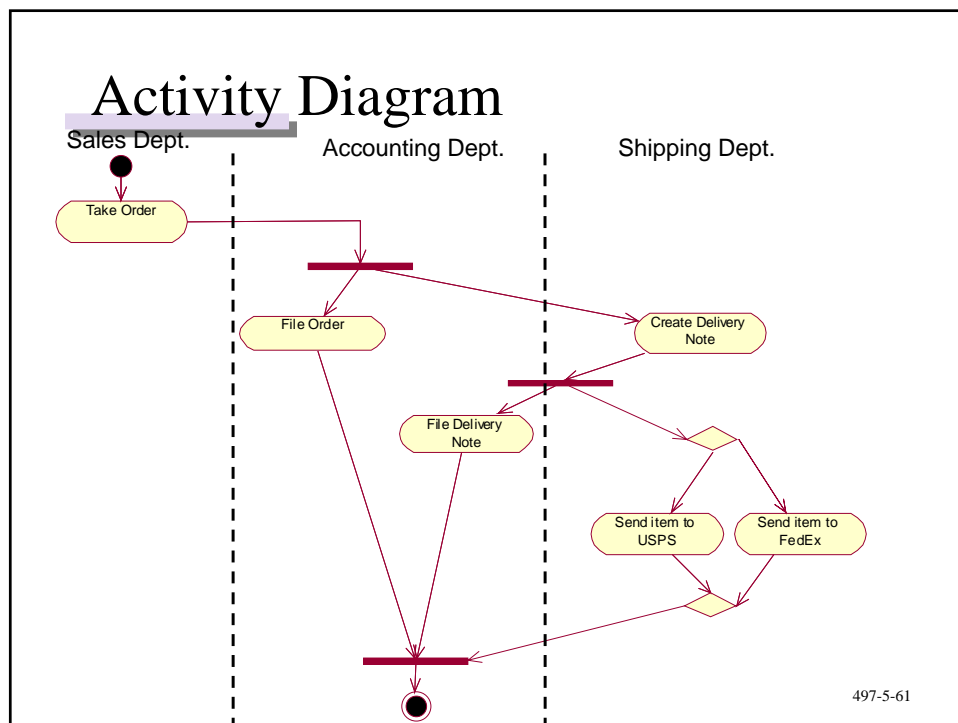


4.5 Activity Diagrams

- Activity diagrams show sequence of activities
- Good for modeling workflows in analysis
- Similar to ANSI flow charts
 - outdated

497-5-60



Elements of Activity Diagrams

- Elements known from state diagrams:
 - activity states
 - transitions
 - start state
 - end state
- New elements
 - fork (horizontal bars)
 - A process is split into two (or more) concurrent processes
 - join (horizontal bars)
 - Several concurrent processes are joined into one
 - branch / merge (diamonds)
 - like if-statements: “then” branch and “else” branch
 - partitions
 - also known as: swimlanes
 - to classify activities

497-5-62

Use of Activity Diagrams

- Analyzing workflows
 - well-suited for parallel processes, workflows
 - similar use as dataflow diagrams
 - difference: focus on activities, not data
- Modeling control flow in methods (design phase)
 - algorithms
 - activities are mapped to statements
- Similar to:
 - flowcharts / control-flow diagrams
 - Petri nets
- Disadvantage: not very object-oriented
 - do not show classes, objects, operations
 - interaction diagrams are usually a better choice

497-5-63

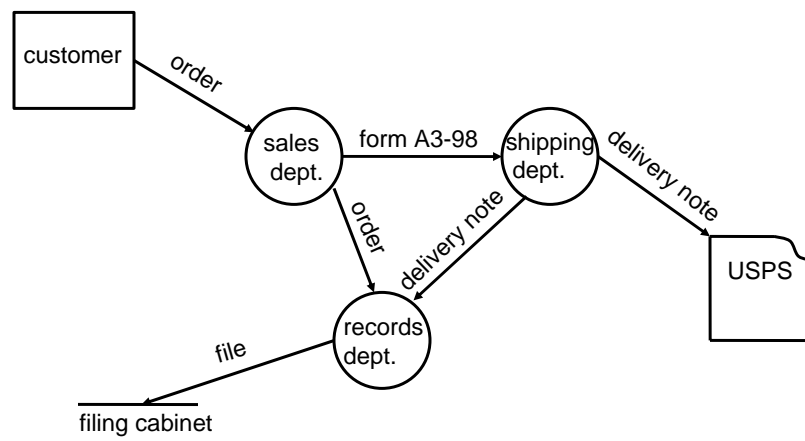
Non-UML Specification Languages

- Dataflow Diagrams
- Entity-Relationship Diagram

- Not part of the UML
- Still used though

497-5-64

Dataflow Diagram

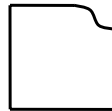


497-5-65

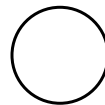
Elements of Dataflow Diagrams



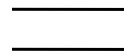
data source



data sink



activity



data store



data flow

497-5-66

Dataflow Diagrams

- Focus on data
 - as opposed to control flow diagrams (like Activity diagrams)
 - no notion of time or sequence
- Good for modeling work in real-life organizations
 - activities are inherently parallel
 - data: forms or papers
 - activities: people or organizations
- No direct equivalent in UML

497-5-67

Assertions and the Object Constraint Language

- Assertions: constraints of classes
 - Preconditions: must be true before a method is executed
 - Postconditions: must be true after a method is executed
 - Invariants: must be true both before and after all method executions in a class
 - “Design by Contract”
- Object Constraint Language (OCL)
 - A formal, textual notation for constraints in UML models
 - Optional part of UML
 - Often used to express assertions
 - Predicate logic, sets
 - Expressions without side-effects

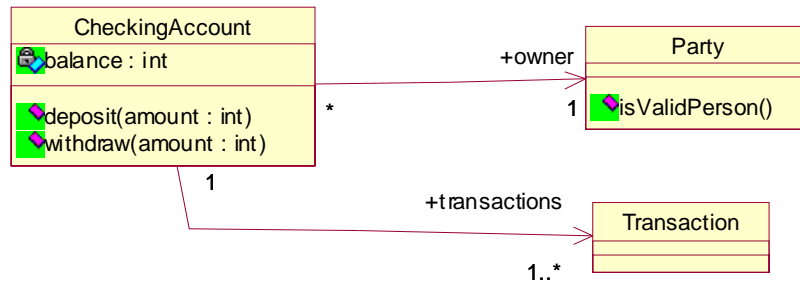
497-5-68

Design by Contract

- Contract between a method and its caller
 - If you fulfill my precondition,
I promise to fulfill my postcondition
 - using pre- and postconditions to specify a method
 - partial or complete specification
- Also useful for
 - reusing a library that someone else wrote
 - making sure it does what you think it does

497-5-69

Assertion Example: Checking Account



497-5-70

OCL Example

```
context CheckingAccount:: deposit( amount: Integer )
  pre: amount > 0
  post: balance = balance@pre + amount
context CheckingAccount:: withdraw( amount: Integer )
  pre: amount > 0 and ( amount <= balance )
  post: balance = balance@pre - amount
context CheckingAccount
  inv: balance >= 0
  inv: owner. isValidPerson()
  inv: ( transactions -> size() ) >= 1
```

- Syntax Elements: context declaration, types, expressions, attributes and operations, "@pre" postfix

497-5-71

OCL

- Context is necessary only when constraint is listed outside of a class diagram
- Dot operator (".") for accessing attributes and operations of objects
 - also: objects connected through associations
 - "self": name of the current object
`self.balance >= 0`
- Arrow operator ("->") for accessing operations of collections
 - the result of associations with multiplicity > 1 is always a set
 - collection operations can be used on objects too: each object is a set with one element (itself)
`owner -> size() = 1`

497-5-72

Some Predefined OCL Types

- Boolean
 - and, or, not, xor
- Integer
 - +, *, -, /, abs()
- Real
 - +, *, -, /, floor()
- String
 - toUpper(), concat()
- Collection
 - isEmpty(): Boolean, size(): Integer, forAll(boolean-expression): Boolean

497-5-73

Assertion Example: Stack

- Stack with a maximum size
 - pop(): Object
 - push(Object)
 - empty(): boolean
 - attributes: top, maxSize, elements
- Preconditions and postconditions?
 - for pop?
 - for push?
 - for empty?
- Class invariants?

497-5-74

Stack in Java

```
class Stack< E > {
    Stack( int maxSize );

    /** Add element to your top. */
    void push( E element );

    /** Remove and return the element at your top. */
    E pop();

    /** Return whether you are empty. */
    boolean isEmpty();

    private E[] elements;
    private int top; //index of the highest element
}
```

Preconditions? Postconditions? Invariants?

Stack Assertions

```
context Stack
  inv: (top >= 0) and (top <= elements. length)
context Stack:: pop(): E
  pre: top > 0
  post: top = top@pre - 1
context Stack:: push( element: E )
  pre: top < maxSize
  post: top = top@pre + 1
context Stack:: empty(): Boolean
  post: result = ( top = 0 )
```

- Is this specification complete?
- Is there implementation bias?

497-5-76

Pre- and Postconditions

- Can (in some cases) specify a method completely
 - CheckingAccount example
 - pre- and postconditions are a complete formal specification of these methods
 - theoretically, an implementation can automatically be generated
 - compare: logical programming (Prolog)
- Generally: incomplete specification
 - Stack example
 - property that pop() returns the parameter of the last non-popped push(): hard to specify in OCL!
 - cause: no assignment possible
 - can be specified if referring to underlying implementation (e.g. array): but that's not good (implementation bias)

497-5-77

Assertions in Design and Implementation

- Used in to design to prepare implementation
 - implementation needs to adhere to constraints
- Some assertions may be replaced by exceptions
 - throw an exception if a precondition is not met
- Programming language support for assertions
 - Eiffel
 - C++: `assert(int)`
 - in `assert.h`
 - Java 1.4
 - “assert” keyword
 - throws `AssertionError` if parameter is false
 - has to be enabled with a command line parameter (`-ea`)

```
double y = squareRoot( x );
assert( Math. abs( x - ( y * y ) ) < 0.001 ) :
    "squareRoot too imprecise";
```
 - An “`assert()`” function can easily be implemented in any 497-5-79 language