Softwaretechnik
Lecture 04: Object-Oriented Analysis

Peter Thiemann
University of Freiburg, Germany

SS 2012
Object-Oriented Analysis

- After introduction of OOP: need for OOA and OOD
- Purpose: Building OO models of software systems
- No generally accepted methodology; many different approaches: Booch, Rumbaugh (OMT), Coad/Yourdon, Jacobson (OOSE), Wirfs-Brock, ...
- Current approaches rely on UML (Unified Modeling Language, Booch/Jacobson/Rumbaugh)
- UML supports many kinds of semi-formal modeling techniques
  - use case diagrams
  - class diagrams
  - sequence diagrams
  - statechart diagrams
  - (activity diagrams)
  - (deployment diagrams)
The Concept “Model”
(according to Herbert Stachowiak, 1973)

Representation
A model is a representation of an original object.

Abstraction
A model need not encompass all features of the original object.

Pragmatism
A model is always goal-oriented.

- Modeling creates a representation that only encompasses the relevant features for a particular purpose.
Variations of Models

Informal models

- informal syntax, intuitive semantics
- ex: informal drawing on blackboard, colloquial description

Semi-formal models

- formally defined syntax (metamodel), intuitive semantics
- ex: many diagram types of UML

Formal models

- formally defined syntax and semantics
- ex: logical formulae, phrase structure grammars, programs
Class Diagram (UML)

- Data-oriented view, cf. ERD
- Representation of **classes** and their **static relationships**
- No information on dynamic behavior
- Notation is graph with
  - **nodes**: classes (rectangles)
  - **edges**: various relationships between classes
- May contain interfaces, packages, relationships, as well as instances (objects, links)
Classes

A class box has compartments for

- Class name
- Attributes (variables, fields)
- Operations (methods)

- only name compartment obligatory
- additional compartments may be defined
- class (static) attributes / operations underlined
Relations Between Classes

Binary Association

- indicates “collaboration” between two classes (possibly reflexive)
- solid line between two classes
- optional:
  - association name
  - decoration with role names
  - navigation (Design)
  - multiplicities (Design)

Generalization

- indicates subclass relation
- solid line with open arrow towards super class
Example Class Diagram

- **class**
  - class name
  - *name of abstract class* (optional)
  - attributes:
    - attribute1
    - attribute2: Typ = default
    - derived attribute
    - *class attribute*
  - methods:
    - *abstractOperation1()*
    - op2(parmList): result type
    - *class operation*

- **inheritance**
  - vehicle
  - flying object
    - car
    - airplane
    - bird

- **association**
  - Class 1
    - role: n
  - class 2
    - role: m

implementation of op2
Example Class Diagram

Diagram showing the relationships between `Part`, `product`, `manufacturer`, `Company`, `subpart`, `partno`, `order`, `orderer`, `0..1` relationship.
Aggregation and Composition

- Aggregation is a particular association **part-of**
- Notation: edge with rhombus as arrow head
- Composition is stronger form of aggregation
- Meaning: object “belongs existentially” to other object
- Object and its components live and die together
- Notation: edge with black rhombus as arrow head

Example

```
{ordered}  2..*  1
\[\begin{array}{c}
\text{point} \\
\text{representation} \\
\end{array}\]
```

```
\[\begin{array}{c}
\text{car} \\
\text{wheel} \\
\end{array}\]
```
Ten Steps Towards an OOA Model

Heide Balzert

1. Data analysis: identify classes
2. Identify associations and compositions
3. Identify attributes and operations for each class
4. Construct object life cycle
5. Introduce inheritance
6. Identify internal operations
7. Specify operations
8. Check inheritance
9. Check associations and compositions
10. Decompose in subsystems
Step: Identify Classes

- identify tangible entities: physical objects (airplane), roles (manager), events (request, form), interactions (meeting), locations (office), organizational units (company)
- top-down: scan verbal requirements
  - nouns → objects, attributes
  - verbs → operations
- bottom-up:
  - collect attributes (data) and operations
  - combine into classes
- name of class: concrete noun, singular, describes all objects (no roles)
- classes related via invariable 1:1 associations may be joined
Step: Identify Associations and Compositions

- permanent relations between objects
- scan verbal requirements for verbs
- technical subsidiarity: composition
- communication between objects $\rightarrow$ association
- determine roles
- snapshot / history required?
- constraints?
- are there attributes / operations for association?
- determine cardinalities
Attributes and Operations by Form Analysis

Upload new Good

Name
Picture
Description
Category
Auction off?

Good

name
picture
description
category
status

display()
edit()
...
Step: Identify Attributes and Operations

CRC Cards (Wirfs-Brock)

- CRC = Class-Responsibility-Collaboration
- initially, a class is assigned responsibilities and collaborators
- collaborator is a class cooperating to fulfill responsibilities
- three-four responsibilities per card (class); otherwise: split class
- developed iteratively through series of meetings
Example CRC Card

- **class name**
- **order**
  - check if on stock
  - determine price
  - check payment
  - ship product
  - item
  - item
  - customer

**responsibilities** | **collaborators**
Use Case: buy product

- Locate product in catalogue
- Browse features of product
- Place product in shopping cart
- Proceed to checkout
- Enter payment info
- Enter shipping info
- Confirm sale
Notation for Designing Datatypes (F#)

type sale = { cart: shoppingCart;
              shipment: shipmentInfo;
              payment: paymentInfo }
and shoppingCart = { contents: product list }
and shipmentInfo = { name: string;
                     address: string }
and paymentInfo = { accountNr: string;
                   bankingCode: string }
and product = { name: string;
               price: int;
               features: feature list }
and feature = { name: string }

▶ Named record types
Classes from Requirements

A graphics program should draw different geometric shapes in a coordinate system. There are four kinds of shapes:

- Rectangles given by upper left corner, width, and height
- Disks given by center point and radius
- Points
- Overlays composed of two shapes
Classes from Requirements

type cartPt = { x: int; y: int }
and shape =
  Rectangle of rectangle
  | Disk of disk
  | Point of point
  | Overlay of overlay
and rectangle = { loc: cartPt; width: int; height: int }
and disk = { loc: cartPt; radius: int }
and point = { loc: cartPt }
and overlay = { lower: shape; upper: shape }

▶ Sum type (shape) for alternatives
Mapping from F# Types to Class Diagrams

Mapping a type definition

\[
\{\text{type } tdef_1 \text{ and } \ldots \text{ and } tdef_n \} = [tdef_1] \cup \cdots \cup [tdef_n]
\]

Mapping a record type

\[
\left[ \text{name} = \{ x_i : t_i, y_j : t_{n_j} \mid c_j \} \right] =
\]

\[
\left[ \text{list} \right] = *
\]

\[
\left[ \text{option} \right] = 0, 1
\]

\[
\left[ \text{} \right] = 1
\]

Mapping a sum type

\[
\left[ \text{name} = T_1 \mid \ldots \mid T_n \right] =
\]

Peter Thiemann (Univ. Freiburg) Softwaretechnik SWT 21 / 41
Applied to Example Code

Class Diagram

```
Shape

Point
  width : int
  height : int

Rectangle

Disk
  radius : int

Overlay

CartPt
  x : int
  y : int
```
... Operations

A graphics program should draw different geometric shapes...

- Each class should have a draw() operation
- Shape should also have draw() operation
- Discovered the “Composite Pattern”!
Example Code with Draw Method
Step: Construct Object Life Cycle

Object Life Cycle

- Object creation
- Initialization
- . . .
- Finalization
- Object destruction

Life Cycle — Type State

- operations can only be executed in particular state
- idea: incoming message (in class diagram) \( \models \) event (in a statechart diagram) that triggers the operation
Example: Java Iterator — Statechart Diagram

interface Iterator<E> {
   /** Returns true if the iteration has more elements. */
   public boolean hasNext();
   /** Returns the next element in the iteration. */
   public E next();
   /** Removes from the underlying collection the last element
    * returned by the iterator (optional operation). */
   public void remove();
}

![Statechart Diagram]

Peter Thiemann (Univ. Freiburg) Softwaretechnik SWT 26 / 41
Statechart Diagram

- Modeling the evolving state of an object
- Based on deterministic finite automaton (FSA)

$$A = (Q, \Sigma, \delta, q_0, F)$$ where

- \( Q \): finite set of states
- \( \Sigma \): finite input alphabet
- \( \delta \): \( Q \times \Sigma \rightarrow Q \) transition function
- \( q_0 \in Q \) initial state
- \( F \subseteq Q \) set of final states
Graphical Representation of FSA

- **nodes:** states of the automaton (*circles* or *rectangles*)
- **arrow pointing to** \( q_0 \)
- **final states indicated by double circle**
- **edges:** if \( \delta(q, a) = q' \) then **transition** labeled \( a \) from \( q \) to \( q' \)

**FSA with output** specifies a translation \( \Sigma^* \rightarrow \Delta^* \)

- \( M = (Q, \Sigma, \Delta, \delta, \lambda, q_0) \)
- replace final states \( F \) by output alphabet \( \Delta \) and output function \( \lambda \)
- **Mealy-automaton:** \( \lambda : Q \times \Sigma \rightarrow \Delta \)
  edge from \( q \) to \( \delta(q, a) \) additionally carries \( \lambda(q, a) \)
- **Moore-automaton:** \( \lambda : Q \rightarrow \Delta \)
  state \( q \) labeled with \( \lambda(q) \)
Example: Digital Clock as a Mealy-automaton

Drawback: FSAs get big too quickly → structuring required
Statechart Diagram (Harel, UML)

- hybrid automata ("Moore + Mealy")
- each state may have
  - **entry action**: executed on entry to state
    - ≅ labeling all incoming edges
  - **exit action**: executed on exit of state
    - ≅ labeling all outgoing edges
  - **do activity**:
    - executed while in state
- composite states
- states with history
- concurrent states
- optional: conditional state transitions
Example: Statechart Diagram

- **State 1**: Event 1/Action 1
- **State 2**: Exit/Action 4
- **State 3**: Entry/Action 3, Exit/Action 4, Do/Activity 4, Include/Submachine Invocation
- **State 4**: Event 4

- **Idle**: Ticket inserted/
- **Waiting for Coin**: Coin inserted (enough)/
- **Coin Inserted (not enough)**: Display remaining amount
- **Coin Inserted (enough)**: Print card
- **Timeout/Release Coins**:
Composite States

- states can be grouped into a composite state with designated start node (→ hierarchy)
- edges may start and end at any level
- transition from a composite state \( \cong \) set of transitions with identical labels from all members of the composite state
- transition to a composite state leads to its initial state
- transitions may be “stubbed”
States with History

- composite state with history — marked (H) — remembers the internal state on exit and resumes in that internal state on the next entry

- the history state indicator may be target of transitions from the outside and it may indicate a default “previous state”

- “deep history” (H*) remembers nested state
Concurrent States

- composite state may contain **concurrent state regions** (separated by dashed lines)
- all components execute concurrently
- transitions may depend on state of another component (synchronisation)
- explicit synchronization points
- concurrent transitions

sequence of states on input abcb:
(A, C), (B, D), (B, D), (B, C), (A, C)
Alternative: Sequence Diagram

- description of the sequence of messages
- → communications protocols

```
response1
message2()
obj3: Class3
time obj1: Class1
Class2()
message1()
obj2: Class2
recursive
message
message3()
```
Alternative: Object and Collaboration Diagrams (UML)

- notation for **objects** and their **links**
- UML notation:
  - **nodes**: objects (rectangles), labeled with **object name:type**
  - **edges**: links between objects
    “objects that know each other”

Properties of object diagrams

- snapshot of a system state
- configuration of a specific group of objects
Example: Object Diagram

anObject: class

: class2

: class3

anotherObject

attribute1 = value1
attribute2 = value2
Dynamic properties $\rightarrow$ collaboration diagrams

- objects $\rightarrow$ **object roles**
- object notation stands for “any object of that class”
- object roles and links may be labeled with constraints
  - $\{\text{new}\}$
  - $\{\text{transient}\}$
  - $\{\text{destroyed}\}$
- labeling links with numbered operations
- numbering implies sequence of execution
Step: Introduce Inheritance

- Use sparingly!
- Use inheritance for abstracting common patterns:
  Collect common attributes and operations in abstract superclass
- Alternative: collect in separate class and use composition
Step: Specify Operations

- Data-driven development: [Jackson]
  Derive structure of operation from data it operates on
- Test-driven development: [Beck]
  Specify a set of meaningful test cases
- Design by contract: [Meyer]
  - Define class invariants
  - Specify operations by pre- and postconditions
- Pseudocode Programming Process (PPP): [McConnell]
  - Start with high-level pseudocode
  - Refine pseudocode until implementation obvious