Building Enterprise Database Applications

using Rational Rose, Java and Mr Architecture

A short course by Kade Hansson

Course Contents

- **UML, object-orientation and design patterns**
- **Java language and essential APIs**
- **Java GUI components and event model**
- **Java I/O and TCP/IP sockets**
- **JDBC, Servlets and JSPs**
- **Mr Architecture**

Session Summary

- What are the **ingredients** in building enterprise database applications?

• Modelling methodology and tools	UML & **Rose**
• Language platform and development environment	**Java** & IDE
• Development architecture and middleware	**Mr Architecture** (JDBC, Servlets, EJB)

- What is **UML**: the unified modelling language?
- What constitutes **object-orientation**?
- What are **design patterns** and what do they mean to me?

Motivation

- Why model?
  - *Would you build a bridge without blueprints?*
    - Models **communicate** complex systems within teams and to stakeholders
    - Models help ensure **sound structures** and architectures are built
- Why visual models?
  - *A picture’s worth a thousand words*
    - Visual abstractions help **comprehension**
    - Better hope of being **universally** understood

Motivation (continued)

- Why **object-orientation**?
  - Closest to real world while still obeying **simple principles**
  - Widely accepted among software engineering community
- Why **UML**?
  - **Rigorous** and **portable**
  - Has a widely understood **visual notation**
  - Modelling meta-language of choice for object-oriented or component systems
- Why **design patterns**?
  - Builds upon object-orientation
  - Promotes a higher-level kind of reuse: **model reuse**

Motivation (continued)

- Why **Rose**?
  - One of the best known UML tools (mentored by Booch, Rumbaugh et al.)
  - **Generates code and relational schemas** from component models
- Why **Java**?
  - **Write once, run anywhere**; client or server
  - Rich and expanding API set
- Why **Mr Architecture**?
  - Based on accepted Java standards (JDBC, Servlets, EJB, transaction API)
  - Promotes **rapid development** by short circuiting EJB
  - Highly **efficient**, **maintainable** and **scalable**

What is UML?

- Common misconception:
  - *UML is just another object modelling notation, like Booch and OMT*
- UML is a modelling meta-language- it is a method of describing notations
  - UML can describe Booch concepts
    (e.g. **Booch Object Scenario Diagram** is a UML Collaboration Diagram)
  - UML can describe OMT (Rumbaugh et al.) concepts
    (e.g. **Rumbaugh Class Diagrams** are basis of UML Class Diagrams)
  - UML can describe other concepts and diagrams not from Booch or OMT
    (e.g. **Jacobson Interaction Diagram** is basis of UML Sequence Diagram)
  - Booch, Rumbaugh and Jacobson work for Rational, the main proponents of UML
  - Allows general and domain specific extensions

UML & Rose

- All UML document types are supported by Rose Enterprise Edition:
  - **Collaboration Diagrams**- showing a scenario involving components (dyn.)
  - **Sequence Diagrams**- showing interactions between components over time (dyn.)
  - **State Diagrams**- showing possible state changes in components (dyn.)
  - **Activity Graphs**- showing flow of control in a single component action (dyn.)
  - **Use Case Diagrams**- showing how external actors interact with a system (st./dyn.)
  - **Object Diagrams**- showing possible configurations of a live system (st./dyn.)
  - **Class Diagrams**- showing relationships between classes of objects (static)
  - **Package Diagrams**- showing dependencies between packages (static)
  - **Component Diagrams**- showing the connections between subsystems (static)
  - **Deployment Diagrams**- showing how a system will operate in practice (static)

Rose & UML

- A Rose Model (extension **.mdl**) is a UML Model
- A Rose Model may contain **many diagrams** of each type
- A Rose Model is organised into a **hierarchial structure**
  (this structure is shown in the left pane of the application window)
- The first level of hierarchy is the level of modelling abstraction
  - This is not defined by UML, but suggested by the Rational Unified Process (RUP)
  - The intent is that you first model **business processes**, secondly **system processes**, thirdly decide on **system design** and finally lay out the **implementation**
- Rose, like UML, shares entities and associations between many diagram types
  - Changing an entity or relationship on one diagram will change it on all diagrams

Rose and Other Notations

- Before Rumbaugh, and later Jacobson, joined Rational, Rose used only Booch notations
- Today, Rose Enterprise Edition can change notations on the fly
- So, if you are more comfortable with Booch or OMT...
  - Use your preferred notation to construct the diagram
  - Change it to UML to express it to your team
- If you don’t have a preferred notation, but find UML "too much"...
  - Use UML but switch off or avoid applying details like:
    - Stereotypes and role names
    - Field lists and method lists

What Should I Model for a Database Application?

- Probably **Use Cases**
  - Maybe not business-level use cases unless these are complex to grasp
  - System-level use cases can help tease out object classes and packages
- Usually **Classes**
  - Where special-purpose classes are needed, it is helpful to model these.
    e.g. classes that fulfil application specific GUIs or IO requirements.
- Definitely **{persistent} Classes**
  - These will allow the construction of prototype beans and database schema from the model in Rose
  - By modelling this in one place, you reduce the possibility of implementation conflicts between database and application code

What Can I Avoid Modelling in a Database Application?

- Most dynamic models
  - Usually **Collaborations** (collaborations are usually straightforward)
  - Sometimes **Sequences** (if use cases are simple)
  - Definitely **States** (because any state machines will be simple)
  - Definitely **Activities** (because method bodies should be short and simple)
- Static models at inappropriate levels of abstraction
  - Definitely **Objects**
    (except to communicate particular problematic situations in team scrums)
  - Probably **Packages**, **Components** and **Deployment**
    (these are more appropriate to systems including their own middleware)

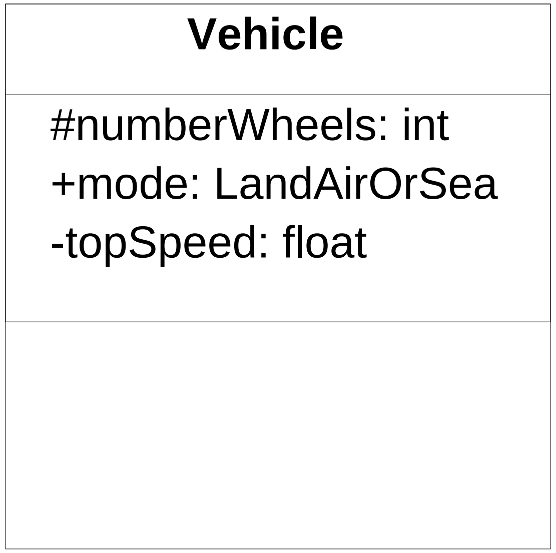
Why Object Models and not Relational Models?

- Object models are straightforward and clear
  - Object models are closer to our perception of reality +
  - Object models are not suited to "seek" operations –
  - **Object models are for people**
- Relational models are more rigorous and mathematic
  - Relational models lead to efficient "seek" operations +
  - Relational models lead to confusion –
  - **Relational models are for algebraists and computers**
- Relational models are a normalized form of object models
  - *Rose allows us to model using objects and implement using relational schemas*

What is Object Orientation?

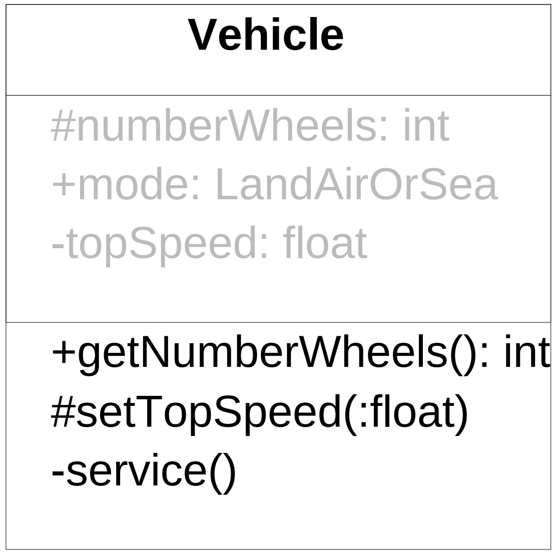
- It’s not just modelling using "objects"
  - Objects have fields (or attributes)
  - Objects have methods (or operations)
  - Objects can extend or alter the behaviour of other objects
    - **Inheritance**
  - Objects so extended can play the role of the objects they extend
    - **Polymorphism**
  - Operations in extended objects can replace those in their generic parent
    - **Overriding**
  - An operation can work on many different types (or classes) of object parameters
    - **Overloading**

Objects have Fields



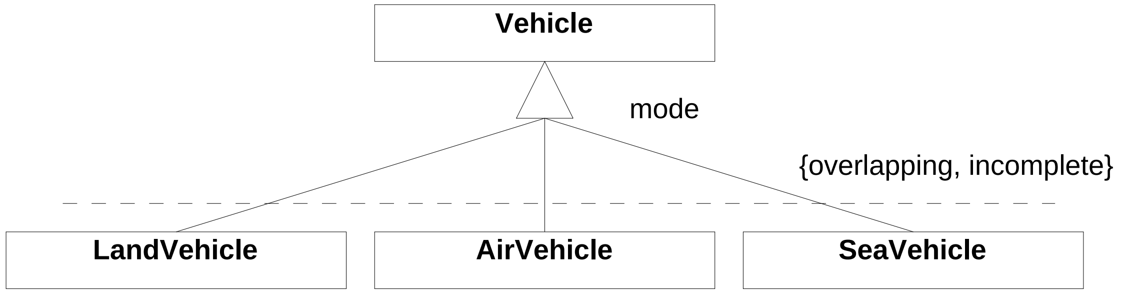
- Fields have visibility
  - **{private}**- visible only within object (UML shorthand: –)
  - **{protected}**- visible only in subclasses of an object (UML shorthand: #)
  - **{public}**- visible to all other objects (UML shorthand: +)

Objects have Methods



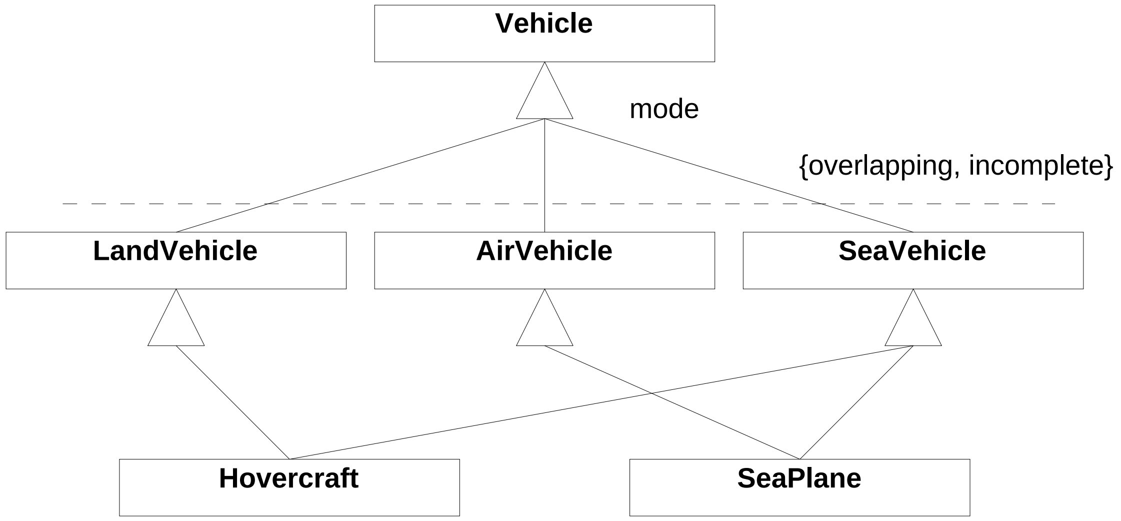
- Methods have visibility just like fields
- Methods can have parameters of particular types (including other classes)
- Methods can have return values of particular types (including other classes)

Inheritance



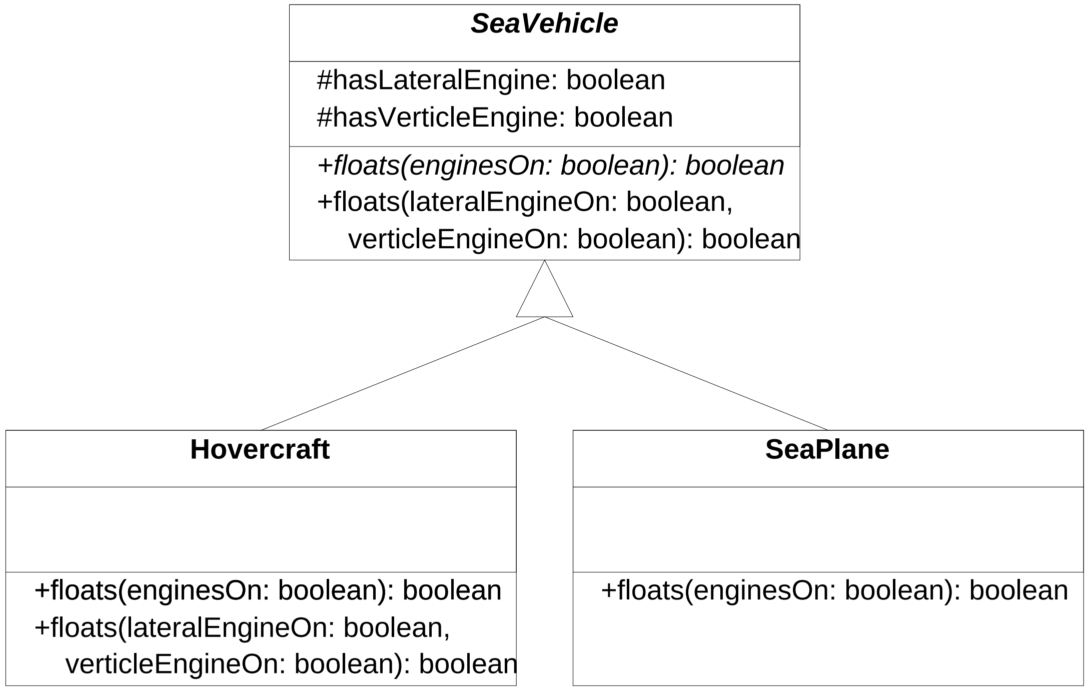
- Inheritance is a **generalization** relationship between classes of objects
- The division of a parent class into children may be accomplished by a **discriminator**:
  - *some field of the parent used to distinguish between the children* (e.g. "mode")
- A subset of generalizations based on a single discriminator may be:
  - ***complete*** **or** ***incomplete***
  - ***overlapping*** **or** ***disjoint***

Polymorphism and Multiple Inheritance



- LandVehicle, AirVehicle and SeaVehicle are each Vehicles (and behave like Vehicle)
  - **single inheritance**
- A Hovercraft is a LandVehicle and a SeaVehicle (and behaves like both)
  - **multiple inheritance**

Overriding and Overloading


SeaVehicle
#hasLateralEngine: boolean
#hasVerticleEngine: boolean
+floats(enginesOn: boolean): boolean
+floats(lateralEngineOn: boolean, verticleEngineOn: boolean): boolean
Hovercraft
+floats(enginesOn: boolean): boolean
+floats(lateralEngineOn: boolean, verticleEngineOn: boolean): boolean
SeaPlane
+floats(enginesOn: boolean): boolean

- floats() in SeaVehicle is **overridden** in Hovercraft (twice) and SeaPlane (once)
- floats() is **overloaded** in all classes

What are Design Patterns?

- They are patterns of structure or dynamics or both
- A design pattern may involve many classes, activities, collaborations etc.
- They may be:
  - general structures applicable across or within programming paradigms
    - **e.g. *Singleton*, *Facade*, *Model-View-Controller*, *Abstract Factory, Adaptor***
  - more specific structures or conventions used within languages or architectures
    - **e.g. *Beans*, Event *Listeners*, *Adapters***

- *It is useful to be able to recognise patterns in models so that common wisdom may be applied*
- *It is useful to be able to recognise the applicability of patterns when modelling*

$\Rightarrow$ **specific patterns need to be introduced by example**