One possible approach is to see it as a structuring technique for software:
- modules, units, packages, subsystems

Software objects combine data structures and functions:
- type ---- > “class” i.e. a template (ADT)
- instance --- > similar to a variable, exists in memory
- implement “information hiding” principle

An implementation technique for programs:
- encapsulation
- generalization and specialization (inheritance)
- naming and scoping, polymorphism
Object - Orientation

> On a more abstract level a system is a collection of distinct interacting objects ("things", "entities")

> An object has
  * structure - data, state
  * behavior - methods, functions, operations

> An object is characterized
  * by a number of operations (methods)
  * and a state which remembers the effect of these operations

> Objects can be seen on two levels:
  * Type – class – interface
  * Variable – instance – value / reference

Example / "Truck"

<table>
<thead>
<tr>
<th>Behavior / Methods, operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>load ()</td>
</tr>
<tr>
<td>unload ()</td>
</tr>
<tr>
<td>move ()</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attributes / state variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>int: load, weight;</td>
</tr>
<tr>
<td>Type: something;</td>
</tr>
<tr>
<td>Coordinates: position;</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>
Example - Object

Domain 1

Car:

Attributes:
- Model
- Year
- Value
- Tax
- Debt

... Operations:
- calcTax:
- remind:
  ...

Domain 2

Car:

Attributes:
- Model
- Year
- Color
- Horsepower
- Maxspeed

... Operations:
- race:
- showoff:
- tune:
  ...

The set of attributes and operations is determined by the application - i.e. it is defined by the behavior during requirements definition and refined during requirements analysis.

Only those operations which determine the behavior are visible from the outside, all internal structure and functions are hidden.

Object - Orientation

A system is a collection of distinct objects having
- structure - data, state
- behavior - methods, functions

The seven magic characteristics of true o-o:
- Identity* name, handle, reference, capability
- Abstraction concentration on the essential
- Classification belongs to a certain category, instance
- Encapsulation information hiding, limited access
- Inheritance specialization / generalization
- Polymorphism generic behavior, overloading
- Persistence state and behavior transcend time, space
Objects – Class and Instance

Abstract Class (Type)

Class (Template)

Superclass

Instance

your_car: Car

attributes

methods

Subclass

Racecar

attributes

methods

my_car

attributes

methods

instantiates

Instance

my_car

attributes

methods

instantiates

your_car: Car

attributes

methods

Racecar

attributes

methods

Instance

your_car: Car

attributes

methods

Racecar

attributes

methods

my_car

attributes

methods

Instance

your_car: Car

attributes

methods

Racecar

attributes

methods

my_car

attributes

methods
**Object-Orientation**

- **A system is a set of interacting objects.**
  - material objects: car, plane, brake shoe, runway, ...
  - immaterial objects: government, management, flight_no, ...

- **Modeling:** mapping real objects (things in the real world) onto conceptual models in the solution space using abstraction = “extracting the essential aspects”

- **Implementing:** mapping the conceptual models onto software archetypes (modules, data structures, procedures)

---

**Relationships**

- **ABS**
  - sensor attributes
  - methods
  - control attributes
  - methods
  - increase
  - apply
  - composition
  - aggregation
  - association

- **Engine**
  - attributes
  - methods

- **Actuator**
  - attributes
  - methods

- **Brakepedal**
  - attributes
  - methods
Object-Orientation - 2

"object" means an entity which has two significant properties (object: class-type / instance-variable):
- attributes – a set of values describing the state
- operations – a set of functions which describe the behavior of the object – the methods

State changes are allowed only through methods! (information hiding): set-/get-attribute();

O-O programming languages must enforce these concepts by an appropriate encapsulation mechanism
- "get", "set" functions: instance.method(params);
- "private", "public", ("friend") keywords

Object-Orientation - 3

An object is a "black box" and acts as a server for other objects called clients

The client objects have to "know" the server and its interface – this combination of identity (internal name, "handle") and functionality is frequently called "capability"
- that is, the client must have the capability to call a server object – send it a "message" to perform a method
Classes and Instances

“Object” is used in two senses:

• as a “type”, i.e. a template for instances – Class
• as an instance of a Class – an object entity, “variable”

Example:
– Class Car; Instances: my_car, your_car, her_car, . . .

Diagram:

Polymorphism

Polymorphism allows to put the responsibility to the called object (callee) instead of the caller

• objext_x.print() instead of printX(object_x)

Sending a “stimulus” to an object to perform a certain operation

Using a unique symbol for the intended operation

• examples:
  – x + y ; k + j ; x/y ; k/j
  – bag.sort(); depending on what types of objects are contained in bag
Inheritance

- Basic idea: collect all common characteristics of a collection of very similar objects in one "parent" class
- If this parent class does not have any immediate instances it is considered an "abstract class"
- All specific objects are created from this class by generating child classes which inherit the common properties and add some specializations
- Subclassing allows the
  - addition of new attributes
  - redefinition ("overloading") of operations which they inherit from their ancestor classes
  - overriding (virtual function calls) redefines the characteristics of the ancestor class – hence, operations with the same name can have different semantics – attention!!!
- Must be supported by an inheritance mechanism of the programming language (genericity, templates)

Main Purposes for Inheritance

- Re-use of code;
  - use ready-made classes from library
  - combine similar parts of classes
- Subtyping – create behaviorally compatible descendants of a class
  - all should have the same interface
  - should only perform an extension, not a redefinition (overriding)
- Specialization
  - descendant is modified and no longer behaviorally identical to parent class
- Conceptual view
  - corresponds to the well-known “is-a” relation
Diagrams

Class Diagrams describe the objects of a system and the various kinds of static relationships
- associations - aggregation, interaction, ...
- subtypes – specialization, generalization

Interaction Diagrams describe how groups of objects collaborate in some behavior
- two forms are used:
  - sequence diagrams
  - collaboration diagrams

Diagrams are models!

Views and Diagrams
Models and Views

- A model is an abstraction of a system or a context
- A model can be documented in UML or some other form of notation/language
- An architectural view is an abstraction of a model,
  • taken from a specific perspective / view
  • enabling the extraction of architecturally essential elements
- Starting point is always the "user view" as defined by the use cases
- The different model views are:
  • the structural view - static relationships
  • the behavioral view - dynamic relationships
  • the implementation view – modules and components
  • the environmental view - deployment

Views and Diagrams

- Structural view:
  • concentrates on the static relationships
  • described by class and object diagrams

- Behavioral view:
  • concentrates on the dynamic relationships
  • described by one or several of the following
    – Sequence diagrams
    – Collaboration diagrams
    – State chart diagrams
    – Activity diagrams
Caveat: when writing the use cases do NOT concentrate on the functions but on the interactions!

Views and Diagrams - 2

- Starting point are the Use Cases and diagrams
- System development is a gradual transformation of a series of models described and documented by a set of diagrams
- Structural view:
  - class diagrams (generic templates, "types")
  - object diagrams – instances, "variables"
- Behavioral view:
  - sequence diagrams
  - collaboration diagrams
  - state chart diagrams
  - activity diagrams
- Implementation view:
  - component diagrams – modules
- Environment view:
  - deployment diagrams
Models

\- **The requirements model**
  • captures the functional requirements

\- **The analysis model**
  • describes a robust and changeable object structure for the system

\- **The design model**
  • Adopts and refines the object structure to the current implementation environment

\- **The implementation model**
  • is an implementation in a specific progr. Language

\- **The test model**
  • aims to validate and verify the system

Requirements Model

\- **Actors and Use Cases**
  • a use case model
  • interface description
  • a problem domain model

The use case model uses actors and use cases to define what exists outside the system and what should be performed by the system.
Object-Oriented Analysis

Contains the following activities, not necessarily in this order

- finding the objects - AM
- organizing the objects - AM
- describing how the objects interact - AM
- defining the objects internally – A/DM
Construction

*Construction = design + implementation*

*Main steps:*
  - identify the implementation environment
  - transform and refine the objects of the analysis model into the design model
  - describe how the objects interact for each use case

*Traceability - correspondence between the analysis and design models*

*Interactions - stimuli, events, signals and messages*

---

**Class-Diagrams**
Class Diagrams

Class diagrams and models can be looked at from three different perspectives:

- conceptual – represent the concepts in the domain without regard to the software, related to the classes that implement them but there is often no direct mapping

- specification – look at the software but concentrate on the interfaces, not the implementation – the “Type”; o-o languages usually combine interface and implementation
  (Modula 2/Ada: DEFINITION and IMPLEMENTATION
  MODULE/Package;
  Java: Class/Interface/Body)

- implementation – this is the actual implementation level frequently very close to or directly in the programming language used

Associations

Associations represent relationships between instances of classes

- conceptual – relationships between classes
  - each association has 2 ends – each can be labeled with a role name (adornment)
  - an association end has a multiplicity indicating how many instances may participate in this relationship (adornment)

- specification – relationships represent responsibilities
  - realized through one or more methods (get, update, . . .)
  - can not infer structure from the interface description

- implementation – here the details become visible
  - e.g. a doubly-linked pointer structure
  - a “ref” implementation (Java)
Vocabulary

- Classes are the most important building blocks
- A class represents a set of objects sharing the same properties (structure, behavior)
- A class implements one or more interfaces
- Classes may represent:
  - software things
  - hardware things
  - things that are purely conceptual
- Modeling involves identifying the things important to the particular view
- These things form the vocabulary of the system

Simple and Path Names

- Temperature Sensor
- Customer
- Car
- BusinessRules::FraudAgent
- java.awt::Rectangle

Things have names - identity
An attribute is a named property of a class that describes a range of values that instances may hold. Usually a short noun or noun phrase that represents some property.

**Things have state - attributes**

### Attributes - 1

<table>
<thead>
<tr>
<th>Customer</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
</tr>
<tr>
<td>address</td>
</tr>
<tr>
<td>phone</td>
</tr>
<tr>
<td>birthDate</td>
</tr>
<tr>
<td>. . .</td>
</tr>
</tbody>
</table>

### Attributes - 2

<table>
<thead>
<tr>
<th>Wall</th>
</tr>
</thead>
<tbody>
<tr>
<td>height: Float</td>
</tr>
<tr>
<td>width : Float</td>
</tr>
<tr>
<td>thickness : Float = 10.0</td>
</tr>
<tr>
<td>isLoadBearing: Boolean</td>
</tr>
<tr>
<td>. . .</td>
</tr>
</tbody>
</table>

An attribute can be further specified by giving its class (type) and possibly an initial value.
Operations - 1

<table>
<thead>
<tr>
<th>Rectangle</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>add()</td>
</tr>
<tr>
<td>grow()</td>
</tr>
<tr>
<td>move()</td>
</tr>
<tr>
<td>isEmpty()</td>
</tr>
<tr>
<td>. . .</td>
</tr>
</tbody>
</table>

An operation is the realization of a service that can be requested from any instance of the class; it is something one can do to an object. Usually a short verb or a verb phrase.

Things have behavior - operations

Operations - 2

<table>
<thead>
<tr>
<th>TemperatureSensor</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>reset()</td>
</tr>
<tr>
<td>setAlarm(t: Temperature )</td>
</tr>
<tr>
<td>value() : Temperature</td>
</tr>
<tr>
<td>. . .</td>
</tr>
</tbody>
</table>

An operation can be further specified by stating its signature
Attributes and Operations

- When drawing a class it is not necessary to show every attribute and every operation.
- In most cases there are too many and only the ones relevant to a specific view are included.
- For these reasons it is common to elide a class and show only some or none of the attributes and operations.
- An empty compartment does not mean that there are no attributes or operations.
- To better organize long lists of attributes and operations one can prefix each group with a descriptive category by using "stereotypes" <<guillemots>> e.g. <<registers>>
### Responsibilities

<table>
<thead>
<tr>
<th>FraudAgent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Responsibilities</td>
</tr>
<tr>
<td>-- determine the risk of a customer order</td>
</tr>
<tr>
<td>-- handle customer-specific criteria for fraud</td>
</tr>
</tbody>
</table>

A responsibility is a contract or an obligation of a class; responsibilities are derived from the UseCases. Usually given in free-form text, written as a phrase, a sentence or a short paragraph.

**Things have responsibilities**
- services

### Modeling Techniques

*Classes rarely stand alone; when building models one usually focuses on groups of classes that interact. These societies of classes form collaborations and are usually visualized in class diagrams.*

**Modeling the Vocabulary of a System**

- Identify those things that users or implementers use to describe the problem or solutions. UseCase-based analysis helps finding these abstractions
- For each abstraction identify the responsibilities. Make sure that each class is crisply defined and that there is a good balance of responsibilities among all classes
- Provide the attributes and operations that are needed to carry out the responsibilities for each class
Modeling the Vocabulary of a System

- As the models get larger, classes tend to cluster in groups that are conceptually and semantically related.

- In the UML packages can be used to model these clusters of classes.

- Most of the models will rarely be completely static but include interactions between the vocabulary.

- The UML provides a number of ways to model the dynamic behaviour.
Modeling the Distribution of Responsibilities in a System

- Identify a set of classes that work together closely to carry out some behavior
- Identify a set of responsibilities for each of the classes
- Split classes that have too many responsibilities into smaller abstractions
- Collapse tiny classes that have trivial responsibilities into larger ones
- Reallocate responsibilities so that each abstraction reasonably stands on its own
- Consider the ways in which those classes collaborate with one another and redistribute their responsibilities accordingly so that no class within a collaboration does too much or too little

Modeling the Vocabulary

<table>
<thead>
<tr>
<th>Model</th>
<th>View</th>
</tr>
</thead>
<tbody>
<tr>
<td>Responsibilities</td>
<td>Responsibilities</td>
</tr>
<tr>
<td>-- manage the state of the model</td>
<td>-- render the model on the screen</td>
</tr>
<tr>
<td></td>
<td>-- manage movement and resizing of the view</td>
</tr>
<tr>
<td></td>
<td>-- intercept user events</td>
</tr>
</tbody>
</table>

Controller

| Responsibilities |
| -- synchronize changes in the model and its views |
Modeling Primitive Types

- Primitive data types such as integers, characters, floating point numbers, strings and enumerations available in most programming languages can be modeled in UML.

- To model primitive types:
  - model the thing to be abstracted as a type or an enumeration which is rendered using class notation with the appropriate stereotype.
  - If the range of values associated with this type needs to be specified, make use of constraints.

<table>
<thead>
<tr>
<th>&lt;&lt;datatype&gt;&gt;</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>idle</td>
</tr>
<tr>
<td></td>
<td>working</td>
</tr>
<tr>
<td></td>
<td>error</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>&lt;&lt;enumeration&gt;&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status</td>
</tr>
<tr>
<td>idle</td>
</tr>
<tr>
<td>working</td>
</tr>
<tr>
<td>error</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>&lt;&lt;datatype&gt;&gt;</th>
<th>&lt;&lt;enum&gt;&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>Status</td>
</tr>
<tr>
<td></td>
<td>idle</td>
</tr>
<tr>
<td></td>
<td>working</td>
</tr>
<tr>
<td></td>
<td>error</td>
</tr>
<tr>
<td></td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>false</td>
</tr>
</tbody>
</table>

Hints and Tips

- Every class should map to some tangible or conceptual abstraction in the user or implementer domain. A well-structured class:
  - provides a crisp abstraction of something drawn from the vocabulary of the problem domain or the solution domain
  - embodies a small well defined set of responsibilities and carries them out very well
  - provides a clear separation of the abstractions specification and its implementation
  - is understandable and simple yet extensible and adaptable.
Hints and Tips - cont

When drawing a class in UML:

• show only those properties of the class that are important to understand the abstraction in its context;

• organize long lists of attributes and operations by grouping them according to their category;

• show related classes in the same class diagrams.