Object – Oriented Technology

Chapter 7: Database Constraints

Suad Alagić

Springer 2016

DATABASE CONSTRAINTS

```
public class Database
{ public Database(String name);
  public final boolean isOpen();
  public final void open()
        ensures this.isOpen();
  public final void close()
        requires this.isOpen();
  public final Object lookUp(String name)
        requires this.isOpen();
  public final boolean bind(Object x, String name)
        requires this.isOpen(),
        requires this.lookUp(name)= null,
        ensures this.lookUp(name).equals(x);
// invariant etc.
}
```

TRANSACTION CONSTRAINTS

```
public abstract class Transaction <T extends Database>
{ protected T schema;
  public Transaction<T>(T schema);
  public final void start()
        requires schema.invariant;
  public final void commit()
        ensures schema.invariant;
  public final void abort()
        ensures schema.invariant;
  public abstract void execute()
        requires schema.invariant,
        ensures schema.invariant;
}
```

QUERIES and CONSTRAINTS

```
public abstract class Query< P extends Comparable,
                             T extends P>;
{ public Query<P,T>();
  public abstract boolean qualification(T x);
  public final Collection<T>
                             selectFrom(Collection<T> S);
  public final Collection<P>
                             selectAndProject(Collection<T> S);
  public final OrderedCollection<P>
                             selectProjectAndOrder(Collection<T> S);
invariant
  (∀ Query<P,T> this) (∀ Collection<T> S) (∀ T x)
  (this.selectFrom(S).contains(x) ⇐
        S.contains(x) ∧ this.qualification(x))
  // other clauses
}
```

SAMPLE SCHEMA

```
public interface Employee {
  String name();
  String ssn();
  Float salary();
  Department department();
  void assignDepartment(Department d)
        ensures this.department().equals(d);
  invariant
  (∀ Employee X,Y)
  (X.equals(Y) ⇐ X.ssn().equals(Y.ssn()));
}
```

SAMPLE SCHEMA

```
public interface Department {
  Integer deptNum();
  Collection<Employee> employees();
  Float allocatedPayroll();
  void addEmployee (Employee e)
        ensures this.employees().contains(e);
  void removeEmployee (Employee e)
        requires this.employees().contains(e);
  invariant
  (∀ Department X,Y)
  (X.equals(Y) ⇐ X.deptNum().equals(Y.deptNum());
}
```

SAMPLE SCHEMA

```
public class Corporation extends Database {
  public interface Employee
{ . . .}
  public interface Department
{. . . }
  public class EmployeeCollection
        implements Collection<Employee> {
  public void add (Employee e);
  }
  public class DepartmentCollection
        implements Collection<Department> {
  public void add (Department d);
  }
```

SAMPLE SCHEMA cont.

EmployeeCollection dbEmployees;
DepartmentCollection dbDepartments;

invariant
$(\forall$ Employee W) $(\forall$ Department Y)
(dbEmployees.contains(W) $\Leftarrow$
dbDepartments.contains(Y) $\wedge$
Y.employees().contains(W),
dbDepartments.contains(Y) $\Leftarrow$
dbEmployees.contains(W) $\wedge$
W.department().equals(Y));
}

SAMPLE TRANSACTION

```
public class HireTrans extends Transaction<Corporation> {
  public HireTrans (Corporation corp,
                    Employee emp, Department dept)
  {....}
public void execute()
  requires corp.dbDepartments.contains (dept)
  { emp.assignDepartment (dept);
  dept.addEmployee (emp);
  corp.dbEmployees.add (emp);
  }
}
```

SAMPLE SCHEMA (LINQ)

```
public abstract class Schema: DataContext {
// . . .
}

public class StockMarket : Schema {
    private [Table]<Stock> Stocks
      { public get { return GetTable<Stock>(); } }
    private [Table]<Broker> Brokers
      { public get { return GetTable<Broker>(); } }
// schema methods
}
```

SAMPLE SCHEMA (LINQ)

```
[Table(Name="Stocks")]
public class Stock {
   [Column(IsPrimaryKey = true, CanBeNull = false)]
   public int code { get; set; }
   [Column]
   public int value { get; set; }
// . . .
}
```

SAMPLE SCHEMA (LINQ)

```
[Table(Name="Brokers")]
public class Broker {
    [Column(IsPrimaryKey = true, CanBeNull=false)]
    public int id { get; set; }
   public ICollection< Stock > brokerStocks {. . . }
// . . .
}
```

SAMPLE SCHEMA (LINQ)

```
[Table(Name = "OwnedStocks")]
internal class OwnedStock {
    [Column(IsPrimaryKey=true,CanBeNull=false)]
    public int code { get; set; }
    [Column](IsPrimaryKey=true,CanBeNull=false)]
    public int id { get; set; }
//. . .
}
```

TRANSACTION CLASS

```
public abstract class Transaction< T > where T : Schema {
    public void execute() { . . . }
    public abstract void update();
// other methods
}
```

SAMPLE TRANSACTION

```
public class UpdateStock : Transaction<StockMarket> {
    Stock stk;
    int newVal;
    public override void update () {
      stk.value = newVal;
      schema.SubmitChanges();
    }

    public UpdateStock(StockMarket sm,
                       Stock stk, int newVal) : base(sm){
      this.stk = stk; this.newVal = newVal;
    }
}
```

SAMPLE SCHEMA

```
class Stock {
string stockId();
float price();
}
class Broker {
string brokerId();
string name();
Set<Stock> stocks();
}
class StockMarket: Schema {
[SpecPublic] private Set<Stock> stocks;
[SpecPublic] private Set<Broker> brokers;
// constraints etc. }
```

SCHEMA CONSTRAINTS

invariant
$\forall\{s_1, s_2 \in$ stocks: $(s_1.stockId() = s_2.stockId()) \Rightarrow s_1.equals(s_2)\}$;
invariant
$\forall\{b_1, b_2 \in$ brokers: $(b_1.brokerId() = b_2.brokerId()) \Rightarrow b_1.equals(b_2)\}$;
invariant
$\forall$ { b $\in$ brokers: $\forall$ {sb $\in$ b.stocks():
$\exists$ {s $\in$ stocks: (sb.stockId() = s.stockId())}}};
invariant
$\forall$ { s $\in$ stocks: s.price() $>$ 0};
// public methods for insertions, updates, and deletions of stocks and brokers
}

SAMPLE TRANSACTION

class StockMerge: Transaction< StockMarket > {

void update(Stock s_1, s_2)

requires ∃ **unique** { s ∈ stocks: s.stockId()=s_1.stockId()}
requires ∃ **unique** { s ∈ stocks: s.stockId()=s_2.stockId()}

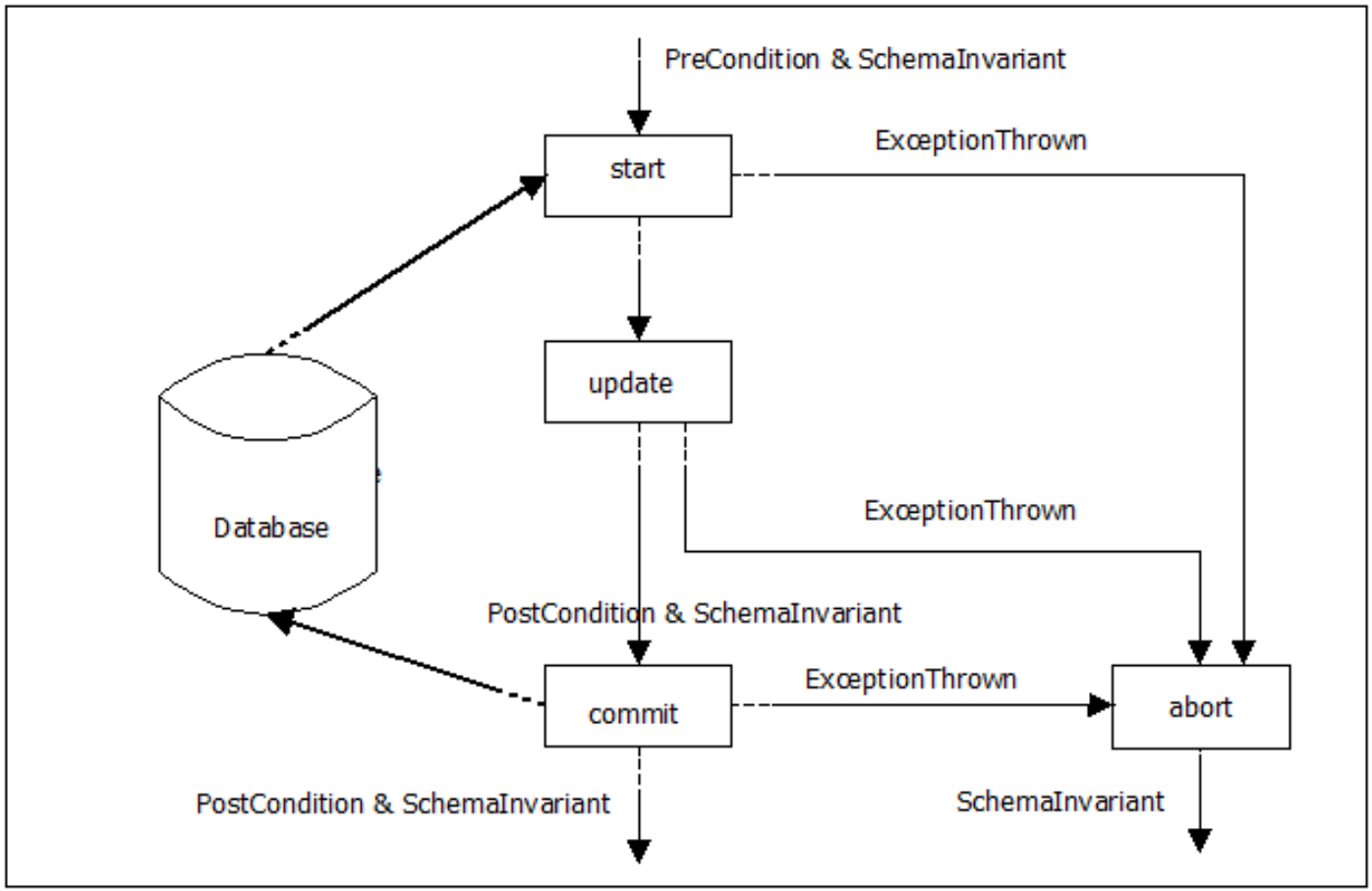
ensures ∃ **unique** { s ∈ stocks: ¬ (s ∈ **old**(stocks)) ∧
(s.price()=(s_1.*price*() + s_2.*price*())/2 }
ensures ∀ {s ∈ stocks: (s.stockId() ≠ s_1.*stockId*())∧
(s.stockId() ≠ s_2.stockId()) }
ensures ∀{s ∈ **old**(*stocks*):
((*s.stockId*() ≠ s_1.*stockId*())∧
(*s.stockId*() ≠ s_2.*stockId*()) ⇒ s ∈ stocks) }
ensures ∀ { b_1 ∈ **old**(brokers): ∃ {**unique** b_2 ∈ brokers:
(b_2.*Id*() = b_1.*Id*()) ∧ ∀ { s ∈ b_1.*stocks*():
(*s.stockId*() ≠ s_1.*stockId*() ∧ *s.stockId*() ≠ s_2.*stockId*())
⇒ s ∈ b_2.*stocks*() }}}

// other constraints }

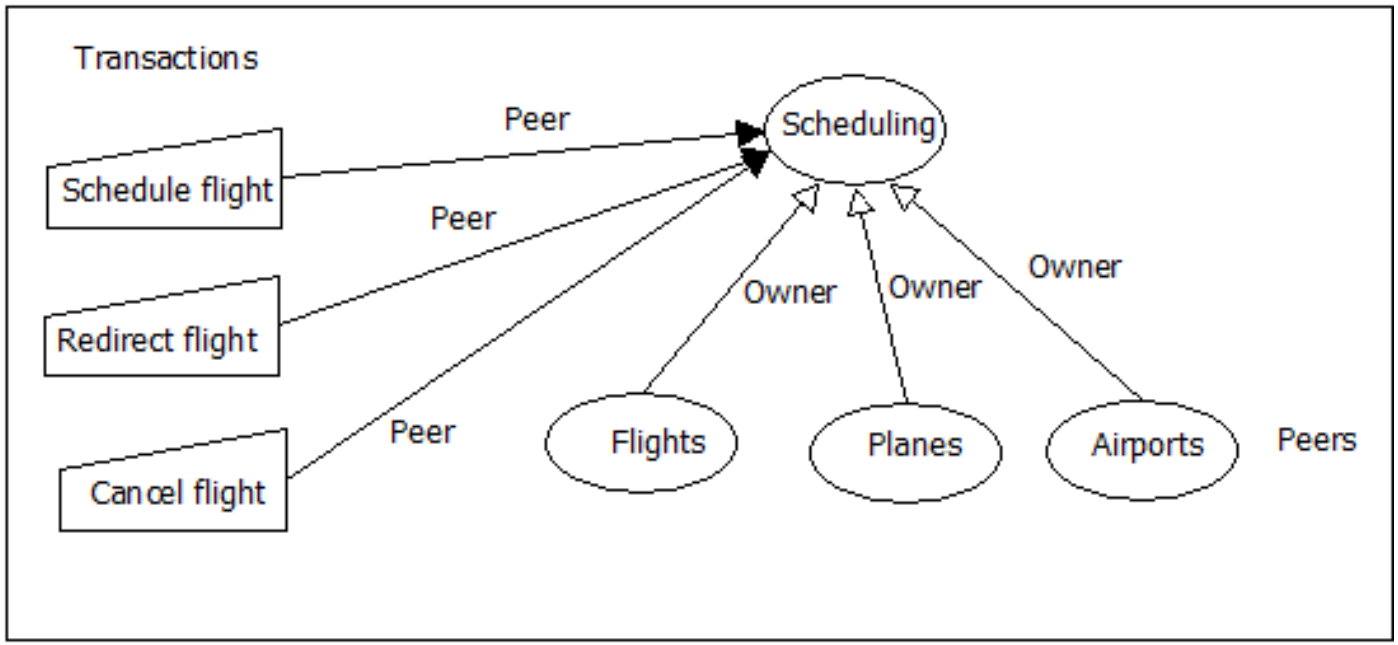
ACID TRANSACTIONS

- **Atomicity**
- **Consistency**
- **Isolation**
- **Durability**

TRANSACTION EXECUTION



OWNERS and PEERS



SAMPLE SCHEMA

```
public interface Schema {. . .}
public class Transaction <T> where T: Schema {
 [SpecPublic][Peer] protected T! schema;
public Transaction(T! schema){ this.schema = schema;}
}
```

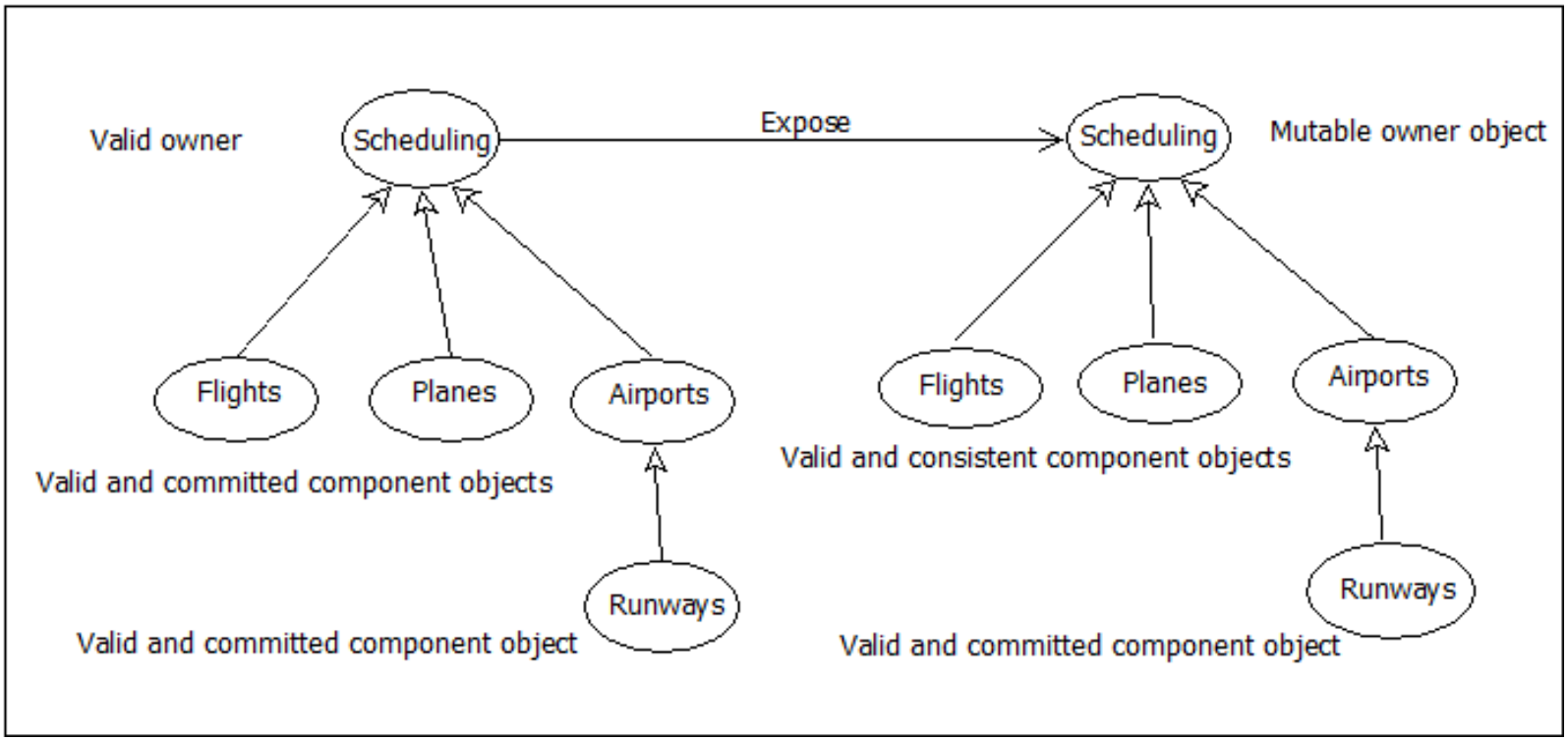
EXPOSE BLOCK

expose(flight scheduling){
close airport;
cancel all flights to or from the closed airport;
}

OBJECT STATES

- **Valid** object state – object invariants hold
- **Mutable** object state – object invariants are not required to hold
- **Consistent** object state – the object is in a **valid** state and
  - the object does not have an owner or
  - the owner is in a **mutable** state
- **Committed** object state – the object is in a **valid** state and
  - the object has an owner
  - the owner is also in a **valid** state.

LEVELS of CONSISTENCY



SAMPLE SCHEMA

public class FlightScheduling: Schema {

[**SpecPublic**][**Rep**] [**ElementsRep**] **private** List<Airplane!>! airplanes;

[**SpecPublic**][**Rep**] [**ElementsRep**] **private** List<Airport!>! airports;

[**SpecPublic**][**Rep**] [**ElementsRep**] **private** List<Flight!>! flights;
// constraints
}

SAMPLE CONSTRAINTS

invariant to $\neq$ from;

invariant departureTime $<$ arrivalTime;

invariant DateTime.Now $>$ arrivalTime $\Rightarrow$
this.flightStatus = FlightStatus.Idle;

invariant DateTime.Now $<$ departureTime $\Rightarrow$
this.flightStatus = FlightStatus.Idle;

invariant DateTime.Now $\geq$ departureTime $\wedge$
DateTime.Now $\leq$ arrivalTime $\Rightarrow$
this.flightStatus = FlightStatus.TakeOff $\vee$
this.flightStatus = FlightStatus.Flying $\vee$
this.flightStatus = FlightStatus.Landing;

SAMPLE TRANSACTION

```
public class ScheduleFlightTransaction:
                        Transaction<FlightScheduling> {
public Flight? scheduleFlight (String! flightId,
                String! toAirportCode, String! fromAirportCode,
                DateTime departure, DateTime arrival,
                Airplane! plane)
// constraints
{// transaction body }
}
```

TRANSACTION CONSTRAINTS

requires toAirportCode $\neq$ fromAirportCode;
requires $\forall$ {int i $\in$ (0: schema.Flights.Count);
schema.Flights[i].FlightId $\neq$ flightId };
requires $\exists$ **unique** {int i $\in$ (0: schema.Airplanes.Count);
schema.Airplanes[i].equals(plane)};
requires $\exists$ **unique** {String code $\in$ ValidCodes.airportsCodes;
code = toAirportCode};
requires $\exists$ **unique** {String code $\in$ ValidCodes.airportsCodes;
code = fromAirportCode };
requires departure $<$ arrival;

modifies schema.flights;

ensures $\exists$ **unique** {int i $\in$ (0: schema.Flights.Count);
schema.Flights[i].FlightId = flightId };

TRANSACTION CONSTRAINTS

requires $\exists$ **unique** {Flight! flight $\in$ schema.Flights;
flight.FlightId = flightId};
requires $\forall$ {Flight! flight $\in$ schema.Flights;
flight.FlightId = flightId $\Rightarrow$
flight.departureTime $>$ DateTime.Now };

modifies schema.flights;

ensures $\forall$ {Flight! flight $\in$ schema.Flights;
flight.FlightId $\neq$ flightId };

TRANSACTION CONSTRAINTS

requires $\exists$ **unique** {Flight flight $\in$ schema.flights;
flight.FlightId = flightId $\wedge$
(flight.FlightStatus $\neq$ FlightStatus.Landing)};
requires $\forall$ {Flight flight $\in$ schema.flights;
flight.FlightId = flightId $\Rightarrow$ flight.from $\neq$ newDest };

modifies schema.flights;

ensures $\forall$ {Flight! flight $\in$ schema.Flights;
flight.FlightId = flightId $\Rightarrow$ flight.to = newDest };

```
[Pure] public List<Flight!>? flightsDepartureBetween
                (DateTime beginDateTime,
                DateTime endDateTime)
requires beginDateTime < endDateTime;
requires beginDateTime > DateTime.Now;
ensures ∀ {Flight! f ∈ result;
            f.departureTime ≥ beginDateTime ∧
            f.departureTime < endDateTime };
{// method body }
```

CONSTRAINTS and QUERIES

```
// open db
IEnumerable<Flight> flights =
from Flight flight ∈ db
where flight.departureTime ≥ beginDateTime ∧
        flight.departureTime < endDateTime
select flight;
// close db;
```

```
// open db
IList<Flight!>? flights =
db.Query<Flight!>(delegate(Flight! f) {
return (f.departureTime ≥ beginDateTime ∧
        f.departureTime < endDateTime); };
// close db;
```

SAMPLE CLASS

```
public class Airport {

[SpecPublic] private String code;
[Additive] protected int numRunways;
[SpecPublic] [Rep] [ElementsRep] protected
List<Runway!>! runways;
// methods and constraints
}
```

SAMPLE CONSTRAINTS

invariant numRunways $\geq 1 \wedge$ numRunways ≤ 30;
invariant runways.Count = numRunways;
invariant /* No multiple occurrences of the same flight in runways*/

METHOD CONSTRAINTS

public virtual void addRunway(Runway! runway)
modifies runways, numRunways;
ensures $\exists$ {Runway! r $\in$ runways; r.equals(runway)};
{//code }

public virtual void closeRunway (Runway! runway)

modifies runways, numRunways;

ensures numRunways $>$ 0;
ensures numRunways = **old**(numRunways) - 1;
ensures $\forall$ { Runway! r $\in$ runways; $\neg$ r.equals(runway)};
{// code }

SPECIFICATION INHERITANCE

public class InternationalAirport: Airport {

invariant numRunways $\geq$ 10;
invariant $\exists$ {Runway! r $\in$ Runways; r.IntRunway };
// IntRunway is a boolen model field in Runway
// constructor, methods

public override void closeRunway (Runway! runway)
ensures numRunways $\geq$ 10;
ensures $\exists$ {Runway! r $\in$ runways; r.IntRunway };
{// code}
}

SPECIFICATION INHERITANCE

```
assert runways ≠ null;
additive expose((Airport)this){
    runways.remove(runway);
    numRunways–;
}
```

public List<Runway!>! Runways {
get { return runways;}
[**Additive**] set {
requires value $\neq$ **null**;
ensures runways = value;
ensures
/*no multiple occurrences of the same flight in runways*/
//code }
}

MODEL FIELDS

model bool IntRunway {
satisfies IntRunway = (length $\geq$ 80 $\wedge$ width $\geq$ 10);}

```
IList<Airport!>? airports =
db.Query<Airport!>(delegate(Airport! arp){
return (arp.Code = a.Code);};
assert airports = null;

IObjectSet? airportsSet =
db.Query(typeof(Airport!));
assert ∃ unique {Airport! arp ∈ airportsSet; arp.equals(a)};
```