificación de zona

Las zonas se encuentran en los patios y contienen subzonas en las cuales se guardarán los vehículos. Estas podrán ser creadas, modificadas y eliminadas por aquellos usuarios con rol de administrador.

Se impuso la restricción de que solo podrán ser eliminadas si no contienen subzonas en ellas.

Alta, baja y modificación de subzonas

Decidimos que las subzonas tendrán un comportamiento muy similar al de las zonas. Las mismas almacenarán vehículos para poder tener un seguimiento de donde se encuentran una vez que llegan a los patios.

Al igual que en las zonas, quienes podrán crearlas, modificarlas y eliminarlas serán los usuarios administradores.

Las subzonas sólo podrán ser eliminadas si no contienen ningún vehículo en ellas en ese momento.

Alta, baja y modificación de movimientos

Los movimientos son registros que se almacenan indicando cuando un vehículo es llevado desde una subzona hacia otra una vez que llegaron al patio.

Los usuarios que podrán realizar movimientos de vehículos serán los operarios de patio y los administradores.

Sin embargo, estas no podrán ser modificadas ni eliminadas ya que es de interés para el cliente poder llevar un control de los mismos.

Alta, baja y modificación de transportes

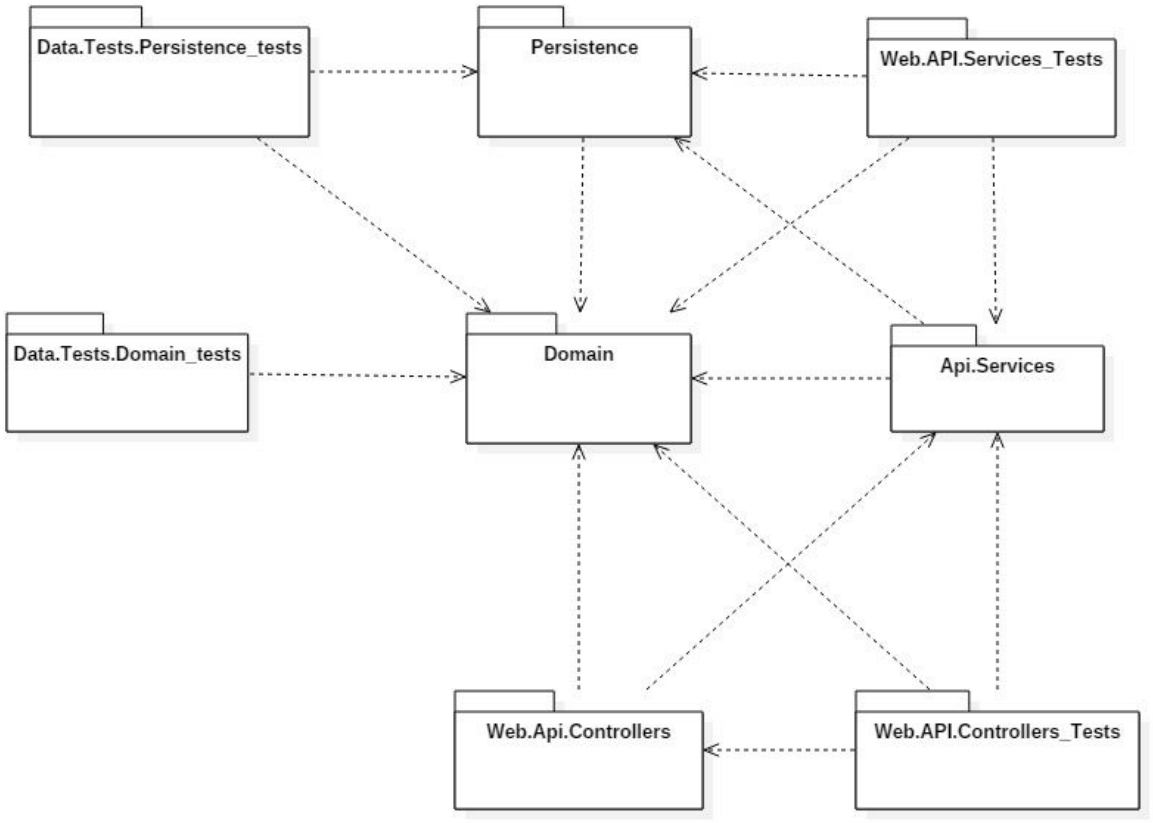
Los usuarios que tienen el rol de transportista o de administrador se encargarán de llevar los lotes desde el puerto hasta el patio. Para poder controlar dichos movimientos se registrará un transporte.

Los mismos solo podrán ser creados. No podrán modificarse ni eliminarse ya que sino se podría perder información sensible como puede ser la verdadera fecha de inicio y de finalización del mismo.

Descripción del diseño

Para mostrar la forma en la que fue diseñada la solución introduciremos una serie de diagramas UML.

Diagrama de paquetes



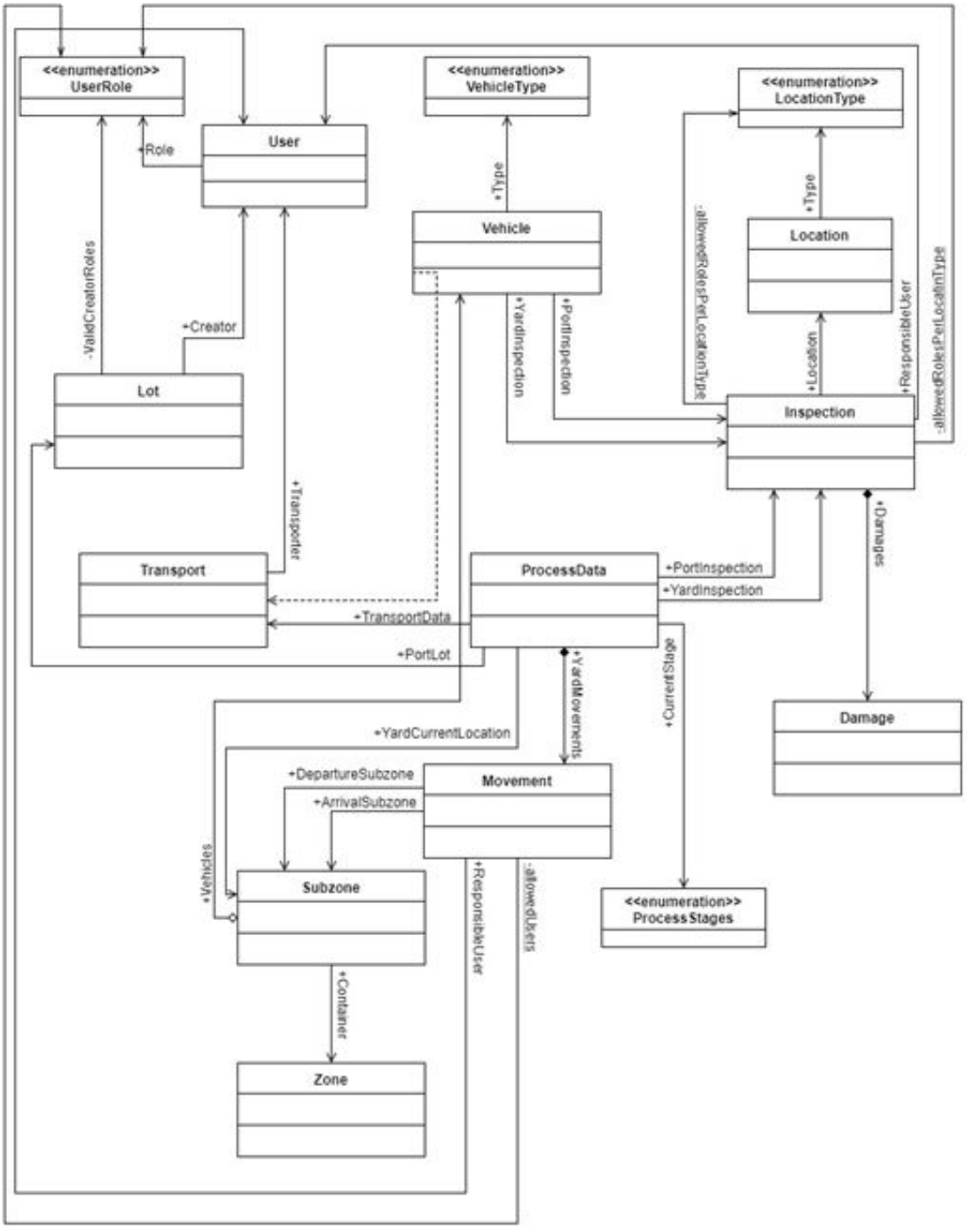
Diagramas de clases

Decidimos que los diagramas de clases que representan los paquetes que contienen las pruebas no eran relevantes al caso y optamos por no incluirlos.

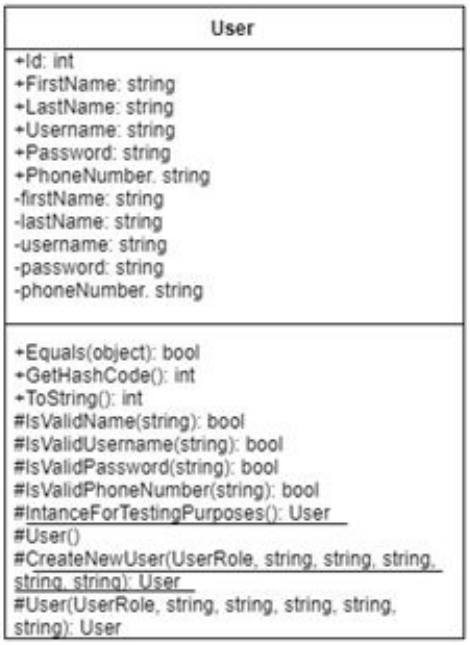
Domain

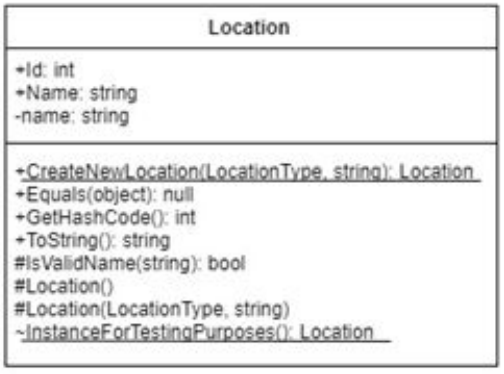
Para el diagrama de clases de este paquete decidimos mostrar las clases con sus relaciones por un lado y luego cada clase por separado con sus propiedades y métodos correspondientes. Se optó por esto ya que si lo realizamos todo en uno el mismo no entraba en una hoja y quedaba muy confuso.

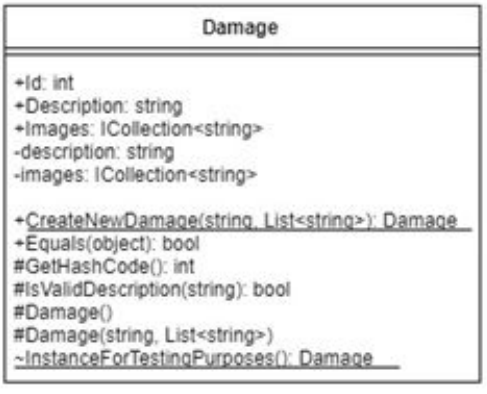
Asimismo, en el diagrama con las relaciones no incluimos las clase Utilities y las clases de excepciones ya que como todas las demás clases dependen de ellas el esquema iba a quedar con muchas flechas solapadas haciendo la lectura del mismo casi imposible.

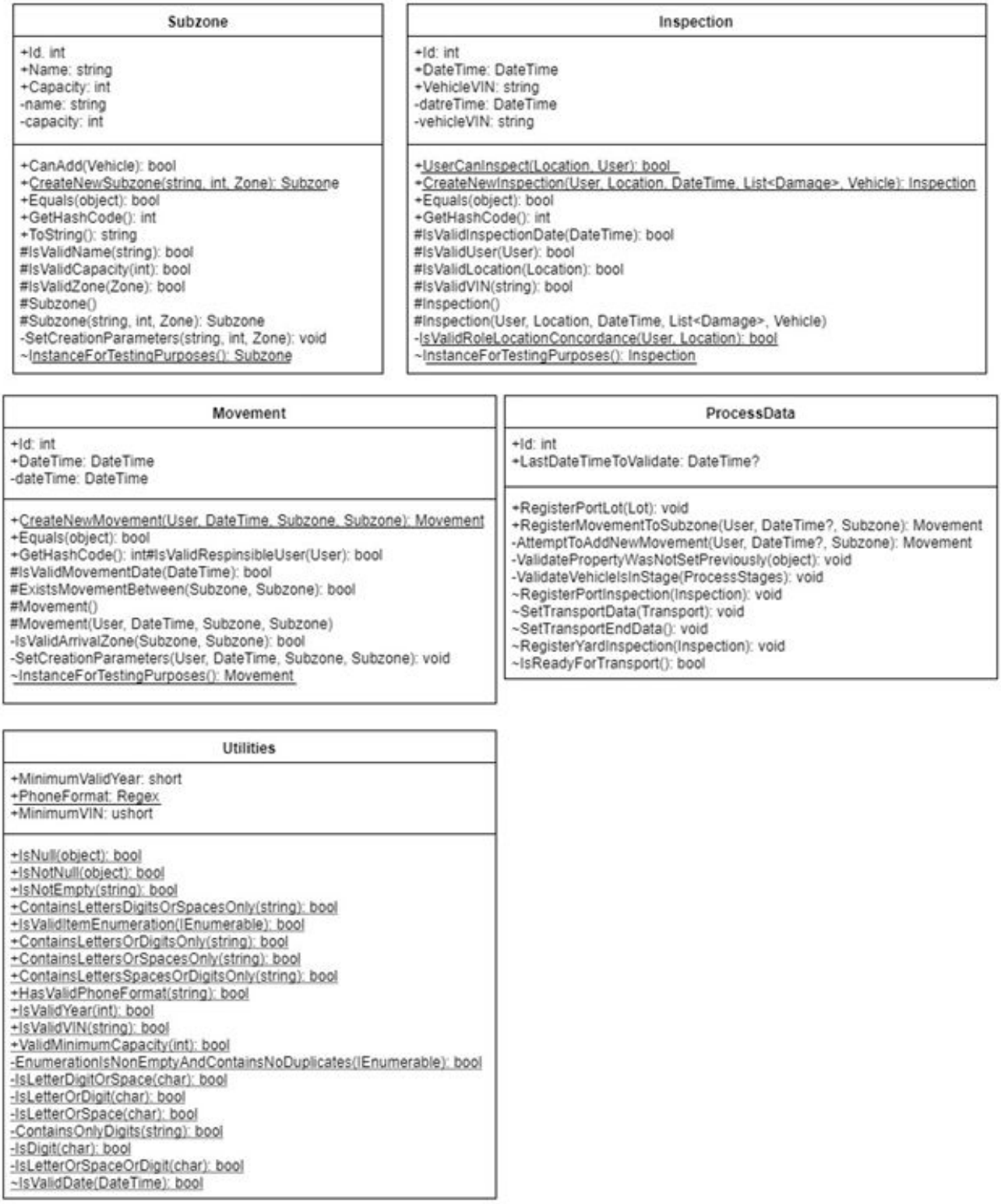


A continuación se muestra cada clase del dominio por separado con sus correspondientes métodos y propiedades:



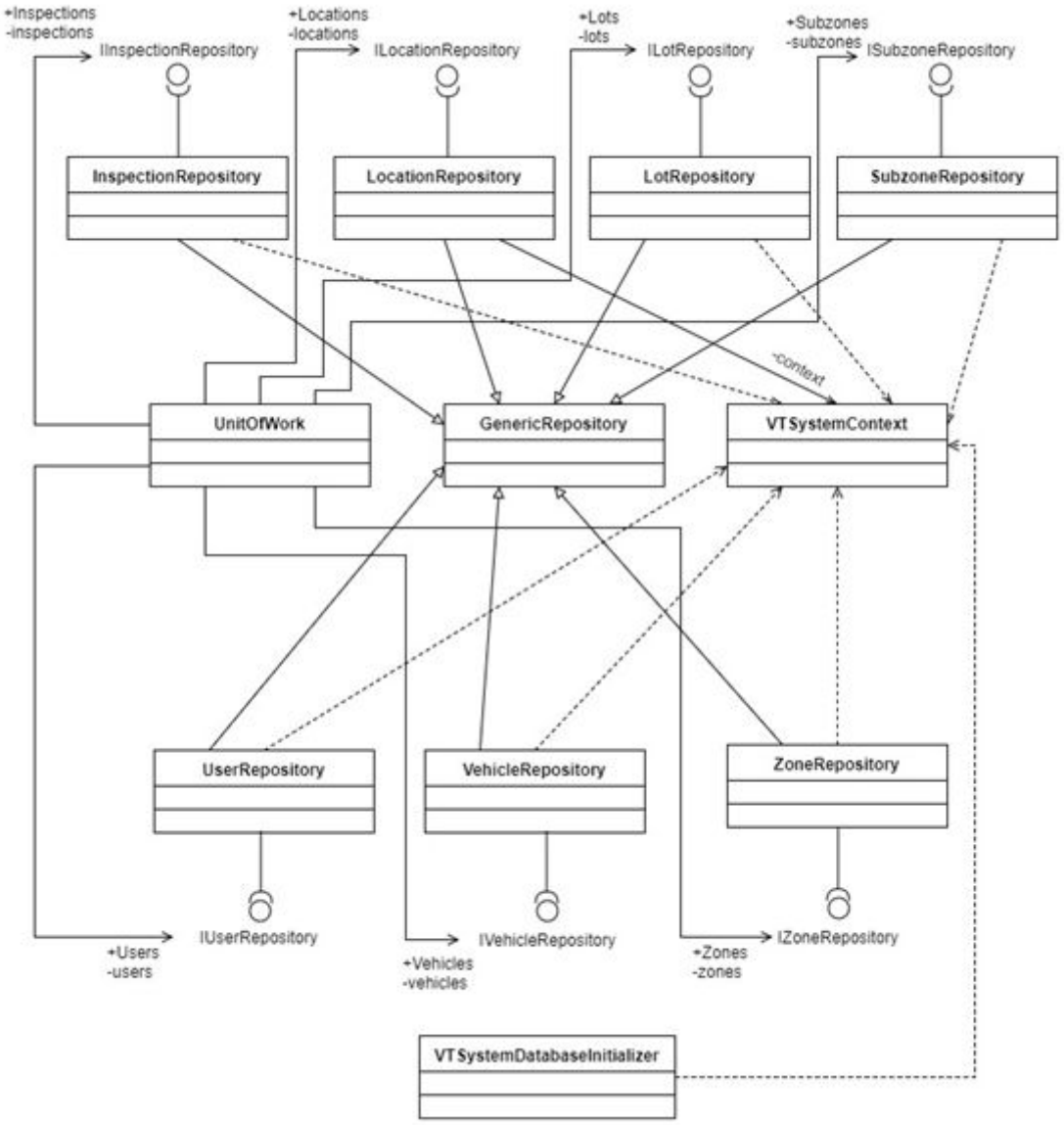


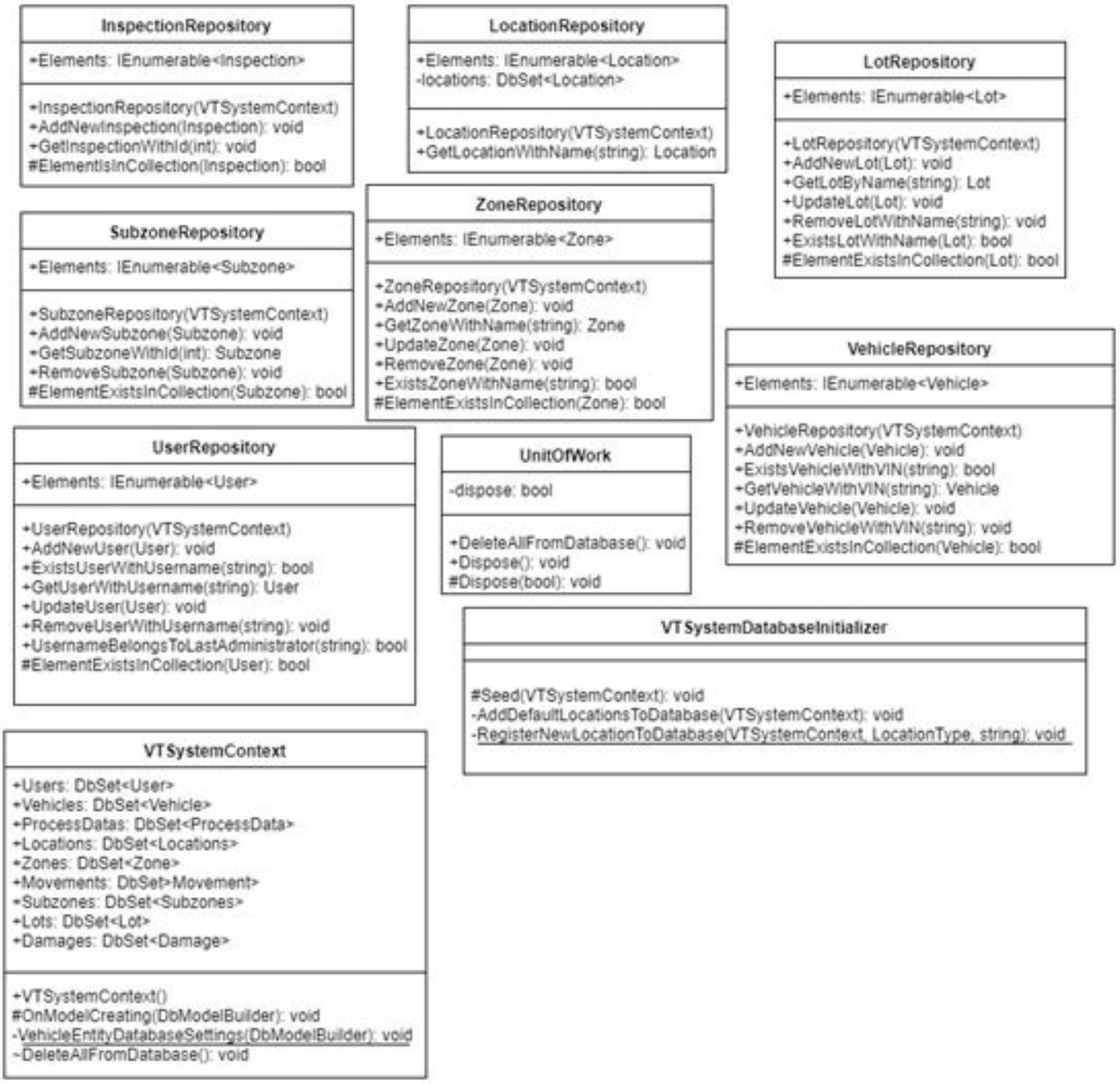



Subzone
+Id: int
+Name: string
+Capacity: int
-name: string
-capacity: int
+CanAdd(Vehicle): bool
+CreateNewSubzone(string, int, Zone): Subzone
+Equals(object): bool
+GetHashCode(): int
+ToString(): string
#IsValidName(string): bool
#IsValidCapacity(int): bool
#IsValidZone(Zone): bool
#Subzone()
#Subzone(string, int, Zone): Subzone
-SetCreationParameters(string, int, Zone): void
~InstanceForTestingPurposes(): Subzone
Inspection
+Id: int
+DateTime: DateTime
+VehicleVIN: string
-datreTime: DateTime
-vehicleVIN: string
+UserCanInspect(Location, User): bool
+CreateNewInspection(User, Location, DateTime, List<Damage>, Vehicle): Inspection
+Equals(object): bool
+GetHashCode(): int
#IsValidInspectionDate(DateTime): bool
#IsValidUser(User): bool
#IsValidLocation(Location): bool
#IsValidVIN(string): bool
#Inspection()
#Inspection(User, Location, DateTime, List<Damage>, Vehicle)
-IsValidRoleLocationConcordance(User, Location): bool
~InstanceForTestingPurposes(): Inspection
Movement
+Id: int
+DateTime: DateTime
-dateTime: DateTime
+CreateNewMovement(User, DateTime, Subzone, Subzone): Movement
+Equals(object): bool
+GetHashCode(): int#IsValidRespinsibleUser(User): bool
#IsValidMovementDate(DateTime): bool
#ExistsMovementBetween(Subzone, Subzone): bool
#Movement()
#Movement(User, DateTime, Subzone, Subzone)
-IsValidArrivalZone(Subzone, Subzone): bool
-SetCreationParameters(User, DateTime, Subzone, Subzone): void
~InstanceForTestingPurposes(): Movement
ProcessData
+Id: int
+LastDateTimeToValidate: DateTime?
+RegisterPortLot(Lot): void
+RegisterMovementToSubzone(User, DateTime?, Subzone): Movement
-AttemptToAddNewMovement(User, DateTime?, Subzone): Movement
-ValidatePropertyWasNotSetPreviously(object): void
-ValidateVehicleIsInStage(ProcessStages): void
~RegisterPortInspection(Inspection): void
~SetTransportData(Transport): void
~SetTransportEndData(): void
~RegisterYardInspection(Inspection): void
~IsReadyForTransport(): bool
Utilities
+MinimumValidYear: short
+PhoneFormat: Regex
+MinimumVIN: ushort
+IsNull(object): bool
+IsNotNull(object): bool
+IsNotEmpty(string): bool
+ContainsLettersDigitsOrSpacesOnly(string): bool
+IsValidItemEnumeration(IEnumerable): bool
+ContainsLettersOrDigitsOnly(string): bool
+ContainsLettersOrSpacesOnly(string): bool
+ContainsLettersSpacesOrDigitsOnly(string): bool
+HasValidPhoneFormat(string): bool
+IsValidYear(int): bool
+IsValidVIN(string): bool
+ValidMinimumCapacity(int): bool
-EnumerationIsNonEmptyAndContainsNoDuplicates(IEnumerable): bool
-IsLetterDigitOrSpace(char): bool
-IsLetterOrDigit(char): bool
-IsLetterOrSpace(char): bool
-ContainsOnlyDigits(string): bool
-IsDigit(char): bool
-IsLetterOrSpaceOrDigit(char): bool
~IsValidDate(DateTime): bool

Persistence

Para representar el diagrama de clases del paquete Persistence decidimos hacerlo de la misma forma que en *Domain*. Representamos las clases con sus relaciones sin sus métodos y atributos y por separado las clases con la información restante correspondiente.

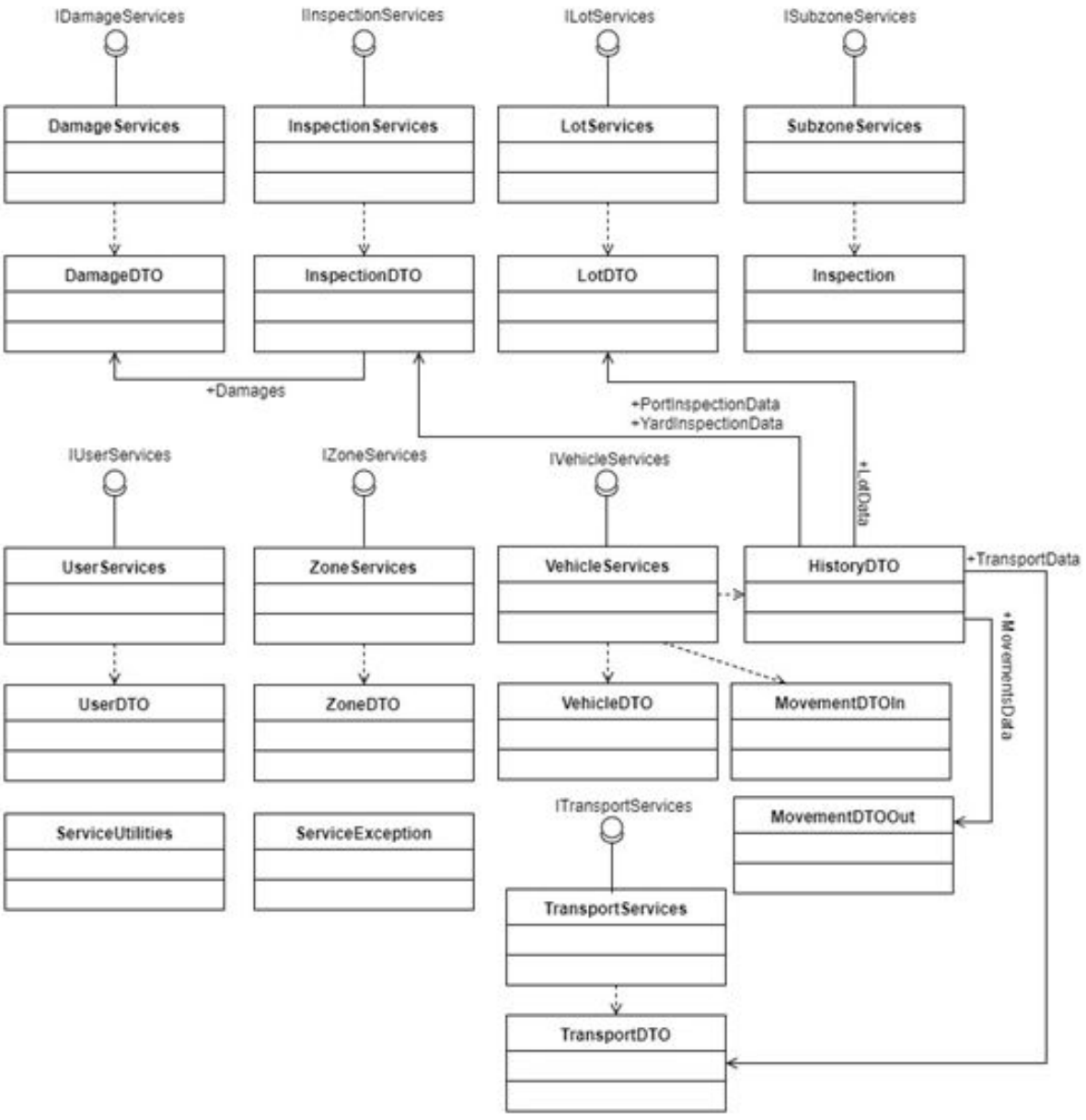


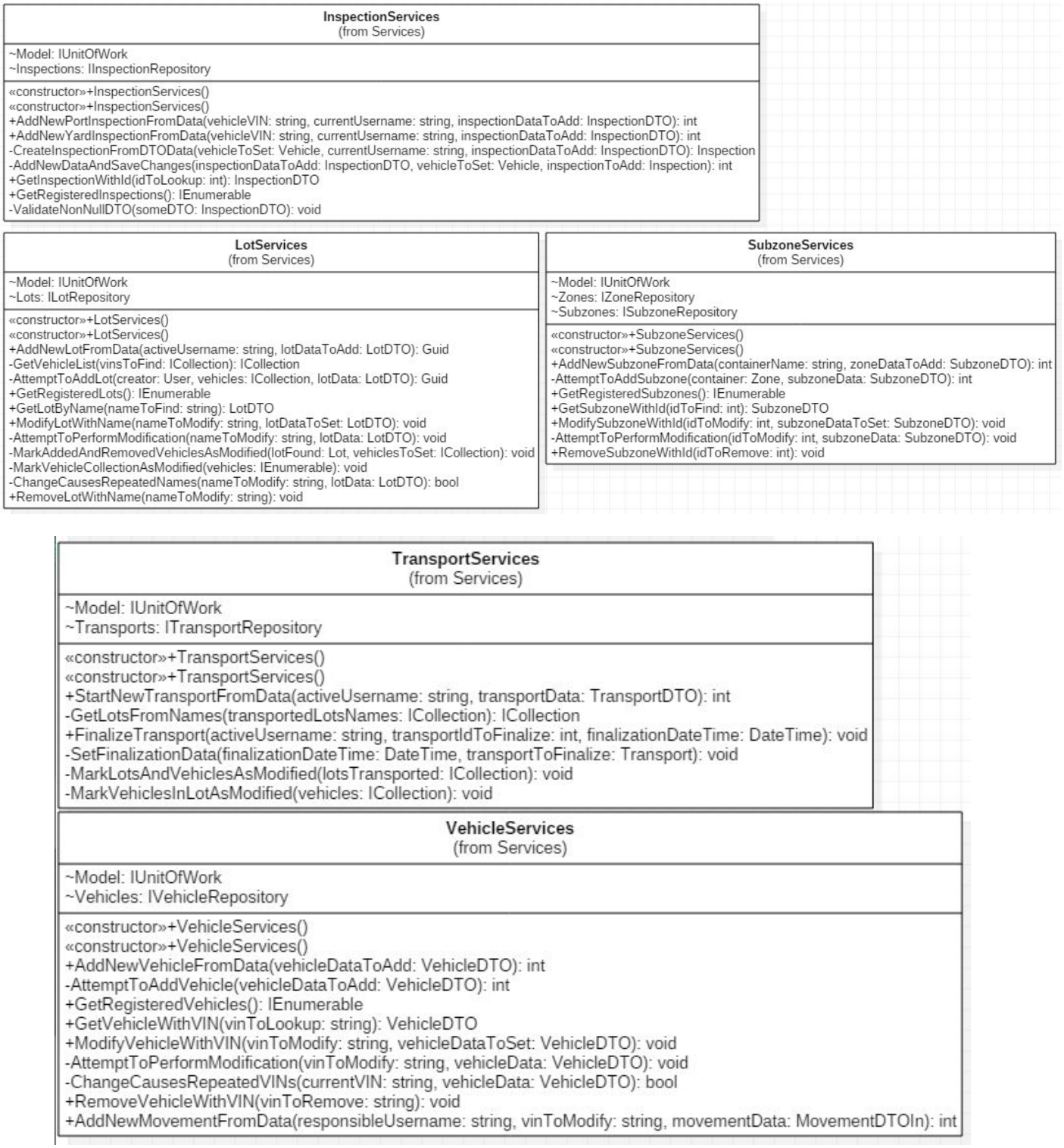

InspectionRepository
+Elements: IEnumerable<Inspection>
+InspectionRepository(VTSystemContext)
+AddNewInspection(Inspection): void
+GetInspectionWithId(int): void
#ElementIsInCollection(Inspection): bool
LocationRepository
+Elements: IEnumerable<Location>
-locations: DbSet<Location>
+LocationRepository(VTSystemContext)
+GetLocationWithName(string): Location
LotRepository
+Elements: IEnumerable<Lot>
+LotRepository(VTSystemContext)
+AddNewLot(Lot): void
+GetLotByName(string): Lot
+UpdateLot(Lot): void
+RemoveLotWithName(string): void
+ExistsLotWithName(Lot): bool
#ElementExistsInCollection(Lot): bool
SubzoneRepository
+Elements: IEnumerable<Subzone>
+SubzoneRepository(VTSystemContext)
+AddNewSubzone(Subzone): void
+GetSubzoneWithId(int): Subzone
+RemoveSubzone(Subzone): void
#ElementExistsInCollection(Subzone): bool
ZoneRepository
+Elements: IEnumerable<Zone>
+ZoneRepository(VTSystemContext)
+AddNewZone(Zone): void
+GetZoneWithName(string): Zone
+UpdateZone(Zone): void
+RemoveZone(Zone): void
+ExistsZoneWithName(string): bool
#ElementExistsInCollection(Zone): bool
VehicleRepository
+Elements: IEnumerable<Vehicle>
+VehicleRepository(VTSystemContext)
+AddNewVehicle(Vehicle): void
+ExistsVehicleWithVIN(string): bool
+GetVehicleWithVIN(string): Vehicle
+UpdateVehicle(Vehicle): void
+RemoveVehicleWithVIN(string): void
#ElementExistsInCollection(Vehicle): bool
UserRepository
+Elements: IEnumerable<User>
+UserRepository(VTSystemContext)
+AddNewUser(User): void
+ExistsUserWithUsername(string): bool
+GetUserWithUsername(string): User
+UpdateUser(User): void
+RemoveUserWithUsername(string): void
+UsernameBelongsToLastAdministrator(string): bool
#ElementExistsInCollection(User): bool
UnitOfWork
-dispose: bool
+DeleteAllFromDatabase(): void
+Dispose(): void
#Dispose(bool): void
VTSystemDatabaseInitializer
#Seed(VTSystemContext): void
-AddDefaultLocationsToDatabase(VTSystemContext): void
-RegisterNewLocationToDatabase(VTSystemContext, LocationType, string): void
VTSystemContext
+Users: DbSet<User>
+Vehicles: DbSet<Vehicle>
+ProcessDatas: DbSet<ProcessData>
+Locations: DbSet<Locations>
+Zones: DbSet<Zone>
+Movements: DbSet>Movement>
+Subzones: DbSet<Subzones>
+Lots: DbSet<Lot>
+Damages: DbSet<Damage>
+VTSystemContext()
#OnModelCreating(DbModelBuilder): void
-VehicleEntityDatabaseSettings(DbModelBuilder): void
~DeleteAllFromDatabase(): void

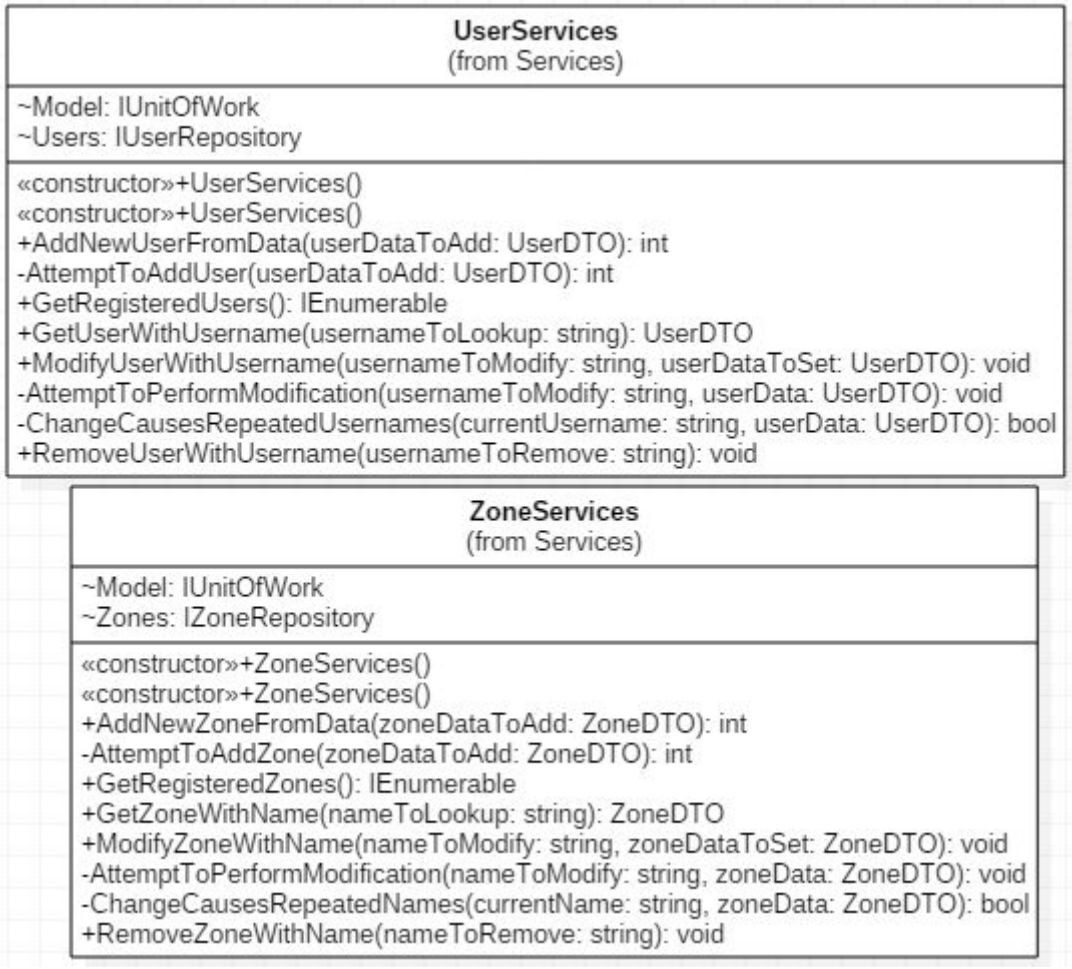
API.Services

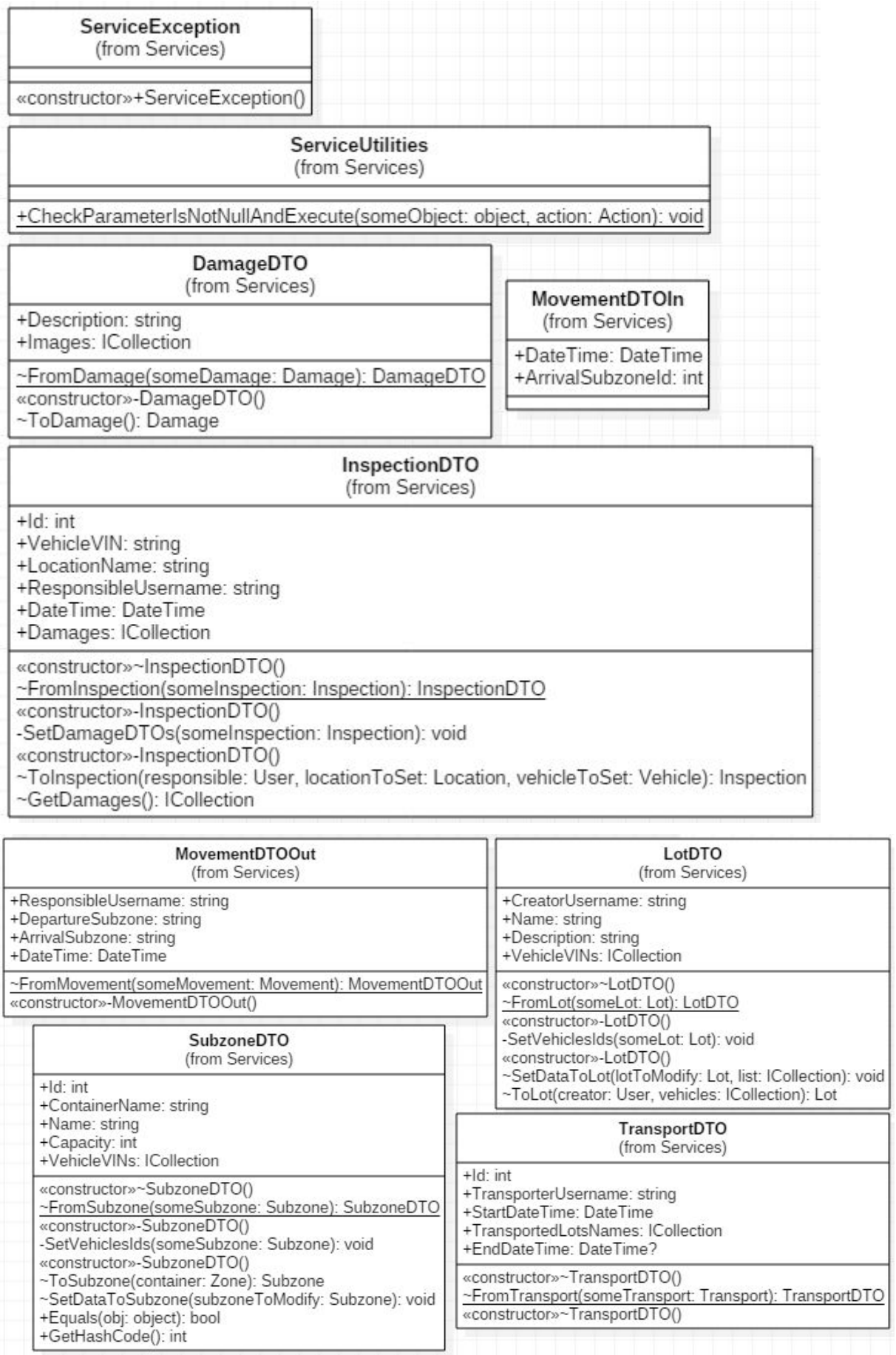
Para este paquete se optó por hacer lo mismo que para los restantes. Además, en el primer diagrama no están representadas las relaciones entre *ServiceUtilities* y *ServiceException* con el resto de las clases.

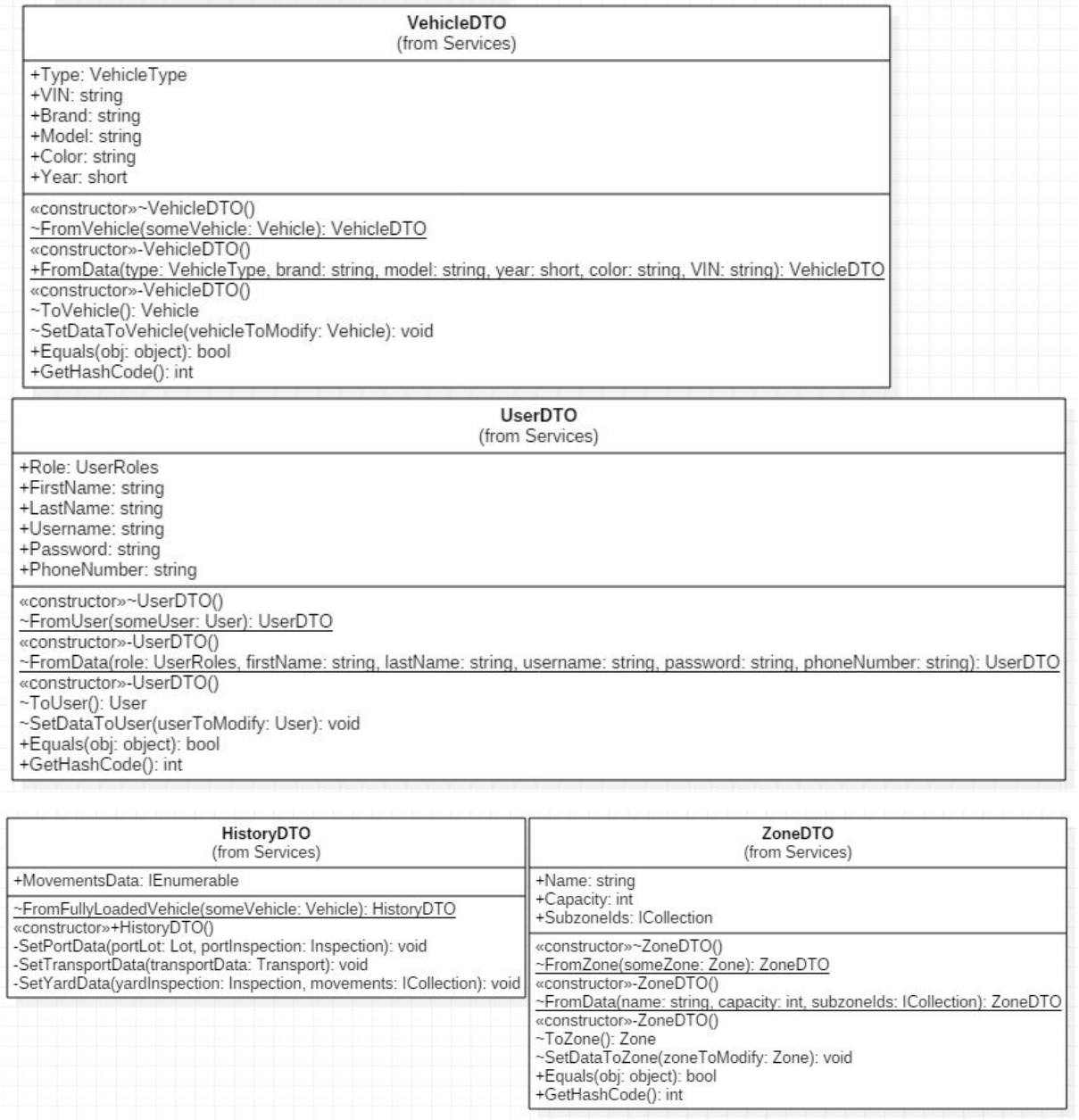
Todas las clases tienen una relación de dependencia con las dos mencionadas anteriormente.




InspectionServices
(from Services)
~Model: IUnitOfWork
~Inspections: IInspectionRepository
«constructor»+InspectionServices()
«constructor»+InspectionServices()
+AddNewPortInspectionFromData(vehicleVIN: string, currentUsername: string, inspectionDataToAdd: InspectionDTO): int
+AddNewYardInspectionFromData(vehicleVIN: string, currentUsername: string, inspectionDataToAdd: InspectionDTO): int
-CreateInspectionFromDTOData(vehicleToSet: Vehicle, currentUsername: string, inspectionDataToAdd: InspectionDTO): Inspection
-AddNewDataAndSaveChanges(inspectionDataToAdd: InspectionDTO, vehicleToSet: Vehicle, inspectionToAdd: Inspection): int
+GetInspectionWithId(idToLookup: int): InspectionDTO
+GetRegisteredInspections(): IEnumerable
-ValidateNonNullDTO(someDTO: InspectionDTO): void
LotServices
(from Services)
~Model: IUnitOfWork
~Lots: ILotRepository
«constructor»+LotServices()
«constructor»+LotServices()
+AddNewLotFromData(activeUsername: string, lotDataToAdd: LotDTO): Guid
-GetVehicleList(vinsToFind: ICollection): ICollection
-AttemptToAddLot(creator: User, vehicles: ICollection, lotData: LotDTO): Guid
+GetRegisteredLots(): IEnumerable
+GetLotByName(nameToFind: string): LotDTO
+ModifyLotWithName(nameToModify: string, lotDataToSet: LotDTO): void
-AttemptToPerformModification(nameToModify: string, lotData: LotDTO): void
-MarkAddedAndRemovedVehiclesAsModified(lotFound: Lot, vehiclesToSet: ICollection): void
-MarkVehicleCollectionAsModified(vehicles: IEnumerable): void
-ChangeCausesRepeatedNames(nameToModify: string, lotData: LotDTO): bool
+RemoveLotWithName(nameToModify: string): void
SubzoneServices
(from Services)
~Model: IUnitOfWork
~Zones: IZoneRepository
~Subzones: ISubzoneRepository
«constructor»+SubzoneServices()
«constructor»+SubzoneServices()
+AddNewSubzoneFromData(containerName: string, zoneDataToAdd: SubzoneDTO): int
-AttemptToAddSubzone(container: Zone, subzoneData: SubzoneDTO): int
+GetRegisteredSubzones(): IEnumerable
+GetSubzoneWithId(idToFind: int): SubzoneDTO
+ModifySubzoneWithId(idToModify: int, subzoneDataToSet: SubzoneDTO): void
-AttemptToPerformModification(idToModify: int, subzoneData: SubzoneDTO): void
+RemoveSubzoneWithId(idToRemove: int): void
TransportServices
(from Services)
~Model: IUnitOfWork
~Transports: ITransportRepository
«constructor»+TransportServices()
«constructor»+TransportServices()
+StartNewTransportFromData(activeUsername: string, transportData: TransportDTO): int
-GetLotsFromNames(transportedLotsNames: ICollection): ICollection
+FinalizeTransport(activeUsername: string, transportIdToFinalize: int, finalizationDateTime: DateTime): void
-SetFinalizationData(finalizationDateTime: DateTime, transportToFinalize: Transport): void
-MarkLotsAndVehiclesAsModified(lotsTransported: ICollection): void
-MarkVehiclesInLotAsModified(vehicles: ICollection): void
VehicleServices
(from Services)
~Model: IUnitOfWork
~Vehicles: IVehicleRepository
«constructor»+VehicleServices()
«constructor»+VehicleServices()
+AddNewVehicleFromData(vehicleDataToAdd: VehicleDTO): int
-AttemptToAddVehicle(vehicleDataToAdd: VehicleDTO): int
+GetRegisteredVehicles(): IEnumerable
+GetVehicleWithVIN(vinToLookup: string): VehicleDTO
+ModifyVehicleWithVIN(vinToModify: string, vehicleDataToSet: VehicleDTO): void
-AttemptToPerformModification(vinToModify: string, vehicleData: VehicleDTO): void
-ChangeCausesRepeatedVINs(currentVIN: string, vehicleData: VehicleDTO): bool
+RemoveVehicleWithVIN(vinToRemove: string): void
+AddNewMovementFromData(responsibleUsername: string, vinToModify: string, movementData: MovementDTOIn): int

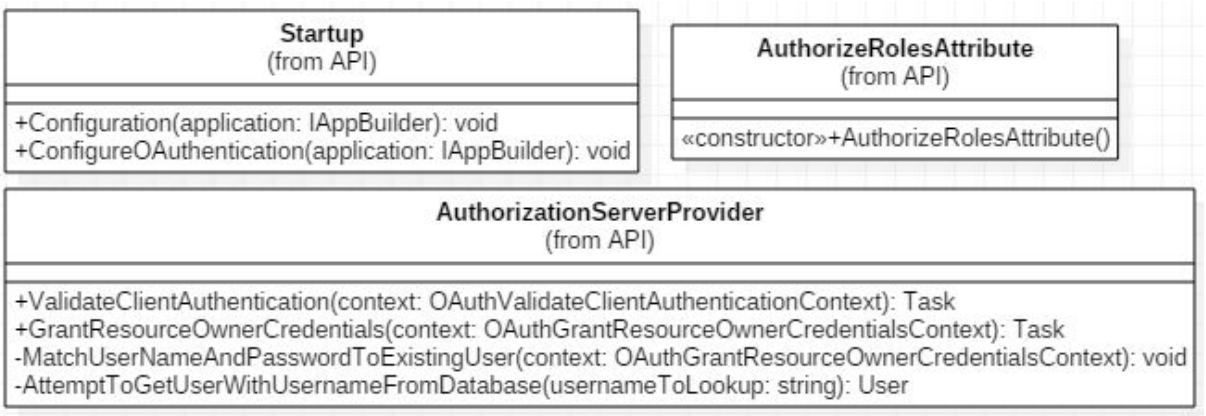

UserServices
(from Services)
~Model: IUnitOfWork
~Users: IUserRepository
«constructor»+UserServices()
«constructor»+UserServices()
+AddNewUserFromData(userDataToAdd: UserDTO): int
-AttemptToAddUser(userDataToAdd: UserDTO): int
+GetRegisteredUsers(): IEnumerable
+GetUserWithUsername(usernameToLookup: string): UserDTO
+ModifyUserWithUsername(usernameToModify: string, userDataToSet: UserDTO): void
-AttemptToPerformModification(usernameToModify: string, userData: UserDTO): void
-ChangeCausesRepeatedUsernames(currentUsername: string, userData: UserDTO): bool
+RemoveUserWithUsername(usernameToRemove: string): void
ZoneServices
(from Services)
~Model: IUnitOfWork
~Zones: IZoneRepository
«constructor»+ZoneServices()
«constructor»+ZoneServices()
+AddNewZoneFromData(zoneDataToAdd: ZoneDTO): int
-AttemptToAddZone(zoneDataToAdd: ZoneDTO): int
+GetRegisteredZones(): IEnumerable
+GetZoneWithName(nameToLookup: string): ZoneDTO
+ModifyZoneWithName(nameToModify: string, zoneDataToSet: ZoneDTO): void
-AttemptToPerformModification(nameToModify: string, zoneData: ZoneDTO): void
-ChangeCausesRepeatedNames(currentName: string, zoneData: ZoneDTO): bool
+RemoveZoneWithName(nameToRemove: string): void


ServiceException
(from Services)
«constructor»+ServiceException()
ServiceUtilities
(from Services)
+CheckParameterIsNotNullAndExecute(someObject: object, action: Action): void
DamageDTO
(from Services)
+Description: string
+Images: ICollection
~FromDamage(someDamage: Damage): DamageDTO
«constructor»-DamageDTO()
~ToDamage(): Damage
MovementDTOIn
(from Services)
+DateTime: DateTime
+ArrivalSubzoneId: int
InspectionDTO
(from Services)
+Id: int
+VehicleVIN: string
+LocationName: string
+ResponsibleUsername: string
+DateTime: DateTime
+Damages: ICollection
«constructor»~InspectionDTO()
~FromInspection(someInspection: Inspection): InspectionDTO
«constructor»-InspectionDTO()
-SetDamageDTOs(someInspection: Inspection): void
«constructor»-InspectionDTO()
~ToInspection(responsible: User, locationToSet: Location, vehicleToSet: Vehicle): Inspection
~GetDamages(): ICollection
MovementDTOOut
(from Services)
+ResponsibleUsername: string
+DepartureSubzone: string
+ArrivalSubzone: string
+DateTime: DateTime
~FromMovement(someMovement: Movement): MovementDTOOut
«constructor»-MovementDTOOut()
LotDTO
(from Services)
+CreatorUsername: string
+Name: string
+Description: string
+VehicleVINs: ICollection
«constructor»~LotDTO()
~FromLot(someLot: Lot): LotDTO
«constructor»-LotDTO()
-SetVehiclesIds(someLot: Lot): void
«constructor»-LotDTO()
~SetDataToLot(lotToModify: Lot, list: ICollection): void
~ToLot(creator: User, vehicles: ICollection): Lot
SubzoneDTO
(from Services)
+Id: int
+ContainerName: string
+Name: string
+Capacity: int
+VehicleVINs: ICollection
«constructor»~SubzoneDTO()
~FromSubzone(someSubzone: Subzone): SubzoneDTO
«constructor»-SubzoneDTO()
-SetVehiclesIds(someSubzone: Subzone): void
«constructor»-SubzoneDTO()
~ToSubzone(container: Zone): Subzone
~SetDataToSubzone(subzoneToModify: Subzone): void
+Equals(obj: object): bool
+GetHashCode(): int
TransportDTO
(from Services)
+Id: int
+TransporterUsername: string
+StartDateTime: DateTime
+TransportedLotsNames: ICollection
+EndDateTime: DateTime?
«constructor»~TransportDTO()
~FromTransport(someTransport: Transport): TransportDTO
«constructor»~TransportDTO()


VehicleDTO
(from Services)
+Type: VehicleType
+VIN: string
+Brand: string
+Model: string
+Color: string
+Year: short
«constructor»~VehicleDTO()
~FromVehicle(someVehicle: Vehicle): VehicleDTO
«constructor»-VehicleDTO()
+FromData(type: VehicleType, brand: string, model: string, year: short, color: string, VIN: string): VehicleDTO
«constructor»-VehicleDTO()
~ToVehicle(): Vehicle
~SetDataToVehicle(vehicleToModify: Vehicle): void
+Equals(obj: object): bool
+GetHashCode(): int
UserDTO
(from Services)
+Role: UserRoles
+FirstName: string
+LastName: string
+Username: string
+Password: string
+PhoneNumber: string
«constructor»~UserDTO()
~FromUser(someUser: User): UserDTO
«constructor»-UserDTO()
~FromData(role: UserRoles, firstName: string, lastName: string, username: string, password: string, phoneNumber: string): UserDTO
«constructor»-UserDTO()
~ToUser(): User
~SetDataToUser(userToModify: User): void
+Equals(obj: object): bool
+GetHashCode(): int
HistoryDTO
(from Services)
+MovementsData: IEnumerable
~FromFullyLoadedVehicle(someVehicle: Vehicle): HistoryDTO
«constructor»+HistoryDTO()
-SetPortData(portLot: Lot, portInspection: Inspection): void
-SetTransportData(transportData: Transport): void
-SetYardData(yardInspection: Inspection, movements: ICollection): void
ZoneDTO
(from Services)
+Name: string
+Capacity: int
+SubzoneIds: ICollection
«constructor»~ZoneDTO()
~FromZone(someZone: Zone): ZoneDTO
«constructor»-ZoneDTO()
~FromData(name: string, capacity: int, subzoneIds: ICollection): ZoneDTO
«constructor»-ZoneDTO()
~ToZone(): Zone
~SetDataToZone(zoneToModify: Zone): void
+Equals(obj: object): bool
+GetHashCode(): int

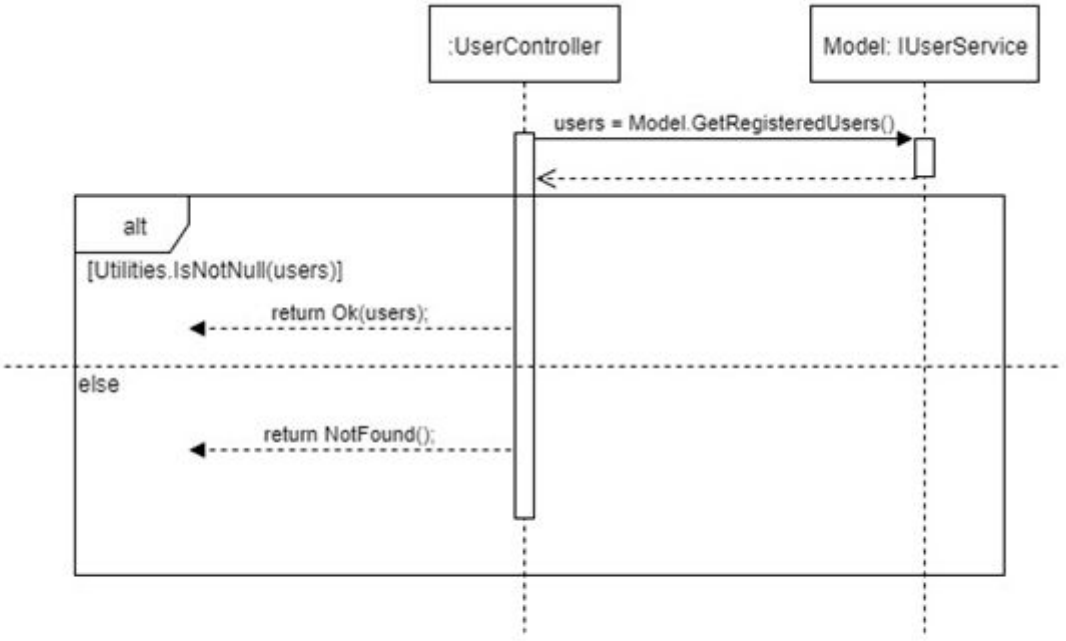
Web.API.Controllers


InspectionsController
(from Controllers)
~Model: IInspectionServices
«constructor»+InspectionsController()
«constructor»+InspectionsController()
+AddNewPortInspectionFromData(vehicleVIN: string, inspectionDataToAdd: InspectionDTO): IHttpActionResult
+AddNewYardInspectionFromData(vehicleVIN: string, inspectionDataToAdd: InspectionDTO): IHttpActionResult
+GetRegisteredInspections(): IHttpActionResult
-AttemptToGetRegisteredInspections(): IHttpActionResult
+GetInspectionWithId(idToLookup: int): IHttpActionResult
SubzonesController
(from Controllers)
~Model: ISubzoneServices
«constructor»+SubzonesController()
«constructor»+SubzonesController()
+AddNewSubzoneFromData(containerName: string, subzoneDataToAdd: SubzoneDTO): IHttpActionResult
+GetRegisteredSubzones(): IHttpActionResult
-AttemptToGetRegisteredSubzones(): IHttpActionResult
+GetSubzoneWithId(idToLookup: int): IHttpActionResult
+ModifySubzoneWithId(idToModify: int, userDataToSet: SubzoneDTO): IHttpActionResult
+RemoveSubzoneWithId(idToRemove: int): IHttpActionResult
TransportsController
(from Controllers)
~Model: ITransportServices
«constructor»+TransportsController()
«constructor»+TransportsController()
+StartNewTransportFromData(transportDataToAdd: TransportDTO): IHttpActionResult
+ModifyUserWithUsername(transportIdToFinalize: int, finalizationDateTime: DateTime): IHttpActionResult
UsersController
(from Controllers)
~Model: IUserServices
«constructor»+UsersController()
«constructor»+UsersController()
+AddNewUserFromData(userDataToAdd: UserDTO): IHttpActionResult
+GetRegisteredUsers(): IHttpActionResult
-AttemptToGetRegisteredUsers(): IHttpActionResult
+GetUserByUsername(usernameToLookup: string): IHttpActionResult
+ModifyUserWithUsername(usernameToModify: string, userDataToSet: UserDTO): IHttpActionResult
+RemoveUserWithUsername(usernameToRemove: string): IHttpActionResult
VehiclesController
(from Controllers)
~Model: IVehicleServices
«constructor»+VehiclesController()
«constructor»+VehiclesController()
+AddNewVehicleFromData(vehicleDataToAdd: VehicleDTO): IHttpActionResult
+GetRegisteredVehicles(): IHttpActionResult
-AttemptToGetRegisteredVehicles(): IHttpActionResult
+GetVehicleWithVIN(vinToLookup: string): IHttpActionResult
+ModifyVehicleWithVIN(vinToModify: string, vehicleDataToSet: VehicleDTO): IHttpActionResult
+RemoveVehicleWithVIN(vinToRemove: string): IHttpActionResult
+AddMovementToVehicleWith(vinToModify: string, movementData: MovementDTOIn): IHttpActionResult
+GetFullHistoryOfVehicleWithVIN(vinToLookup: string): IHttpActionResult

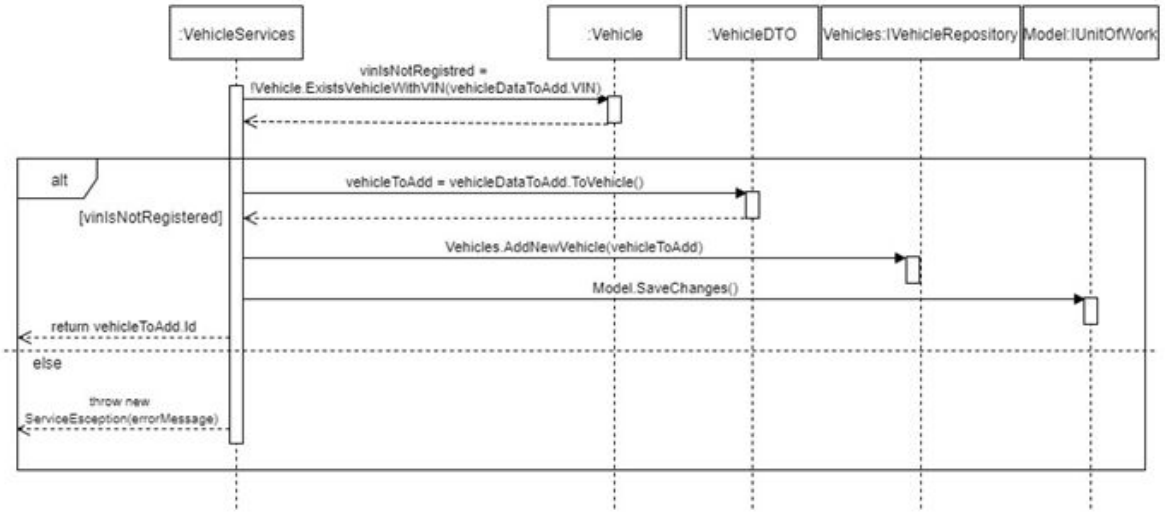


Diagramas de interacción

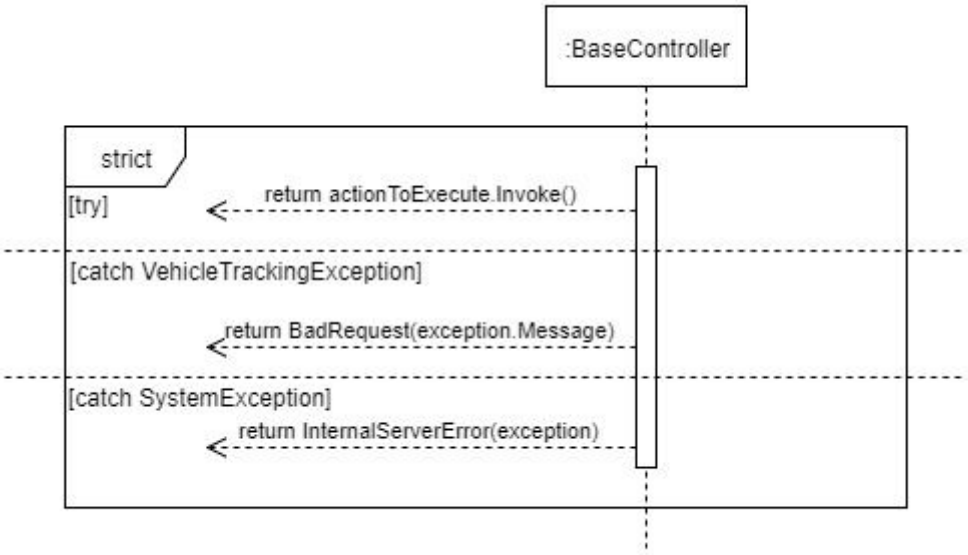
UsersController - AttemptToGetRegisteredUsers



VehicleServices - AttemptToAddVehicle



BaseController - ExecuteActionAndReturnOutcome



UserServices - GetRegisteredUsers

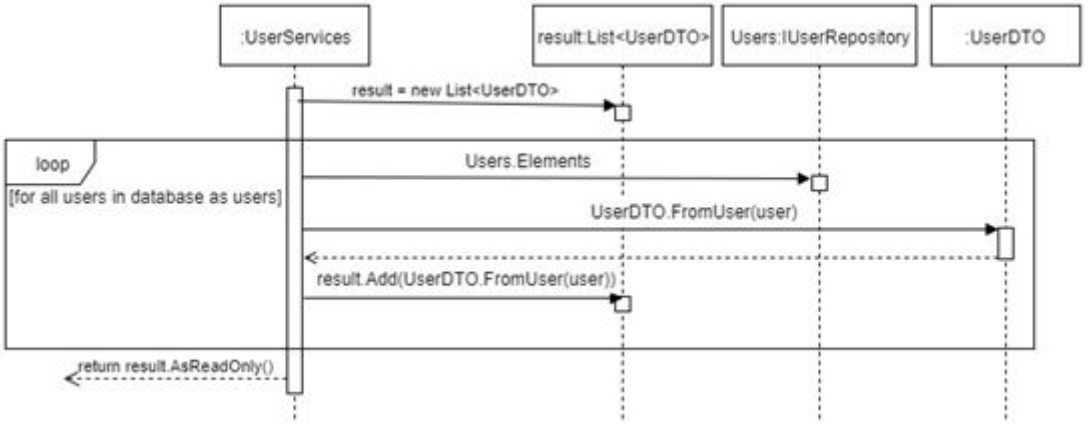


Diagrama de componentes

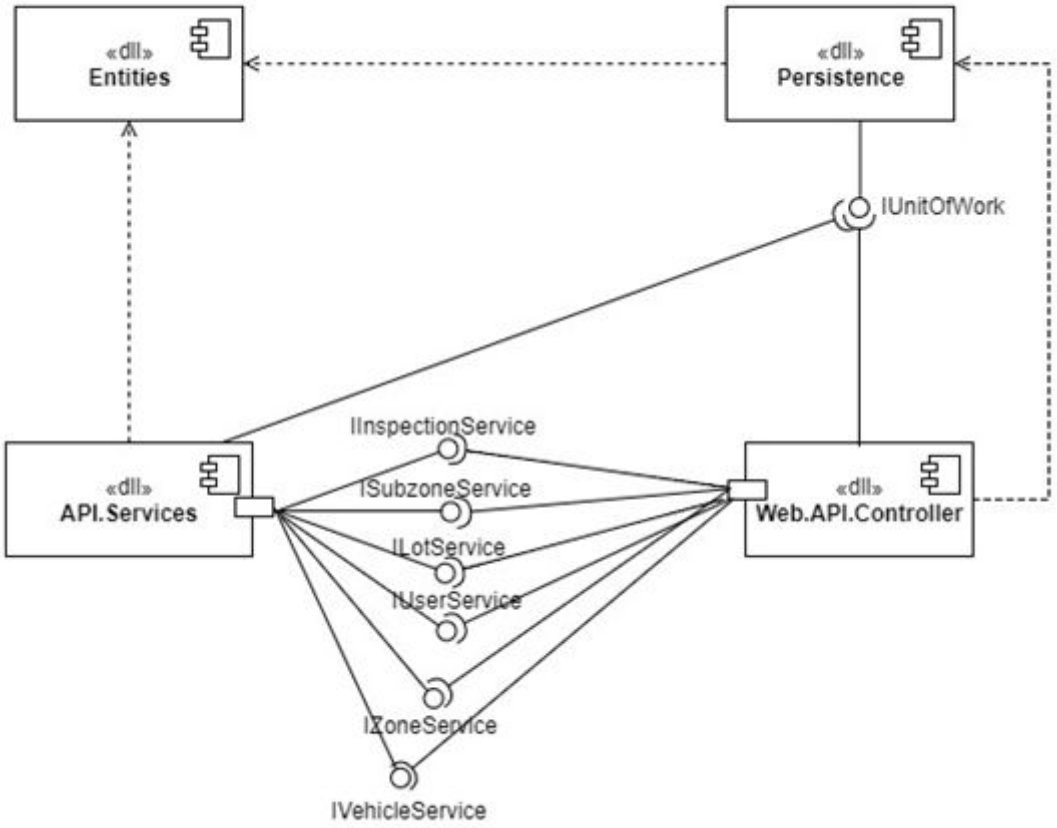
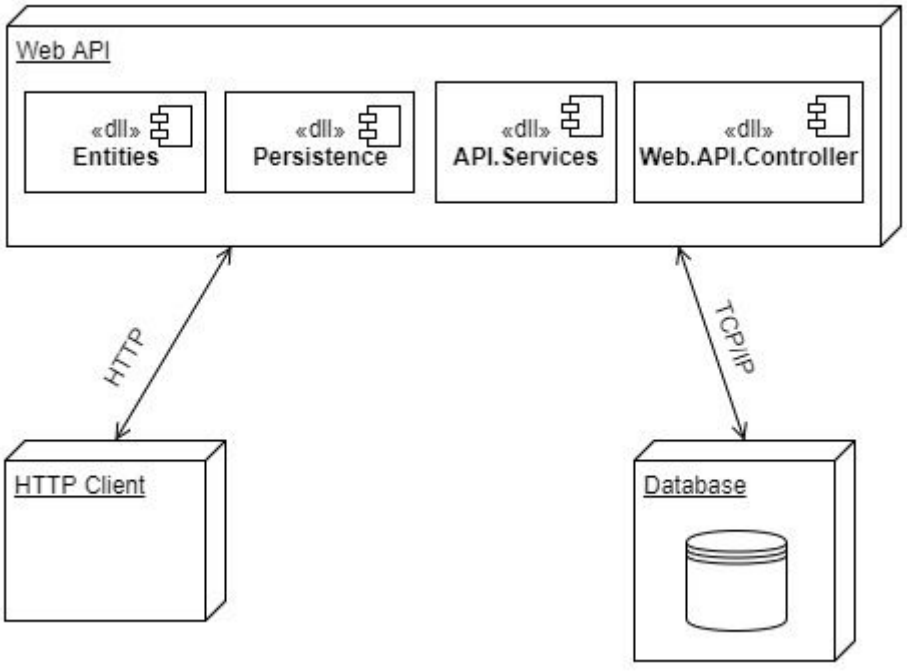
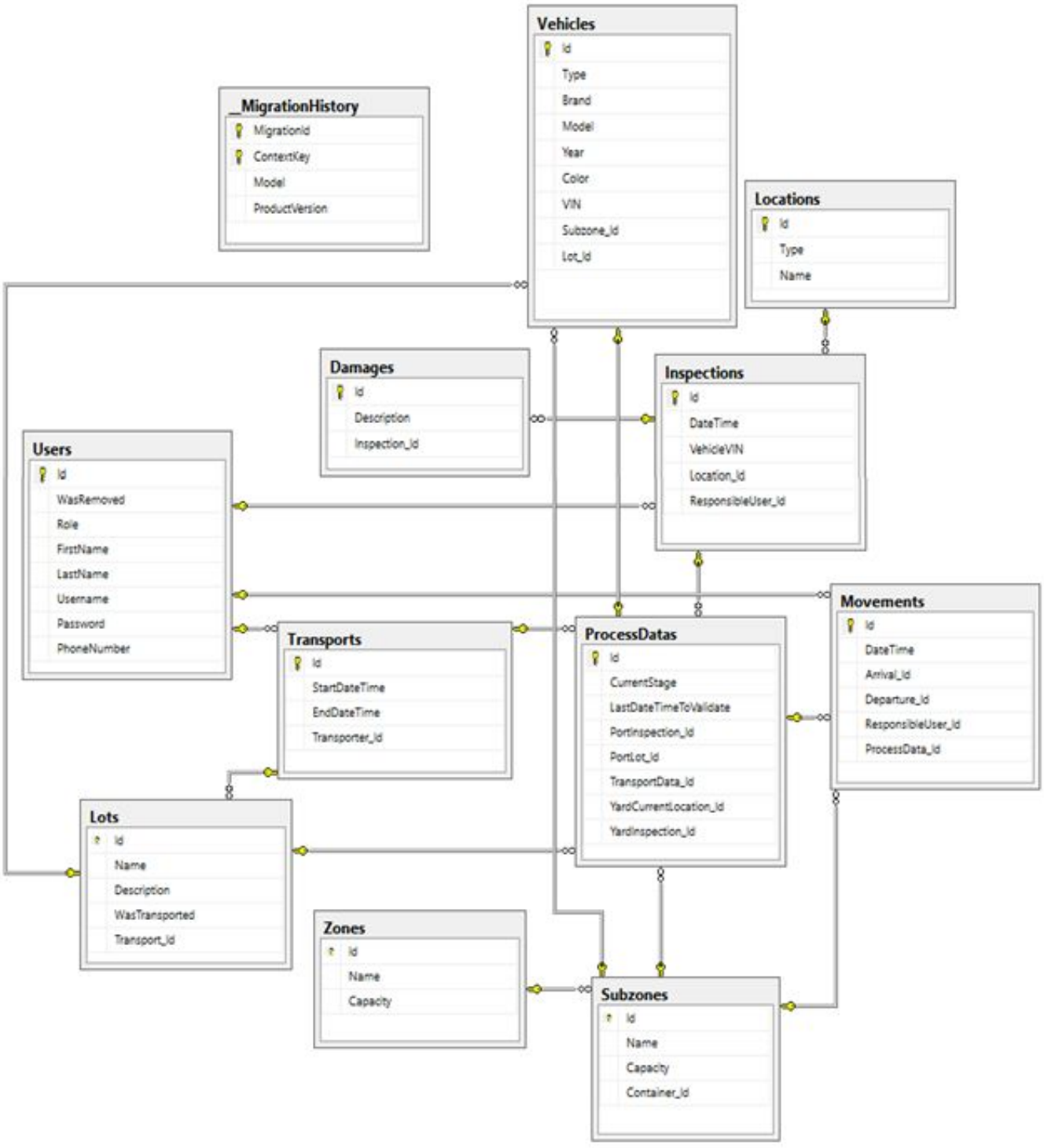


Diagrama de entrega



Modelo de tablas



Justificación del diseño

La solución concebida como parte del presente obligatorio está compuesta por, a grandes rasgos, cuatro paquetes que dividen el trabajo que debiera realizarse para poder resolver el problema planteado, más otros cuatro paquetes de pruebas unitarias sobre los mismos. Se intentará entonces a continuación brindar un comentario general acerca de cada uno de ellos.

En primer lugar, existe un paquete (o *namespace*) denominado “Domain”. En éste se almacenaron las distintas clases e interfaces vinculadas a las entidades de la realidad presentada en la propuesta y a los elementos que ésta involucra; a saber, entidades “*Vehicle*”, “*Lot*”, “*Transport*”, entre otras. Se intentó lograr que, como regla general, cada clase fuera conocedora de sus datos, cuidando la exposición de los mismos a otras, y responsable de las validaciones vinculadas a la integridad y coherencia de los mismos. Para esto, se implementaron distintos métodos de validación de las distintas restricciones de negocio sobre las entidades, que serían llamados en los métodos *set* de las propiedades asociadas a las mismas, por ejemplo. Así, el objetivo general a grandes rasgos perseguido sería lograr concentrar la mayoría de estas restricciones sobre las propias entidades, lo cual se encuentra alineado con el patrón GRASP “Experto”, y no delegarlas a capas superiores (*Services*, etc.), lo que se habrá de abordar en mayor detalle más adelante.

Por otro lado, como punto particular de esta sección surge el asunto del manejo de los distintos datos asociados a las distintas etapas que va atravesando un vehículo a lo largo del proceso que lleva a cabo la empresa. Durante el proceso de desarrollo se contemplaron diversas alternativas: inicialmente, se pensó en que la clase “*Vehicle*” agregara un objeto de una cierta clase “*Stage*”, de la que heredarían tres clases distintas “*PortStage*”, “*TransportStage*” y “*YardStage*”. Éstas guardarían, cada una de ellas atributos privados con los datos de la etapa asociada a la misma, más expondrían a través de la clase padre propiedades para todas ellas, lanzando una excepción o retornando un valor nulo, por ejemplo, de corresponder esto (si por ejemplo se intenta acceder a la inspección de patio de un vehículo en el puerto, que de acuerdo a la lógica de negocio descrita en la propuesta resulta inválido). Asimismo, se pensó que cada una de estas clases podría entonces agregar al objeto asociado con la etapa anterior del mismo vehículo, manteniendo entonces el histórico para el estado actual del mismo, y a su vez contar con un método *“AdvanceToNextStage()”* que fuera el encargado de instanciar el objeto asociado a la etapa siguiente a la actual y ya setear esta propiedad, así como la asociada a la etapa anterior para la siguiente (con *this*), en el vehículo que correspondiera.

Esta alternativa, y si bien cuenta con la gran ventaja de que permite, mediante este último método, una gran flexibilidad en cuanto al manejo del agregado de nuevas etapas al proceso, simplemente impactando esto en deberse crear una nueva clase que cuente con las características deseadas, exponer éstas en la clase padre y modificar este método según corresponda para corregir el orden de las mismas, finalmente fue descartada. La razón de esto sería, como habitualmente, la relativa mayor complejidad de la misma frente al retorno que ofrecía, particularmente considerando que resulta poco probable que se agreguen nuevas etapas al proceso. Esto más allá del importante riesgo que se hubiera asumido de seguir en esta línea debido la utilización de *Entity Framework,* que es sabido habitualmente representa presenta problemas y limitaciones en el cargado de entidades que refieren a tipos padre (dado que el diseño propuesto era un pseudo-*Composite*, en definitiva), y en cualquier caso hubiera complejizado el manejo de las consultas a la base de datos. Algunos indicios de esto son visibles en el histórico de *commits* del obligatorio. Eventualmente, se evaluó brevemente tirar todos los datos a la clase "Vehicle", descartándose esto por considerarse que no era propia esta información de la entidad vehículo, además de resultar demasiado cargada en cuanto a responsabilidades esta clase en tal caso. Finalmente, se llegó a la opción finalmente perseguida, el concentrar los datos del proceso en una clase "*ProcessData*", de la cual "*Vehicle*" guarda una referencia, que se considera aceptable más que presenta una oportunidad de mejora a futuro, puesto que el agregado de una nueva etapa o el modificado de los datos de la misma impactarían directamente en esta clase.

Un segundo paquete sería el denominado "*Persistance*", que, como su nombre lo indica, contiene las distintas clases e interfaces involucradas en permitir un correcto almacenamiento en base de datos de las entidades del dominio. En esta ocasión nuevamente, y como ya fue mencionado anteriormente, se utilizó *Entity Framework* como librería para mediar entre la aplicación generada y una base de datos, de SQL Server en este caso. Así, esta herramienta generaría automáticamente el modelo de tablas en la base de datos, para luego efectuar consultas de obtención y manipulación de las entidades mediante las distintas clases que ofrece para ello. Esta interacción se encapsularía en una serie de clases "*Repository*", que contienen un objeto *context* y una serie de métodos que permiten trabajan sobre el mismo, heredando éstos, asimismo, de una clase "*GenericRepository<T>*" para evitar la repetición de código y permitir un mejor diseño. Se coordinaría luego la materialización de los cambios en la base de datos mediante una clase "*UnitOfWork*", que contiene referencias a los distintos *Repositories* y permite la llamada al método "*SaveChanges()*", si bien a pesar de ello se identifican ciertas ineficiencias en el cargado de las entidades, sobre todo respecto de la clase "*Vehicle*". Todas estas clases serían expuestas hacia otros paquetes mediante interfaces, aumentando la flexibilidad del diseño y amortiguando las dependencias entre las mismas.

De esta forma, se intentaría optimizar la conexiones necesarias con la misma, puesto que todos los *Repositories* serían instanciados, una vez fueran llamados desde los *Services*, con un mismo contexto. Curiosamente, se descubrió que presumiblemente por la introducción de esta clase se redujeron en una gran medida los problemas que anteirormente se debió enfrentar por la utilización de *Entity Framework* (que ni cursando Diseño de Aplicaciones 1 dos veces había sido posible descifrar hasta el momento). En este sentido, se considera una introducción positiva al diseño esta clase. Por otro lado, con respecto a la estrategia de cargado de las entidades, dada la naturaleza de la aplicación que llamaría a estos métodos, una API Web, se optaría por utilizar *Eager Loading,* dado que resultaría claro distinguir cuándo sería necesario cargar los datos y cuándo no (por ejemplo, un GET vs. un GET por *Id*). Sin embargo, a pesar de ello se detectan algunas ineficiencias en este sentido, a ser abordadas en un futuro segundo obligatorio, más allá de posibles mejoras a futuro: en este obligatorio no se utilizó ni se hizo el intento de utilizar *Lazy loading* debido a la dificultad que éste presenta, y debido a las bajas probabilidades de éxito que se evaluó esto tendría, pero debido a la introducción de una *Unit of work* se estima esto ahora resultaría posible, lo cual puede ser una opción interesante para explorar a futuro, de tenerse el tiempo suficiente.

En tercer lugar, se tiene un paquete "*API.Services*", que a grandes rasgos se encuentra dividido en dos partes: una serie de clases de servicios vinculadas a las distintas entidades del problema, con las que interactuarían los distintos *Controllers* de la API, y otra de objetos DTO (*data transfer objects*), que permitirían recibir y especificarían el formato de los distintos datos a leerse del cuerpo de las peticiones HTTP recibidas por la aplicación. Con respecto a éstos primeros, serían ellos los que ordenarían y efectuarían las distintas interacciones con los *Repositories* anteriormente mencionados, encargados de crear nuevas instancias de las entidades de dominio en base a datos recibidos en un DTO asociado, y llamar a su guardado en un *Repository* y al método *SaveChanges()*, por ejemplo.

Asimismo, en estas clases se manejarían distintas restricciones de negocio que tendrían que ver con cuestiones más globales, que involucran al estado actual del sistema; a saber, por ejemplo, que no se pueda registrar un usuario con un nombre ya ingresado o que no se pueda eliminar una zona que contiene subzonas dentro. Así, se intentaría generar mayor independencia de lo que son validaciones de la lógica del problema de la comunicación con la base de datos, generando mayor independencia entre estas partes y mayor flexibilidad en caso de que a futuro sea necesario cambiar cualquiera de ellas. Éstas clases, al igual que los *Repositories* respecto de ellas, serían expuestas a través de interfaces para minimizar dependencias, y además permitir y facilitar la generación de pruebas unitarias mediante *Mocking*.

Por otro lado, con respecto a las clases de *Data Transfer Objects*, éstas simplemente contendrían los campos que se esperaba recibir o enviar para una determinada entidad, sin ningún tipo de validación de los mismos (manejándose esta en la lógica del dominio o el Service correspondiente), más quizás algún que otro método auxiliar encargado del pasaje de entidades del dominio a DTO y viceversa. Se los decidió incluir en el mismo paquete que los *Services* dado que éstos dependen de los mismos, y cualquier otra clase que debiera utilizarlos necesitaría incluir forzosamente el potencial paquete conteniendo exclusivamente a los DTOs. Ésto más allá de que tampoco se buscaba generar un exceso de paquetes/proyectos en la solución, y espejando además una situación similar con el asunto de las excepciones propias generadas para la aplicación, considerada buena práctica al permitir evitar la captura silenciosa de excepciones del sistema no contempladas. Éstas se incluirían dentro del paquete asociadas a las mismas por idéntica razón, dado que serían lanzadas por clases dentro de él y para permitir su manejo serían necesarios en definitiva dos *usings,* corrección que costó algunos puntos en el anterior curso.

Como cuarto paquete tendríamos, finalmente, al paquete de la Web API propiamente dicha, que contendría las clases "*Controllers*", con métodos encargados de proveer una implementación a los distintos *endpoints* asociados a la aplicación, como punto de acceso de clientes a la misma. Como se mencionó anteriormente, éstos interactuarían con el resto de la lógica a través de los previamente descritos "*Services*", y con las restantes capas inferiores. Asimismo, en este paquete se encontrarían otras clases encargadas de dar soporte al requerimiento de la autenticación de usuarios registrados en el sistema, dada mediante la clase "*AuthorizationServerProvider*", la cual sería utilizada para controlar el acceso de los distintos roles de usuarios a ciertos *endpoints* con privilegios especiales.

Con respecto a los restantes paquetes, asociados a las pruebas unitarias de las distintas clases, se resolvió organizar a éstas en dos proyectos: uno conteniendo las pruebas unitarias de las clases de dominio y persistencia, que impactarían en una base de datos de *testing,* y otro de las clases de servicios y controladores, que buscarían aislarse de esto mediante *mocking*, para asegurar la unitariedad de pruebas. Por otro lado, finalmente, con respecto a la dependencia entre clases, como se mencionó anteriormente se intentó disminuir generando una serie de interfaces a la cual se acoplaría la capa superior de la aplicación, en lugar de al tipo concreto. Esto permitiría el testing con *mocking*, además de aumentar la flexibilidad de la aplicación en caso de desearse cambiar a futuro la implementación concreta utilizada. Sin embargo, el observador atento notará que a pesar de ello el objeto particular en el que es instanciada esta variable del tipo de la interfaz lo es en un

constructor sin parámetros de cada clase, los cuales son llamados en cadena desde el *Controller* (que es inicializado automáticamente por la API cuando se recibe una *request* HTTP), cuando claramente hubiera tenido más sentido utilizar una estrategia de inyección de dependencias. En cuanto a esto, se intentó dejar las distintas clases y tipos encaminadas hacia ello, si bien por un tema de limitaciones de tiempo y desconocimiento de la tecnología se decidió no implementarlo en este obligatorio, siendo por tanto una oportunidad de mejora para el siguiente. A pesar de ello, sin embargo, se considera que el diseño implementado en esta oportunidad es uno aceptable, que cumple con los principales lineamientos vistos en el curso anterior de forma satisfactoria.

Clean code

En este proyecto hicimos uso de los estándares que establece "*Clean Code*", de Robert C. Martin, para que el código que se escribe sea más limpio y por lo tanto más mantenible. De esta manera, cuando otro desarrollador, o nosotros mismos, vuelve a leer el proyecto después de un tiempo, le será muy simple entender el código antes escrito.
Para evidenciar el uso de este libro, decidimos separarlo en los capítulos que el mismo contiene.

Nombres

Se utilizan nombres descriptivos, claros y legibles. Si es el nombre de un método, el mismo debe representar la funcionalidad del mismo.

```
internal void SetTransportStartData(Transport transportToSet)
{
    ValidateVehicleIsInStage(ProcessStages.PORT);
    bool isValidTransportData = Utilities.IsNotNull(transportToSet) &&
        transportToSet.LotsTransported.Contains(PortLot);
    if (isValidTransportData)
    {
        TransportData = transportToSet;
        CurrentStage = ProcessStages.TRANSPORT;
    }
    else
    {
        string errorMessage = string.Format(CultureInfo.CurrentCulture,
            ErrorMessages.InvalidDataOnProcess, "Datos del transporte");
        throw new ProcessException(errorMessage);
    }
}
```

Funciones

Se debe poder inferir su comportamiento de un vistazo. Por lo tanto, deben ser cortas, y realizar una única cosa. Deben tener buenos nombres y la mínima cantidad de argumentos, más de 3 ya viola los principios del Clean Code.

Principalmente, se debe eliminar la duplicidad, el código repetido. Si hay dos funciones que realizan lo mismo, entonces debe haber una manera de encapsularlas en una sola.

```
internal bool IsReadyForTransport()
{
    return CurrentStage == ProcessStages.PORT &&
        Utilities.IsNotNull(PortInspection);
}
```

En nuestro caso, utilizamos la clase Utilities para encapsular todos aquellos métodos que realizan validaciones. De esta forma evitamos que las distintas clases tengan código repetido.

Comentarios

Si se necesita un comentario para aclarar algo entonces es señal de que se debería rediseñar.

Únicamente son apropiados para remarcar importancia de algo, aspectos legales, advertir consecuencias, por ejemplo. En el resto de los casos se puede generar confusión.

Tratamos de no incluir comentarios salvo que fueran extremadamente necesarios obteniendo como resultado final un código libre de ellos.

Formato

Se debe generar un estándar de codificación. Se puede utilizar uno ya hecho como el de Java por ejemplo, o elegir un conjunto de reglas dentro de un grupo de trabajo. En nuestro caso, se intentaron seguir las convenciones del lenguaje que establece la MSDN, utilizando *properties* en lugar de *setters* y *getters,* utilizando nombres que comenzaran por mayúsculas, etc.

La importancia no reside en qué reglas se toman, si no en que todos los que trabajen en el proyecto se atenga a las mismas.

Objetos y estructura de datos

Se debe diferenciar entre objetos y estructuras de datos, viendo cuándo es preferible una que la otra. Los objetos esconden sus datos y exponen funciones para utilizarlos, las estructuras de datos exponen sus datos pero no tienen funciones para manipularlos.

Procesar errores

Se debe separar el manejo de errores de la lógica del proyecto. Dicho de otra manera, se esconde la verdadera funcionalidad del código.

Se utilizan bloques try-catch-finally que identifican los puntos del programa que pueden producir una excepción. Son menos invasivas a nivel de código que los códigos de retorno.

Se deben evitar los retornos *null*.

```
private void ValidateVehicleIsInStage(ProcessStages expectedStage)
{
    if (CurrentStage != expectedStage)
    {
        throw new ProcessException(ErrorMessages.InvalidOperationOnVehicle);
    }
}
```

Tratando de seguir con esto, en nuestro proyecto realizamos las validaciones a nivel de dominio lanzando excepciones, creadas por nosotros, que luego son controladas a nivel de la aplicación API REST.

Límites

Se debe definir de forma clara la frontera entre el código y los paquetes de terceros o desarrollados por otros equipos. De esta manera, se minimizan las partes de nuestro código que dependen de elementos externos.

Pruebas de unidad

Clean Code establece que hay trabajar utilizando la metodología de TDD (Test Driven Development). De esta manera fue que se realizó la implementación de nuestro proyecto.

Clases

Primero se deben declarar constantes públicas, luego las estáticas privadas, variables de instancia privadas, y por último los métodos. Además los métodos privados deben estar junto a los públicos que los utilizan.

Se debe mantener la encapsulación de las clases, pequeñas con pocas responsabilidades.

Las nuevas funcionalidades se deben introducir extendiendo el sistema, NO modificando código existente. Las clases deben depender de abstracciones, ya que están expresan conceptos, y luego las concretas que dependen de ellas tienen los detalles de implementación. De otra manera, habría mayor impacto de cambio.

Emergencia

Reglas de Kent Beck:
• Ejecutar todos los tests: Estos verifican si el sistema hace lo que se quiere que haga. Al construir un sistema testeable, las clases son más simples, tienen un único propósito y se reduce el acoplamiento.
• Eliminar la duplicación: Generando menos trabajo, menos riesgo y complejidad.
• Expresar la intención del programador: Se deben escoger buenos nombres, funciones y clases pequeñas y tests bien escritos.
• Minimizar el número de clases y métodos: Siguiendo las recomendaciones anteriores uno se puede exceder creando demasiadas clases pequeñas. Hay que tener cuidado y mantener un número reducido de clases.

Durante la refactorización de puede aplicar todo nuestro conocimiento para mejorar el diseño: aumentar la cohesión, reducir el acoplamiento, separar responsabilidades, reducir funciones y clase, escoger nombres mejores, etc.

TDD (Test Driven Development)

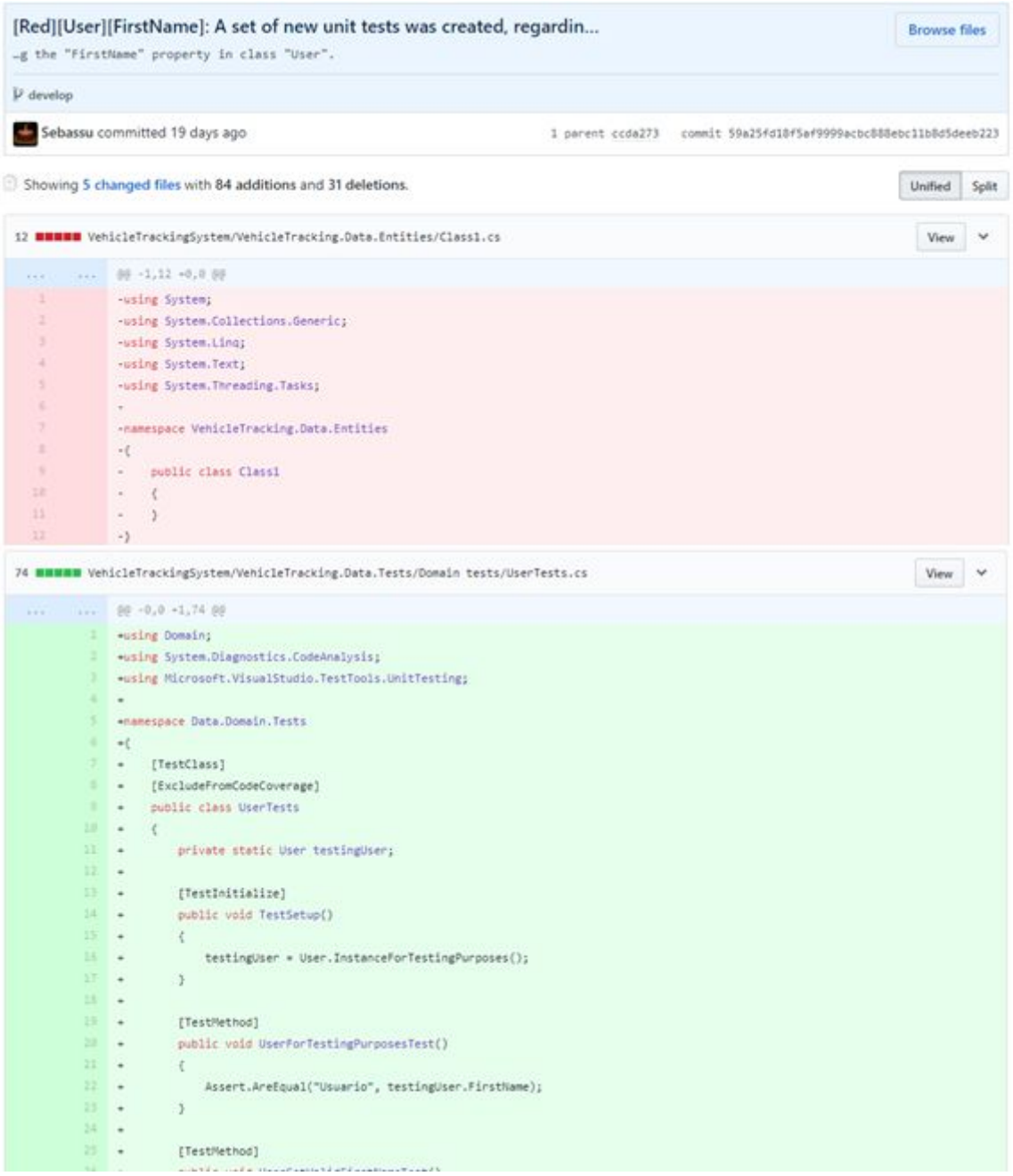
Para la realización de este trabajo se utilizó la forma de trabajo llamada Test Driven
Development, mejor conocida como TDD. La misma consiste en realizar las pruebas previo a implementar las funciones para asegurarnos de que estamos realizando solo lo pedido y que se está haciendo bien.

Esta metodología consta de tres partes, Red, Green y Refactor. La etapa Red es la primera y consiste en escribir solamente el test de la función que deseamos probar. Luego viene la etapa Green, en ella se escribe el código mínimo necesario para que la prueba hecha en la etapa anterior pase, sin importar si el código es desprolijo, ineficiente o muy complejo. Por último, en la etapa de Refactor nos encargamos de eliminar todos los code smells sin alterar el comportamiento interno del código.

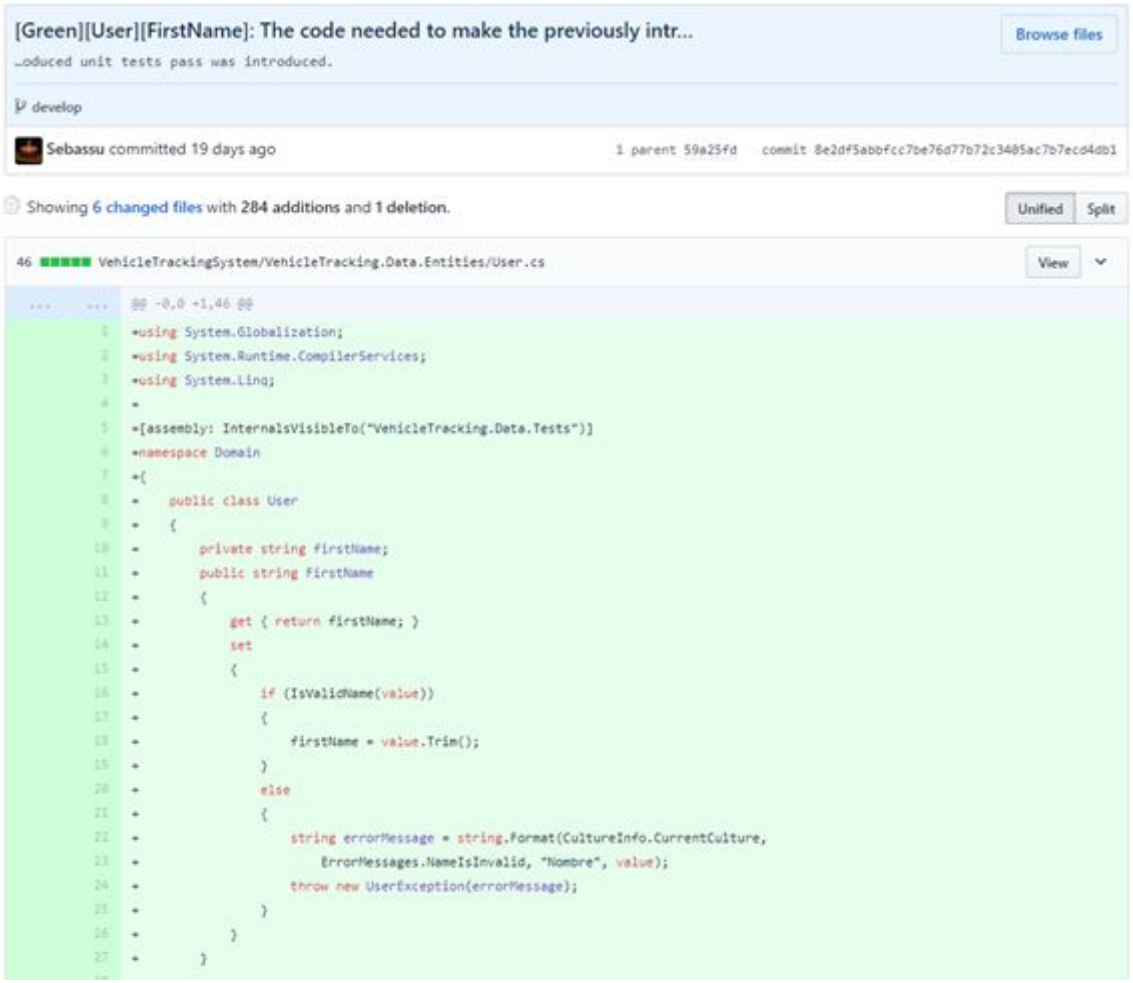
A continuación, se mostrará evidencia de la utilización de TDD a lo largo del proyecto. Solo se presentarán algunos casos ya que no es posible mostrar todas las funcionalidades implementadas.

Evidencia Nº 1

Etapa Red



Etapa Green



[Green][User][FirstName]: The code needed to make the previously intr...

Browse files

...oduced unit tests pass was introduced.

develop

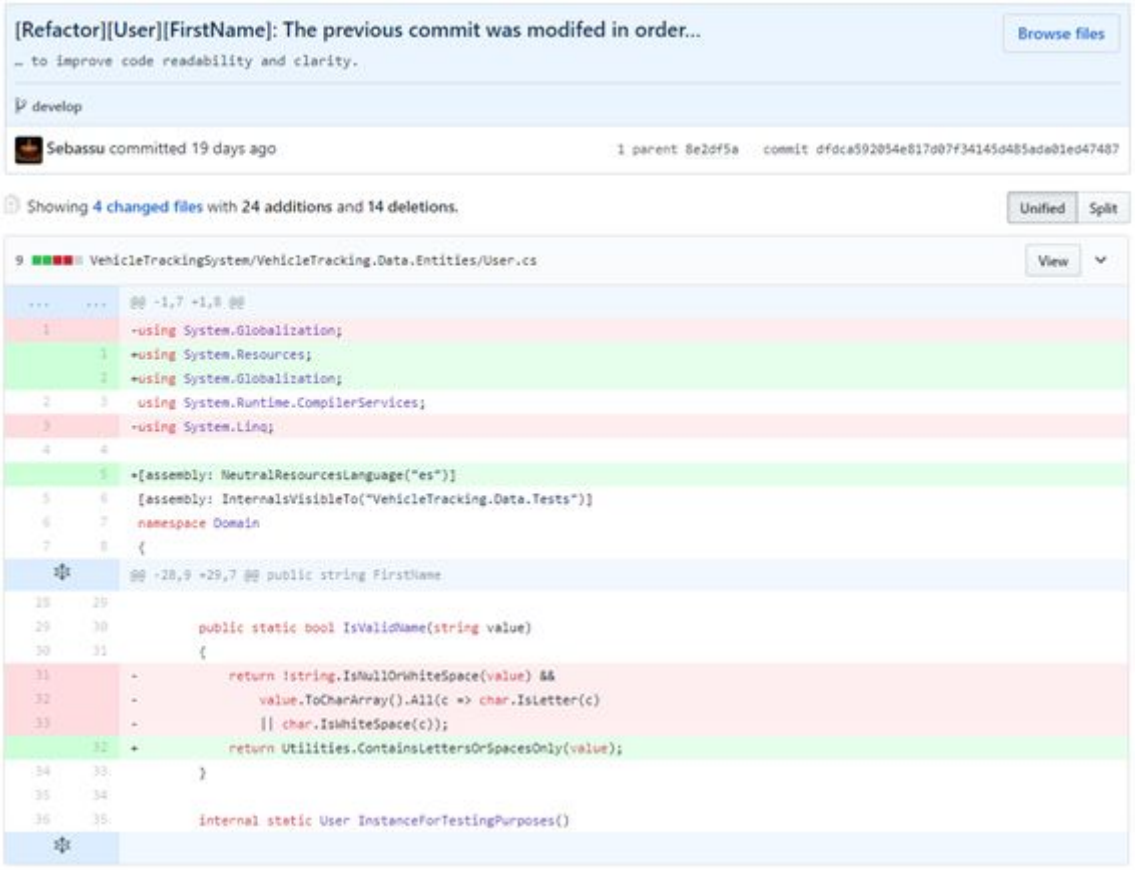
Sebassu committed 19 days ago 1 parent 59a25fd commit 8e2df5abbfcc7be76d77b72c3405ac7b7ecd4db1

Showing 6 changed files with 284 additions and 1 deletion. Unified Split

46 VehicleTrackingSystem/VehicleTracking.Data.Entities/User.cs View

```
@@ -0,0 +1,46 @@
+using System.Globalization;
+using System.Runtime.CompilerServices;
+using System.Linq;
+
+[assembly: InternalsVisibleTo("VehicleTracking.Data.Tests")]
+namespace Domain
+{
+    public class User
+    {
+        private string firstName;
+        public string FirstName
+        {
+            get { return firstName; }
+            set
+            {
+                if (IsValidName(value))
+                {
+                    firstName = value.Trim();
+                }
+                else
+                {
+                    string errorMessage = string.Format(CultureInfo.CurrentCulture,
+                        ErrorMessages.NameIsInvalid, "Nombre", value);
+                    throw new UserException(errorMessage);
+                }
+            }
+        }
```

Refactor



[Refactor][User][FirstName]: The previous commit was modifed in order...

Browse files

… to improve code readability and clarity.

develop

Sebassu committed 19 days ago 1 parent 8e2df5a commit dfdca592054e817d07f34145d485ada01ed47487

Showing 4 changed files with 24 additions and 14 deletions. Unified Split

9 VehicleTrackingSystem/VehicleTracking.Data.Entities/User.cs View

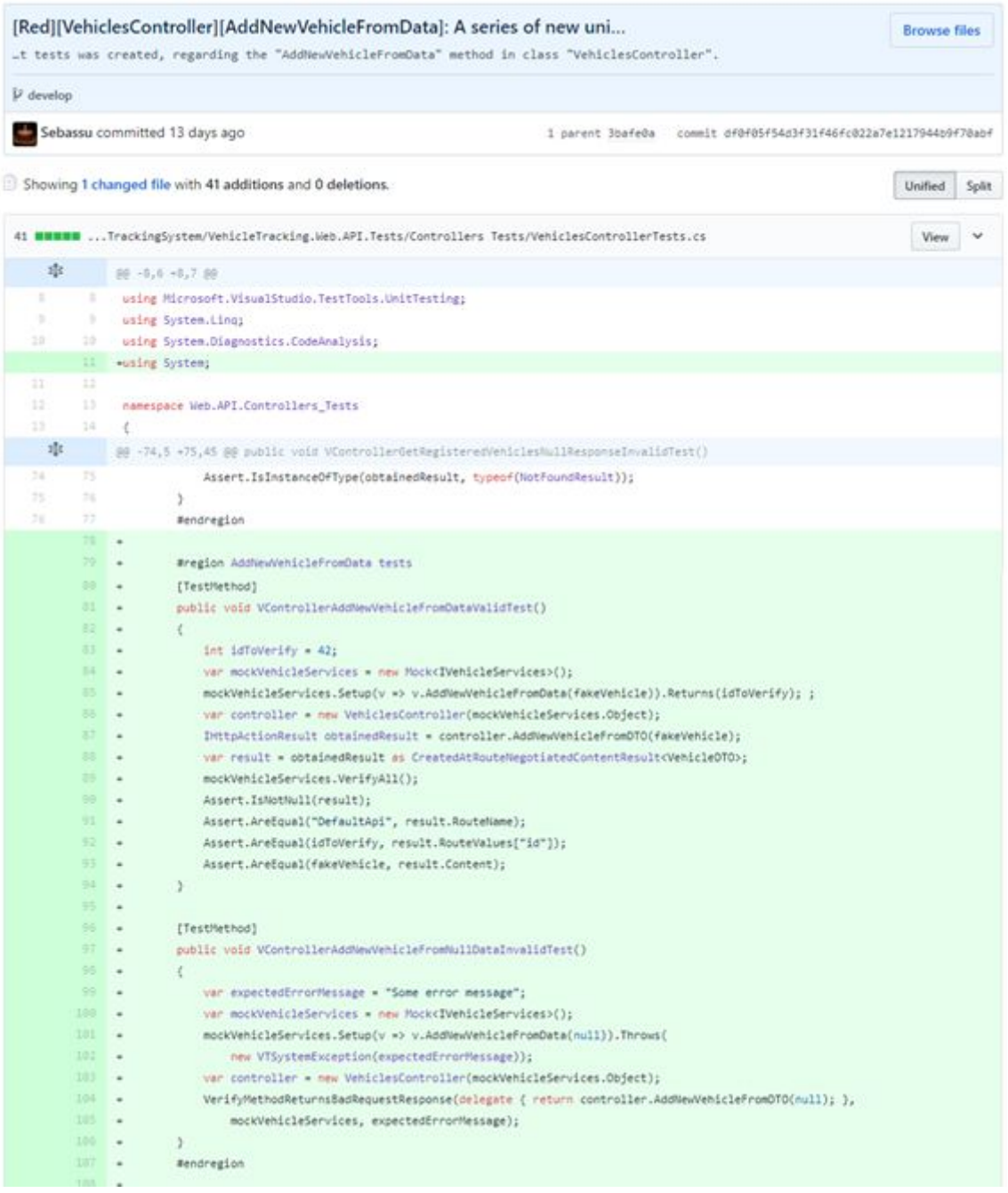
```
@@ -1,7 +1,8 @@
-using System.Globalization;
+using System.Resources;
+using System.Globalization;
 using System.Runtime.CompilerServices;
-using System.Linq;

+[assembly: NeutralResourcesLanguage("es")]
 [assembly: InternalsVisibleTo("VehicleTracking.Data.Tests")]
 namespace Domain
 {
@@ -28,9 +29,7 @@ public string FirstName
         public static bool IsValidName(string value)
         {
-            return !string.IsNullOrWhiteSpace(value) &&
-                value.ToCharArray().All(c => char.IsLetter(c)
-                || char.IsWhiteSpace(c));
+            return Utilities.ContainsLettersOrSpacesOnly(value);
         }

         internal static User InstanceForTestingPurposes()
```

Evidencia Nº 2

Etapa Red



[Red][VehiclesController][AddNewVehicleFromData]: A series of new uni...

Browse files

…t tests was created, regarding the "AddNewVehicleFromData" method in class "VehiclesController".

develop

Sebassu committed 13 days ago 1 parent 3bafe0a commit df0f05f54d3f31f46fc022a7e1217944b9f70abf

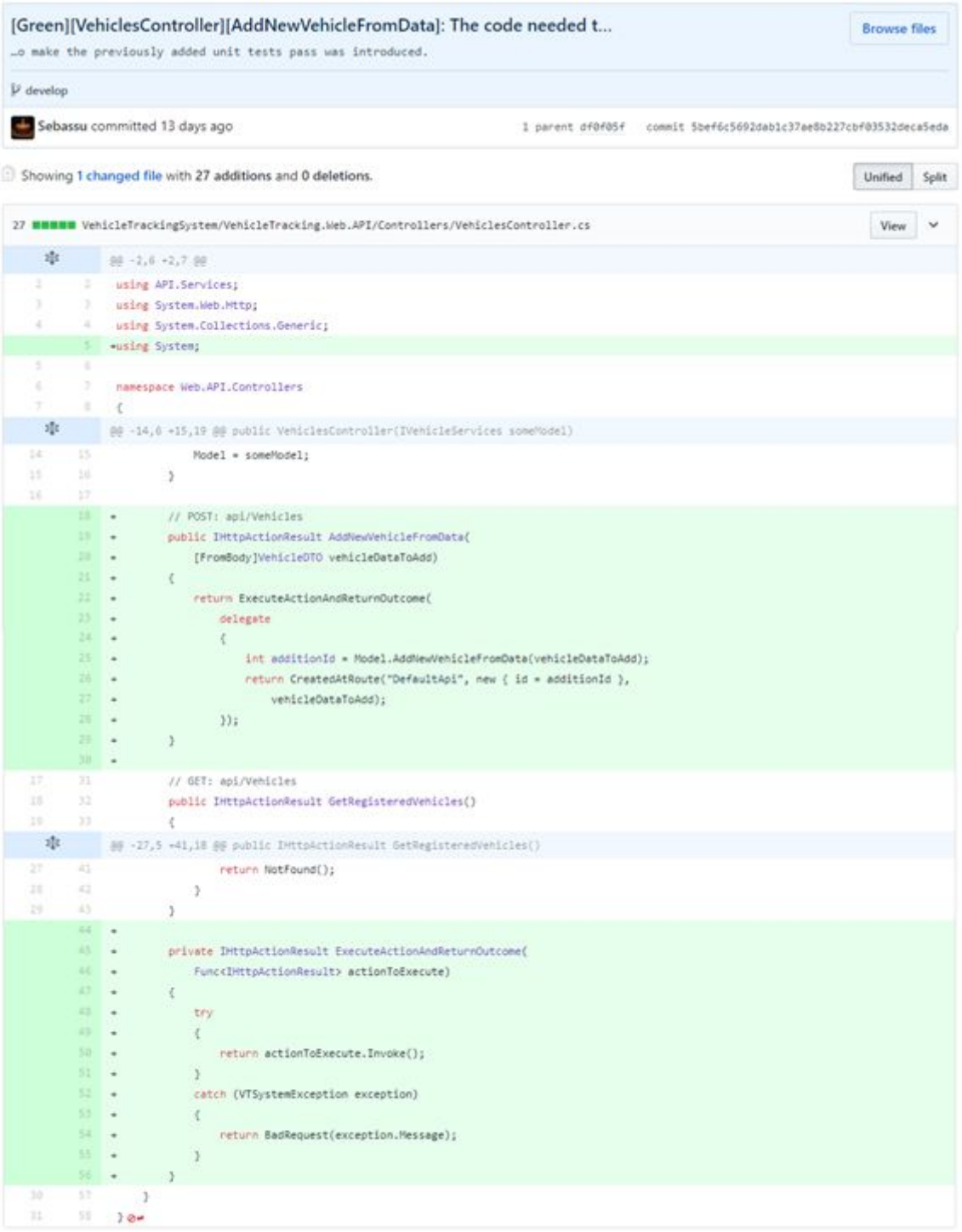
Showing 1 changed file with 41 additions and 0 deletions. Unified Split

41 ■■■■■ ...TrackingSystem/VehicleTracking.Web.API.Tests/Controllers Tests/VehiclesControllerTests.cs View

```
@@ -8,6 +8,7 @@
using Microsoft.VisualStudio.TestTools.UnitTesting;
using System.Linq;
using System.Diagnostics.CodeAnalysis;
+using System;

namespace Web.API.Controllers_Tests
{
@@ -74,5 +75,45 @@ public void VControllerGetRegisteredVehiclesNullResponseInvalidTest()
            Assert.IsInstanceOfType(obtainedResult, typeof(NotFoundResult));
        }
        #endregion
+
+       #region AddNewVehicleFromData tests
+       [TestMethod]
+       public void VControllerAddNewVehicleFromDataValidTest()
+       {
+           int idToVerify = 42;
+           var mockVehicleServices = new Mock<IVehicleServices>();
+           mockVehicleServices.Setup(v => v.AddNewVehicleFromData(fakeVehicle)).Returns(idToVerify); ;
+           var controller = new VehiclesController(mockVehicleServices.Object);
+           IHttpActionResult obtainedResult = controller.AddNewVehicleFromDTO(fakeVehicle);
+           var result = obtainedResult as CreatedAtRouteNegotiatedContentResult<VehicleDTO>;
+           mockVehicleServices.VerifyAll();
+           Assert.IsNotNull(result);
+           Assert.AreEqual("DefaultApi", result.RouteName);
+           Assert.AreEqual(idToVerify, result.RouteValues["id"]);
+           Assert.AreEqual(fakeVehicle, result.Content);
+       }
+
+       [TestMethod]
+       public void VControllerAddNewVehicleFromNullDataInvalidTest()
+       {
+           var expectedErrorMessage = "Some error message";
+           var mockVehicleServices = new Mock<IVehicleServices>();
+           mockVehicleServices.Setup(v => v.AddNewVehicleFromData(null)).Throws(
+               new VTSystemException(expectedErrorMessage));
+           var controller = new VehiclesController(mockVehicleServices.Object);
+           VerifyMethodReturnsBadRequestResponse(delegate { return controller.AddNewVehicleFromDTO(null); },
+               mockVehicleServices, expectedErrorMessage);
+       }
+       #endregion
```

Etapa Green



[Green][VehiclesController][AddNewVehicleFromData]: The code needed t...

Browse files

…o make the previously added unit tests pass was introduced.

develop

Sebassu committed 13 days ago 1 parent df0f05f commit 5bef6c5692dab1c37ae8b227cbf03532deca5eda

Showing 1 changed file with 27 additions and 0 deletions. Unified Split

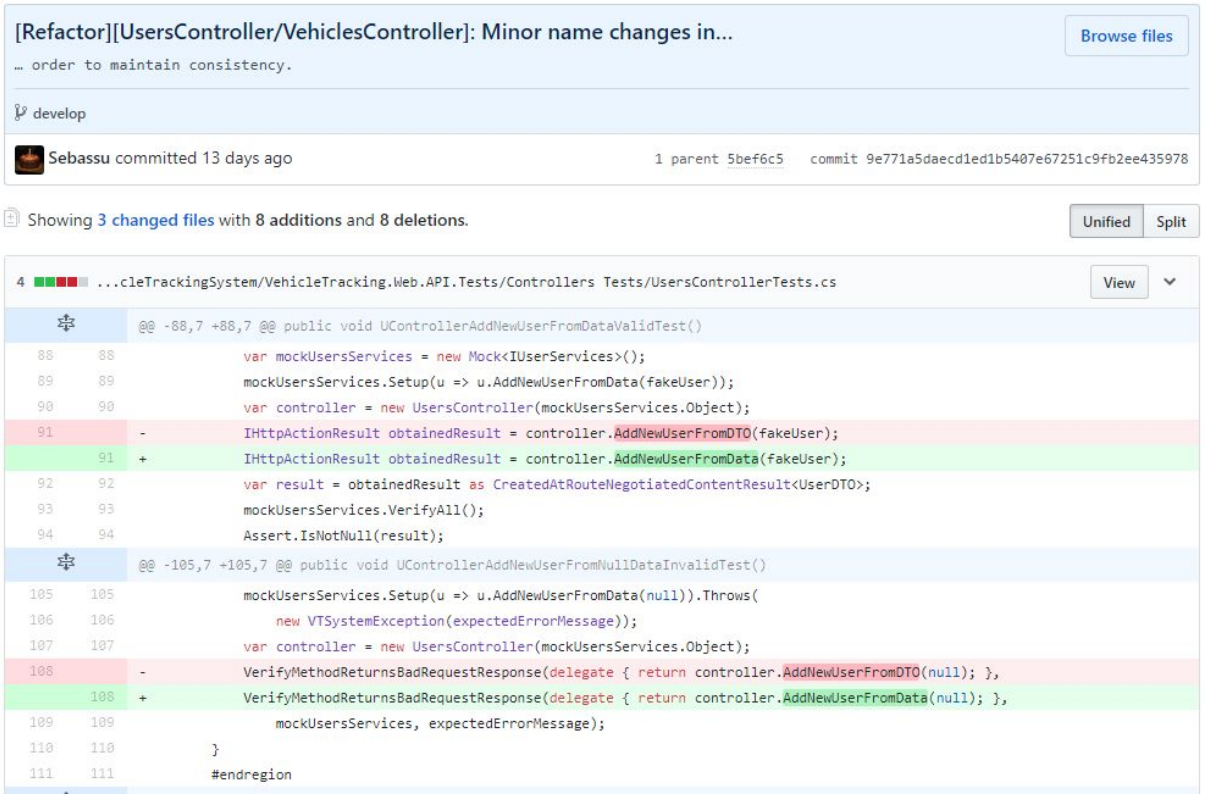
27 VehicleTrackingSystem/VehicleTracking.Web.API/Controllers/VehiclesController.cs View

```
@@ -2,6 +2,7 @@
using API.Services;
using System.Web.Http;
using System.Collections.Generic;
+using System;

namespace Web.API.Controllers
{
@@ -14,6 +15,19 @@ public VehiclesController(IVehicleServices someModel)
            Model = someModel;
        }

+       // POST: api/Vehicles
+       public IHttpActionResult AddNewVehicleFromData(
+           [FromBody]VehicleDTO vehicleDataToAdd)
+       {
+           return ExecuteActionAndReturnOutcome(
+               delegate
+               {
+                   int additionId = Model.AddNewVehicleFromData(vehicleDataToAdd);
+                   return CreatedAtRoute("DefaultApi", new { id = additionId },
+                       vehicleDataToAdd);
+               });
+       }
+
        // GET: api/Vehicles
        public IHttpActionResult GetRegisteredVehicles()
        {
@@ -27,5 +41,18 @@ public IHttpActionResult GetRegisteredVehicles()
                return NotFound();
            }
        }
+
+       private IHttpActionResult ExecuteActionAndReturnOutcome(
+           Func<IHttpActionResult> actionToExecute)
+       {
+           try
+           {
+               return actionToExecute.Invoke();
+           }
+           catch (VTSystemException exception)
+           {
+               return BadRequest(exception.Message);
+           }
+       }
    }
}
```

Refactor

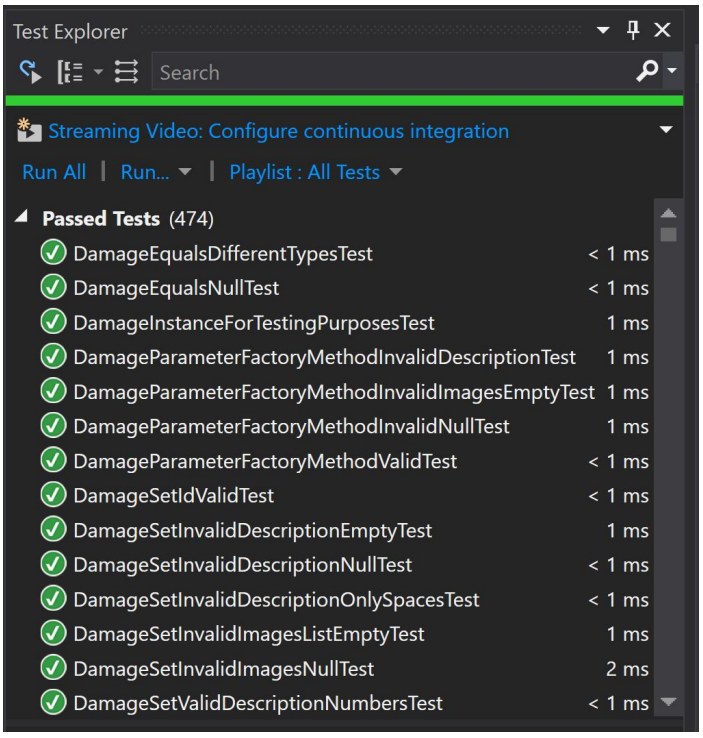


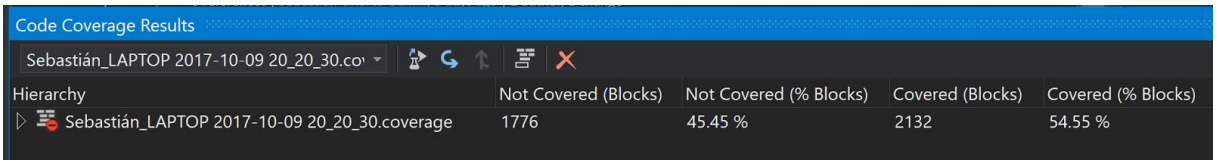
Cobertura de pruebas

Uno de los beneficios de haber utilizado la metodología de TDD es que la realizar las pruebas antes de implementar las funcionalidades nos aseguramos que todas las mismas hayan sido probadas.

Sin embargo, cerca de la fecha de entrega se dejó de hacer TDD estrictamente debido a que estábamos muy justos con los tiempos de trabajo por lo que la cobertura de las pruebas bajó significativamente.

Es por esta razón que la cobertura de las pruebas es relativamente baja en comparación con lo que debería de haber sido (casi 100%).





De haber implementado todo el trabajo utilizando TDD, hubiese sido muy beneficioso ya que si en el futuro se modifica la funcionalidad de determinado método mientras se pretendía refactorizar el código por ejemplo, las pruebas comenzarán a fallar, y encontrar la solución será más sencillo.

Este “beneficio” solo lo tendremos en aquellas funcionalidades que efectivamente implementamos siguiendo la metodología mencionada.

Evaluación del proyecto

El trabajo consistió en realizar una implementación del API REST que se nos detalló tanto en la letra del obligatorio como en el foro de aulas. Fue un desafío realizarla ya que si bien sabíamos como implementar la mayoría del mismo por el hecho de que era muy similar al obligatorio de una materia anterior, ninguno había trabajado con APIs.

Esto implicó que tuviéramos una etapa de investigación para aprender a utilizar esta nueva tecnología lo cual nos consumió bastante tiempo. Sin embargo, una vez que aprendimos pudimos avanzar bastante rápido con ello.

Con respecto al equipo de trabajo, el mayor desafío al cual nos vimos enfrentadas fue no conocernos entre nosotros. Esto provocó que en el comienzo trabajáramos más lento ya que no conocíamos como era la forma de trabajar del otro.

Además teníamos combinar nuestros horarios para ver si coincidiamos algun dia de la semana para poder juntarnos. Por suerte encontramos dos dias en la en los que nos podíamos juntar, y el haber tenido algunos feriados durante los días de trabajo también ayudó a poder terminar el proyecto sin ningún problema.

En cuanto a la metodología de trabajo, lo que necesitó más dedicación fue el desarrollo del código ya que era una aplicación bastante extensa. Si bien en un comienzo le dedicamos bastante tiempo al diseño del sistema decidimos comenzar a desarrollar codigo antes de haberlo terminado ya que sabíamos que el mismo iba a cambiar a medida que avanzáramos con el proyecto.

Aunque nos basamos en el diseño que habíamos propuesto, tuvimos que modificar una serie de cosas debido a que nos dimos cuenta que el diseño que habíamos propuesto en un principio nos iba a dificultar la implementación de determinadas funcionalidades o la persistencia de los datos, tal como fue explicado anteriormente en la justificación del diseño.

Haciendo una reflexión acerca de la forma trabajada, creemos que la misma fue bastante acertada. Aunque al final no utilizamos la metodología de TDD debido a que pensamos que no nos daría el tiempo para terminar todo el proyecto, pudimos terminarlo dentro de las fechas estipuladas y consideramos que obtuvimos un buen resultado final más allá de la cobertura de pruebas, la cual podría haber sido mejor.