Software Fault Tolerance
Programming Language Features

Jörg Kienzle
Software Engineering Laboratory
School of Computer Science
McGill University

Overview

(Kienzle Chapter 1 & 9)

- Ada
- Object-Orientation
- Reuse
- Managing State (Access, Copying & Serialization)
- Concurrency
- Exceptions
- Reflection
- Aspect-Orientation

History of Ada

- Military programming language (DoD)
  - Reason: Too many programming languages
- Late 70's
- Ada 83 (ISO standard)
  - Object-based
- DoD: Ada mandate
- Ada 95 (ISO standard)
  - Object-oriented
- DoD: "Ada recommended"
- Ada 2005 (ISO standard)

Augusta Ada Lovelace
(1815 - 1852)

Ada Philosophy

- Reliability
- Maintainability
- Programming as a human activity
- Efficiency
- Portability
- Standardization and validation

Ada Fundamentals

- Imperative programming language inspired by Pascal
- Strong typing
- Modularity and information hiding
- Separation of interface and implementation
  - Separate compilation
- Exceptions
- Object-orientation
- Templates (called generic packages)
- Concurrency
- Real-time

Ada Structure

- Core language
  - Fundamentals
  - Object-orientation
  - Hierarchical library units (packages)
  - Threads (tasks) and data-oriented synchronization (protected objects)
- Specialized annexes
  - Systems programming
  - Real-time, Numerics
  - Distributed systems
- Language interfaces (to C, Cobol, Fortran, …)

Additional Info on Ada

- Reference Manuals
  - Ada 2005:
    `http://www.adaic.org/standards/05rm/html/RM-TTL.html`
  - Taft, T. S.; Duff, R. A.; Brukardt, R. L.; Ploedereder, E.: "Consolidated Ada Reference Manual", ISO/IEC Standard 8652/1995, Lecture Notes in Computer Science 2219, Springer, 2000. ISBN 3-540-43038-5
- Used in Avionics (Boeing, Airbus), Banking, TGV, European Space Agency
- Free Ada Compiler: GNAT (GNU Ada Translator)
  - libre.adacore.com

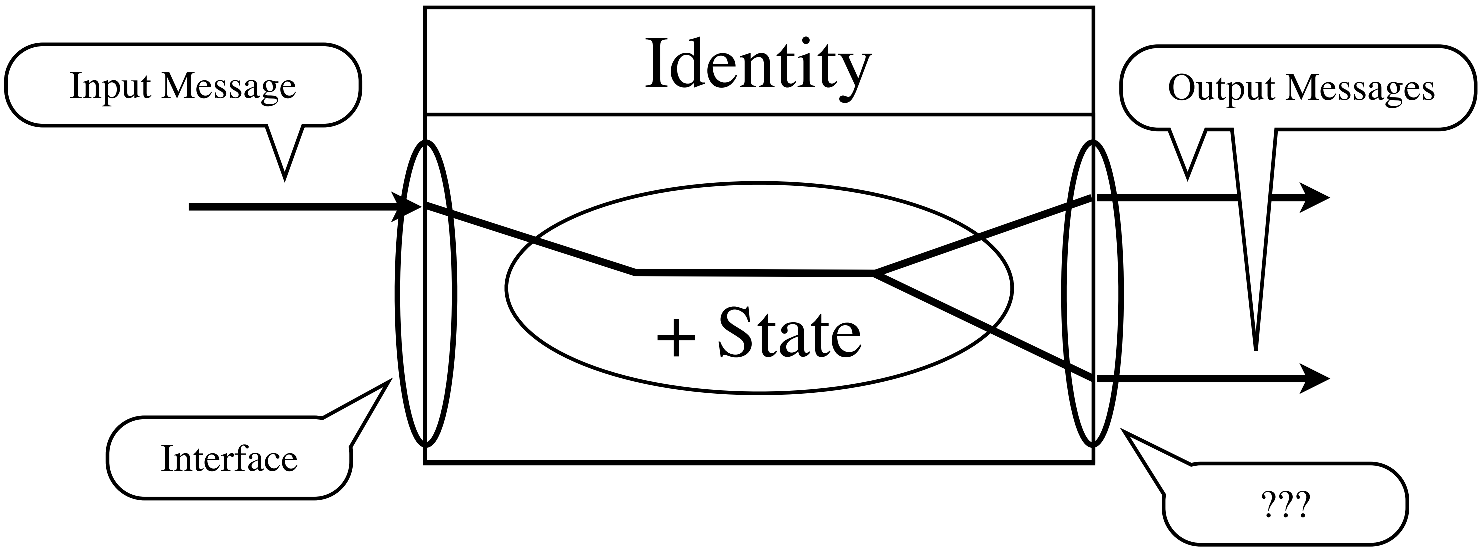
Object-Orientation (1)

- Built on old principles
- Abstraction
  - Extraction of essential properties while omitting inessential details
- Encapsulation and Information Hiding
  - Separation of the external view from the internal details
  - Aspects that should not affect / are not relevant to other parts of the system are made inaccessible

Object-Orientation (2)

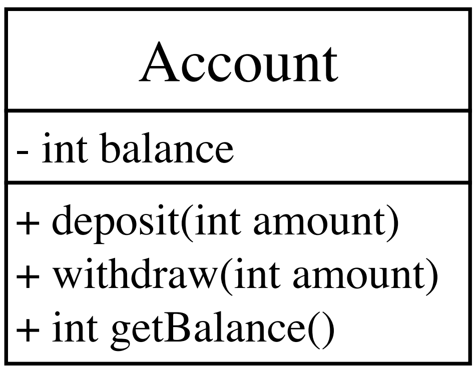
- Modularity
  - Decomposition into a set of cohesive and loosely coupled units
- Classification
  - Ability to group objects according to common properties
  - Abilities for an object to belong to more than one classification

An Object



+ Behavior

A Class: UML and Java Views




```java
class Account {
 public Account() {};

 private int balance = 0;

 public void deposit(int amount) {
  balance += amount;
 }

 public void withdraw(int amount) {
  balance -= amount;
 }

 public int getBalance() {
  return balance;
 }
}
```

Ada Packages

- Pack, isolate and encapsulate resources
- The specification defines the interface to the outside
- The body contains the implementation
- Two separate files

```
package Accounts is
 …
[private]
 …
end Accounts;
```

```
package body Accounts is
 …
end Accounts;
```

Ada Tagged Types

- Specification

```
type Account is tagged private;

procedure Withdraw(A : in out Account;
                   Amount : in Integer);
```

- Body

```
type Account is tagged record
 Balance : Integer;
end record;

procedure Withdraw (A : in out Account;
                    Amount : in Integer) is
begin
 A.Balance := A.Balance - Amount;
end Withdraw;
```

Ada O-O Summary

Class

=

Tagged Type Definition

\+ Operations using the Type as Parameter

\+ Package

Account Class in Ada

```
package Accounts is
  type Account is tagged private;
  procedure Deposit
   (A : in out Account;
   Amount : in Positive);
  procedure Withdraw
   (A : in out Account;
   Amount : in Positive);
  function Get_Balance
   (A : in Account)
  return Natural;
private
  type Account is tagged record
    Balance : Natural;
  end record;
end Accounts;
```

Interface (Visible to the Outside)

Visible to child packages

Visible to no one

```
package body Accounts is
  procedure Deposit
  (A : in out Account;
   Amount : in Positive) is
  begin
  A.Balance :=
   A.Balance + Amount;
  end Deposit;
  procedure Withdraw
  (A : in out Account;
   Amount : in Positive) is
  begin
  A.Balance :=
   A.Balance - Amount;
  end Withdraw;
  function Get_Balance
  (A : in Account) return Natural is
  begin
   return A.Balance;
  end Get_Balance;
end Accounts;
```

Inheritance

Import / Include

Child package

```
with Canvases; use Canvases;
package Shapes is

  type Shape is abstract
   tagged null record;

  procedure Draw
   (S        : in Shape;
    Canvas : access
               Canvas'Class)
   is abstract;

end Shapes;
```

Abstract class and method

```
package Shapes.Circles is

  type Circle is new Shape with
   record
    -- Added fields
    Center : Point;
    Radius : Float;
   end record;

  procedure Draw
   (C        : in Circle;
    Canvas : access Canvas'Class);

  function Radius
   (C : in Circle) return Float;

end Shapes.Circles;
```

Subclass

Constructors / Destructors

- There are no default constructors / destructors in Ada
- If you need such a functionality, you can inherit from a special class `Controlled`

```
package Ada.Finalization is

  type Controlled is abstract tagged private;
  procedure Initialize (Object : in out Controlled);
  procedure Adjust (Object : in out Controlled);
  procedure Finalize (Object : in out Controlled);

  type Limited_Controlled is abstract tagged
     limited private;
  -- Initialize and Finalize
end Ada.Finalization;
```

Generics / Templates

- Generics (or templates as they are called in C++) provide the possibility of parameterizing code with types
  - Algorithms can be written in terms of to-be-specified-later types
  - Pioneered by Ada 83
- Generics can facilitate code reuse
- Generics provide type safety
- Typical examples: generic container classes
  - List
  - Stack

Reuse with Templates (1)

```
generic
 type Element is private;
package Stacks is
 type Stack_Type (Max : Natural) is private;
 procedure Push
    (Stack: in out Stack_Type;
     X: in Element);
 procedure Pop
    (Stack: in out Stack_Type;
     X: out Element);
 Stack_Overflow : exception;
private
 type Table_Type is
  array (Natural range <>) of Element;
 type Stack_Type (Max : Natural) is record
  Table: Table_Type (1 .. Max);
  Top: Natural range 0 .. Max := 0;
 end record;
end Stacks;
```

Generic package specification of a Stack ADT

Discriminant

Generic formal parameter

Unconstrained Array

Reuse with Templates (2)

```
with Stacks;
package Character_Stacks is
 new Stacks (Character);

declare
 S : Character_Stacks.Stack_Type (20);
begin
  Character_Stacks.Push (S, 'J');
end;
```

Generic package instantiation and use

Controlled Copying

- C++
  - Override the assignment operator
  - Provide a “copy constructor”
- Java
  - On assignment only object references are copied
  - Interface `Cloneable`
  - Implement `clone()`
    - `CloneNotSupportedException` thrown if not implemented

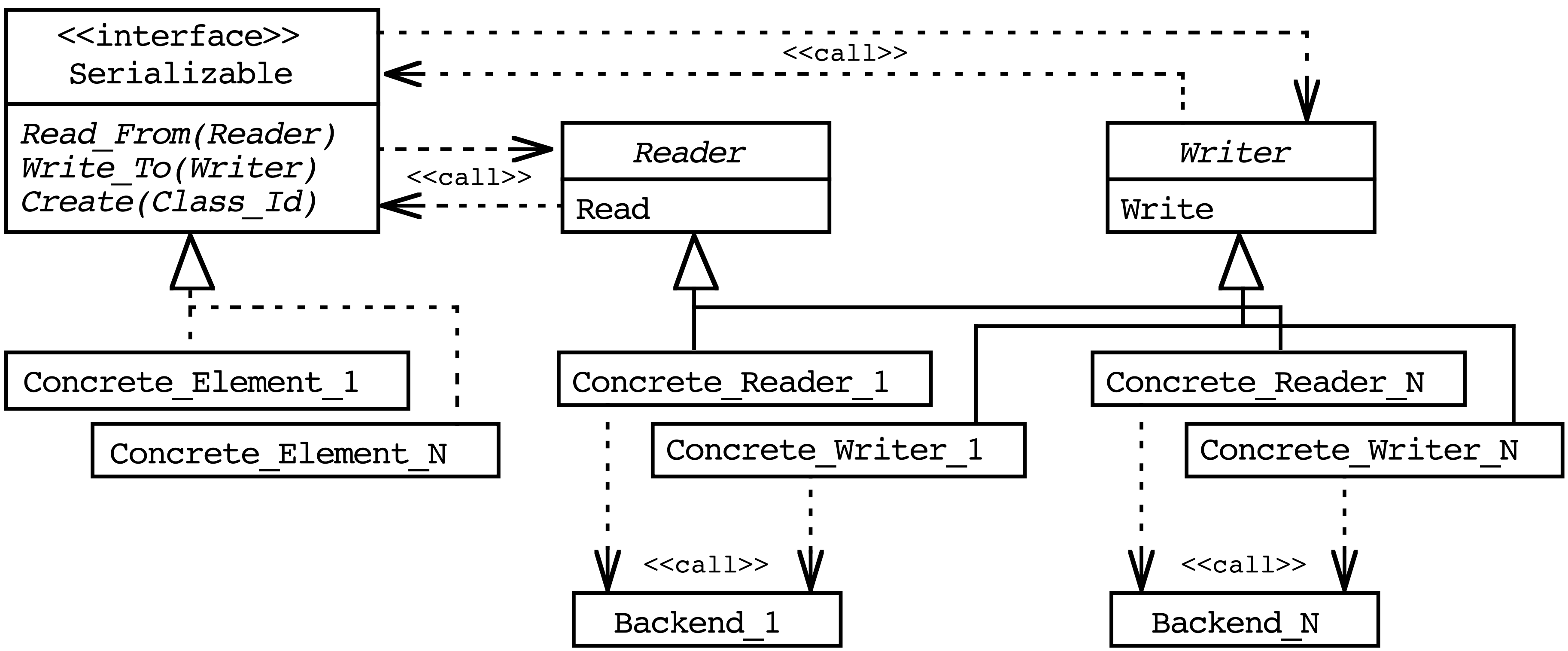
Limited Types in Ada

- Limited types are types for which neither assignment nor (non)equality are implicitly predefined
  - Private types declared with the keyword limited
  - A type derived from a limited type
  - A composite type having a limited component
  - Task types and protected types
- The equality operator can be explicitly declared; non-equality is then implicitly defined
- Assignment cannot be explicitly declared

Serialization (1)

- Serializer design pattern [RSB+98]
- Read arbitrarily complex object structures from, and write them to varying data structure backends
- Participants
  - Reader / Writer
    - Declare read / write operations for every value type
  - Concrete Reader / Concrete Writer
    - Implement the operations for a particular backend / external representation format
  - Serializable interface
  - Backend

Serialization (2)



Ada Streams

- Abstract tagged `Root_Stream_Type`
- Defines abstract operations for reading and writing arrays of bytes

```
procedure Write (Stream : in out Root_Stream_Type;
                 Item : in Stream_Element_Array);
```

- Predefined attributes `'Read` / `'Write` and `'Input` / `'Output` for any non-limited type
- It is possible to redefine the default implementations or provide implementations for limited types

Example: Stream_IO

```
with Ada.Streams.Stream_IO;
use Ada.Streams.Stream_IO;
declare
 My_File : File_Type;
 S : Stream_Access;
 I : Integer; My_String : String := “Hello”;
 T : A_Tagged_Type’Class := … ;
begin
 Create (My_File, “file_name”);
 S := Stream (My_File);
 --  do some work
 Integer’Write (S, I);
 String’Output (S, My_String);
 A_Tagged_Type’Class’Output (S, T);
 Close (My_File);
end;
```

Reasons for Concurrency

- Distributed Systems
  - Active components
  - Communication and synchronization
- Centralized systems
  - Systems that handle sporadic incoming events
  - "Concurrent nature" of the problem
  - Using concurrency in the application is intuitive and simple
  - Performance reasons (multiprocessor systems)

Classification of Concurrent Systems

- Independent
  - No communication
- Competing
  - Designed separately
  - Compete for resources
- Cooperating
  - Perform some job together
  - Use each others help and results
  - Communicate directly or through shared resources

Concurrency and Object-Orientation

- Are object-oriented systems “naturally” concurrent?
- In my view:
  - Concurrency adds a “new dimension” to systems
  - In sequential OO systems, only one method executes at a given time
  - In concurrent OO systems, multiple methods execute concurrently
  - One method executes multiple times concurrently

Orthogonal and Integrated Languages

- Classification based on the relationship between objects and processes [BGL98]
- Orthogonal languages
  - Processes are special entities different from objects
- Integrated languages
  - Active objects encapsulate one or more processes
  - Homogenous: only active objects (Ada83)
  - Inhomogenous: active and passive objects (Ada95)

Ada Concurrency by Example

- The three of us want to have dinner together.
- We have to go out and buy meat, bread and wine.
- To speed up things, we split and go shopping concurrently.

Ada Tasks (1)

```
type Food_Type is
  (Meat, Bread, Wine);

package Shoppers is

  task type Shopper_Task is
    entry Start
      (Item: in Food_Type;
       Quantity: in Positive);
  end Shopper_Task;

end Shoppers;
```

Active Object Specification

Callable “Method”

```
package body Shoppers is
  task body Shopper_Task is
    Produce: Food_Type;
    Needs: Positive;
  begin
    accept Start
      (Item: in Food_Type;
       Quantity: in Positive) do
    -- performed under mutual exclusion!
     Produce := Item;
     Needs := Quantity;
    end Start;
    -- go shopping
    delay some_random_value;
  end Shopper_Task;
end Shoppers;
```

Body executes as soon as instance is declared

Sleep for a specified duration

Ada Tasks (2)

Task instance declaration

Entry calls

```
with Shoppers;
procedure Go_Shopping is
  Meat_Shopper, Bread_Shopper,
   Wine_Shopper: Shoppers.Shopper_Task;
begin
  -- All task objects are activated and run until
  -- they encounter the accept statement.
  -- Now they wait until the entry is called.
  Meat_Shopper.Start (Meat, 2);
  Bread_Shopper.Start (Bread, 1);
  Wine_Shopper.Start (Wine, 3);
  -- all tasks are now running concurrently
end Go_Shopping;
-- the main procedure terminates when all dependent
-- tasks have terminated
```

Ada Rendezvous

- The main procedure is a client of the `Shopper` tasks (the servers).
- The client tasks get a service from the server task by calling one of its entries.
- The server task accepts entry calls by executing `accept` statements.
- If the client issues the entry call before the server is ready to service (accept) it, the client task is suspended and put into a waiting queue.
- If the server task arrives at an `accept` statement and there is no waiting client, then the server task is suspended and waits until a client calls the entry.
- In both cases, the client and the server finally "meet"

Advanced Rendezvous

- Associate conditions with a rendezvous
- Wait for more then a single rendezvous at a time
- Time-out if no rendezvous is forthcoming within a specific period
- Withdraw an offer to rendezvous if no rendezvous is immediately available
- Terminate if no clients can possibly call a task's entries

Thirsty Shoppers

- If someone offers coffee, accept, but only once!

```
package body Shoppers is
 task body Shopper_Task is
  Produce: Food_Type; Needs: Positive;
  Thirsty: Boolean := True;
 begin
  loop
   select
    accept Start (Item: in Food_Type;
      Quantity: in Positive) do
     Produce := Item; Needs := Quantity;
    end Start;
    exit;
   or
    when Thirsty =>
     accept Coffee_Is_Ready;
    Thirsty := False; --  Drink Coffee
   end select;
  end loop;
  -- go shopping
 end Shopper_Task;
end Shoppers;
```

Selective Accept

Conditional Accept

Shared Resources

- When concurrent processes simultaneously access a resource, interference might result, e.g. the state of the resource might become corrupted.
- Rules for sharing a resource
  - Multiple concurrent readers
    (if there is no writer).
  - A single writer
    (no other readers or writers).

Account Class not Thread Safe

```
package Accounts is
   type Account is tagged private;
   procedure Deposit
    (A : in out Account;
    Amount : in Positive);
   procedure Withdraw
    (A : in out Account;
    Amount : in Positive);
   function Get_Balance
    (A : in Account)
   return Natural;
private
   type Account is tagged record
     Balance : Natural;
   end record;
end Accounts;
```

```
package body Accounts is
 procedure Deposit
 (A : in out Account;
  Amount : in Positive) is
 begin
 A.Balance :=
  A.Balance + Amount;
 end Deposit;
 procedure Withdraw
 (A : in out Account;
  Amount : in Positive) is
 begin
 A.Balance :=
  A.Balance - Amount;
 end Withdraw;
 function Get_Balance
 (A : in Account) return Natural is
 begin
  return A.Balance;
 end Get_Balance;
end Accounts;
```

Statements not Atomic

Ada Protected Type

```
package PAccounts is

 protected type PAccount is
  procedure Deposit
   (Amount: in Positive);
  -- Note that there is an implicit
  -- PAccount parameter!
  procedure Withdraw
   (Amount: in Positive);
  function Get_Balance
   return Natural;
 private
  Balance: Natural := 0;
 end PAccount;

end PAccounts;
```

Fields are always private

Multiple reader (functions) / single writer (procedures)

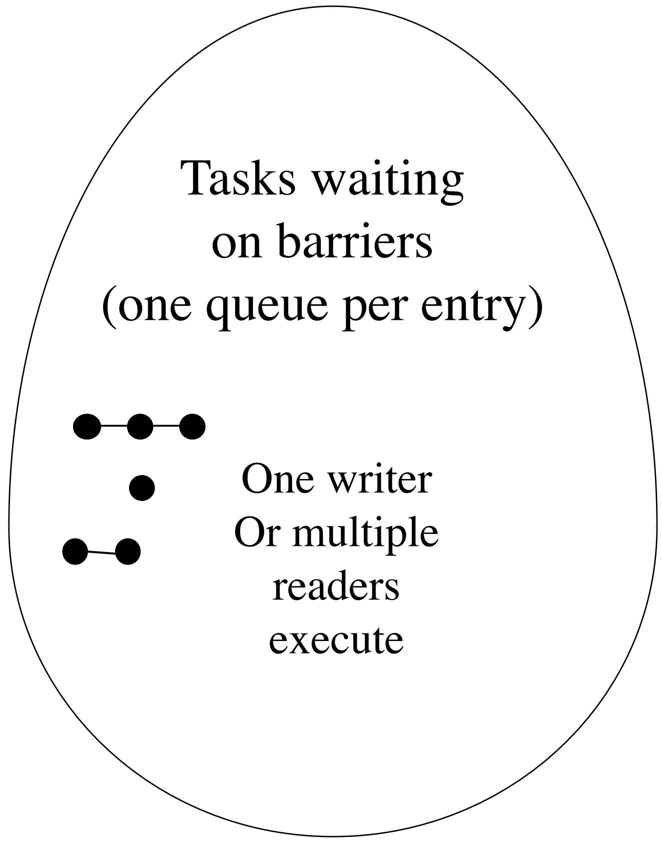
```
package body PAccounts is
 protected body PAccount is
  procedure Deposit
   (Amount: in Positive) is
  begin
   Balance := Balance + Amount;
  end Deposit;
  procedure Withdraw
   (Amount: in Positive) is
  begin
   Balance := Balance - Amount;
  end Withdraw;
  function Get_Balance
   return Natural is
  begin
   return Balance;
  end Get_Balance;
 end PAccount;
end Accounts;
```

Rules for Protected Types

- Functions are not allowed to modify the state of the protected object. They are considered readers.
- Procedures may modify the state of the protected object. They are considered writers.
- Data-based synchronization is possible through entries. Entries are similar to procedures, except that a condition can be associated with the entry call. If the condition is not verified, the call is queued.
  A condition is a boolean expression. It may contain function calls. It is unfortunately not possible to access parameters of the call.

The Eggshell Model

Tasks on the outside waiting (on one queue) to get in



At the end of a protected call, tasks already queued on entries take precedence over tasks waiting outside.

Tasks queued on the outside must wait until the object is quiescent

Default queuing policy FIFO within priorities

To execute, a task must enter the yolk!

Intelligent Printer (1)

- The intelligent printer knows how many sheets of paper are left in its tray.
- Jobs that require more paper than available will be put on hold.
- As soon as new sheets are inserted, the printer will first try and serve the jobs put on hold.

Intelligent Printer (2)

```
protected Printer is
   entry Print (Number_Of_Sheets : Natural);
   procedure Insert_Paper (Number : Natural);
private
   entry On_Hold (Number_Of_Sheets : Natural);
   procedure Real_Print
     (Number_Of_Sheets : Natural);
   Current_Sheets : Natural := 0;
   To_Try : Natural := 0;
end Printer;
```

Interface (Visible to the Outside)

Entries used for implementation purpose

Private fields

Intelligent Printer (3)

```
protected body Printer is
 entry Print (Number_Of_Sheets : Natural) when To_Try = 0 is
 begin
  if Number_Of_Sheets > Current_Sheets then
   requeue On_Hold with abort;
  else
   Real_Print (Number_Of_Sheets);
  end if;
 end Print;

 procedure Insert_Paper (Sheets : Natural) is
 begin
  Current_Sheets := Current_Sheets + Sheets;
  To_Try := On_Hold'Count;
 end Insert_Paper;
```

Intelligent Printer (4)

```
entry On_Hold (Number_Of_Sheets : Natural) when To_Try > 0 is
begin
 To_Try := To_Try - 1;
 if Number_Of_Sheets > Current_Sheets then
  requeue On_Hold with abort;
 else
  Real_Print (Number_Of_Sheets);
 end if;
end On_Hold;

procedure Real_Print (Number_Of_Sheets : Natural) is
begin
  -- Print job
  Current_Sheets := Current_Sheets - Number_Of_Sheets;
end Real_Print;
end Printer;
```

Advanced Concurrency Features (1)

- Task priorities (at least 31 values)
- Task identities

```
package Ada.Task_Identification is
  type Task_ID is private;
  function Current_Task return Task_ID;
end Ada.Task_Identification;
```

+ other procedures for examining a task's state, comparing and displaying ID's, and aborting tasks

- Task attributes
  - Associate data with every task
  - Instantiate the generic package Ada.Task_Attributes, passing as a parameter the type of the data and the initial value

Advanced Concurrency Features (2)

```
generic
  type Attribute is private;
  Initial_Value : in Attribute;
package Ada.Task_Attributes is
  function Value (T : Task_Id := Current_Task)
   return Attribute;
  procedure Set_Value
   (Val : in Attribute;
    T : in Task_ID := Current_Task);
end Ada.Task_Attributes;
```

Nice trick to monitor task termination:
Associate a controlled data type with the task

Concurrency in Java

- Class `Thread`, interface `Runnable`
- *Synchronized* Classes
- Add data to threads:
  - Class `ThreadLocal` or `InheritableThreadLocal`

```
class ThreadLocal {
 public ThreadLocal();
 public Object get();
 public void set(Object value);
 protected Object initialValue();
}

static ThreadLocal myData = new ThreadLocal();
myData.set(t);
```

Exceptions (1)

- Modern programming languages provide support for *exceptions*
  - Ada, C++, Java, Smalltalk, Eiffel
- No standard exception handling and ways of using exceptions
- Exceptions represent situations in which the normal execution of an operation can not be completed

Exceptions (2)

- Features a programming language must offer:
  - Means for declaring exceptions
  - Means for defining exception handling contexts
  - Means for declaring exception handlers and associating them with exception handling contexts
- In the context of fault tolerance, exceptions are used for forward error recovery [Goo75]

Exceptions (3)

- When an exception is raised in a context
  - Normal execution stops
  - Handler is searched among all attached handlers
  - Some schemes allow direct propagation to the enclosing context (often called signaling)
  - If no handler can be found at this level, the exception is propagated to the enclosing (or calling) context (found by inspecting the stack)
    - Dynamic hierarchical model
- Termination model vs. resumption model

Exceptions in Ada (1)

- Exceptions are provided to address
  - Error conditions - arithmetic overflow, storage exhaustion, array-bound violations, subrange violations, peripheral time-outs - by means of predefined exceptions, raised implicitly
  - Abnormal program conditions - errors in user input data, incorrect usage of abstract data types, need for special algorithms to deal with singularities - by means of user-defined exceptions, raised explicitly

Exceptions in Ada (2)

- Exceptions attached to blocks, procedures or functions

Declaring Exceptions

```
Ex1, Ex2: exception;

procedure X is
  [declarations]
begin
  …
exception
  when Ex1 => …;
end;
```

Handling an exception

```
[declare
  declarations]
begin
  …
exception
  when Ex2 =>
    …; raise Ex1;
  when others =>
    …;
end;
```

Throwing an exception

Matches any exception

Exceptions in Ada (3)

- Exceptions are not objects
- Predefined package Ada.Exceptions
  - Defines exception occurrences
  - Save / copy / re-raise exceptions
  - Allows to associate data in form of a string to an exception
  - Query exception name
  - Obtain debugging information
- Major drawback: Exceptions cannot be declared in method signatures
- Anonymous exceptions

Exceptions and Object-Orientation

- Exceptions are classes
  - Related exceptions can be grouped in exception hierarchies
- Makes it possible to
  - Write class-wide handlers
  - Add data through fields
- Some programming languages allow handlers to be attached to classes
  - Smalltalk

Exceptions and Concurrency

- Most main-stream languages do not integrate exceptions and concurrency
- Facile [TLP93] allows to declare the same exception handling context in several processes
- If there is no handler, the process terminates
- Complaints in ABCL1 [IY91]
- Exceptions raised during a rendez-vous in Ada are propagated to both caller and callee

Reflection

- General methodology for describing, controlling, and adapting the behavior of a computational system
- Static and dynamic execution characteristics of a system are made concrete in a so-called metaprogram
- By specializing the metaprogram, the programmer can observe or change the execution of the main program

Reflection and Object-Orientation

- Metaobjects [MAE87] can represent the base objects
  - Metaobjects observe, modify and control the behavior of the objects they represent
  - Many-to-many associations possible
- Hierarchical structure ⇒ reflective tower
- Interface between adjacent levels is called the meta-object protocol [KdB91]
- Transparency
  - Objects at one level are unaware of the presence and workings of the objects at the levels above

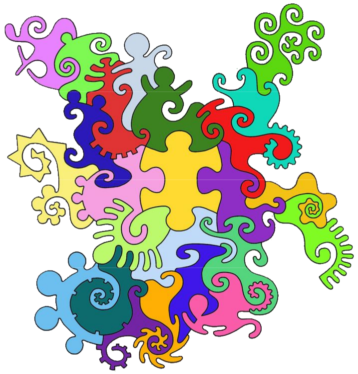
Reflection in Java

- Java provides limited reflection
- `java.lang.reflect`
  - Obtain information about classes and objects
  - Get number of methods and fields of a class
  - Query field types, types of parameters or return types of functions
  - Access “unknown” field values
  - Query class and member access modifiers

Background on Aspect-Orientation

- Aspect-oriented software development (AOSD) techniques aim to provide systematic means for the identification, separation, representation and composition of *crosscutting* concerns

- Aspect-Oriented Programming proposed in 1997
- First International Conference on Aspect-Oriented Software Development (AOSD) in 2002
- Today researchers apply aspect-orientation to all software engineering phases
- Industry adoption has started
  - Siemens / Motorolla / IBM



Problems with Object-Orientation

- Object-Orientation
  - Decompose problem into a set of abstractions
  - Objects
    - Encapsulate state and behavior
    - Are assigned responsibilities
- “Tyranny of the dominant decomposition” [T+99]
- Result:
  - Similar / identical code-fragments, all implementing some common functionality, are often scattered through the code

Object-Oriented Bank Application

```
class Bank {
  void transfer(Account a, Account b) {
    a.withdraw(100);
    b.deposit(100);
  }
}
```

```
class Account {
  int balance;
  void withdraw(int amount) {
    balance -= amount;
  }
  void deposit(int amount) {
     balance += amount;
  }
}
```

O-O Bank with Security

```
class Bank {
  void transfer(Account a, Account b) {
    a.authorize();
    a.withdraw(100);
    b.authorize();
    b.deposit(100);
  }
}
```

```
class Account {
  int balance;
  boolean authorized = false;
  int PIN = ...;
  public void withdraw(int amount) {
    if (!authorized)
      throw new SecurityException();
    else
      balance -= amount;
    authorized = false;
  }
  public void deposit(int amount) {
    if (!authorized)
      throw new SecurityException();
    else
      balance += amount;
    authorized = false;
  }
  public void authorize() {
    p = GUI.askForPIN();
    if (p == PIN) authorized = true;
  }
}
```

O-O Bank with Security

```
class Bank {
  void transfer(Account a, Account b) {
    a.authorize();
    a.withdraw(100);
    b.authorize();
    b.deposit(100);
  }
}
```

```
class Account {
  int balance;
  boolean authorized = false;
  int PIN = ...;
  public void withdraw(int amount) {
    if (!authorized)
      throw new SecurityException();
    else
      balance -= amount;
    authorized = false;
  }
  public void deposit(int amount) {
    if (!authorized)
      throw new SecurityException();
    else
      balance += amount;
    authorized = false;
  }
  public void authorize() {
    p = GUI.askForPIN();
    if (p == PIN) authorize = true;
  }
}
```

The security concern crosscuts the modules created by the object-oriented decomposition

Aspect-Oriented Programming

- Provide *new modularization features* at the programming language level *that allow to modularize crosscutting concerns*
- Modules implementing crosscutting functionality are called *aspects*
  - Aspects encapsulate *crosscutting state and behavior*
- Aspects are *woven* together to create final executed code
- Weaving happens at so-called *joinpoints*
- Benefits:
  - Simpler structure, improve readability, customizability and reuse
- Current main-stream aspect languages:
  - AspectJ (www.eclipse.org/aspectj), AspectC#, AspectC++, AspectC

AspectJ

- Aspect-Oriented extension of Java
- Pointcuts
  - Make it possible to name a set of joinpoints, e.g. method calls, setting or getting field values, etc.
- Advice
  - Specify behavior at joinpoints
- Introduction
  - Add fields / methods to classes
- Aspects
  - Group together pointcuts, advice and introductions

AspectJ Joinpoints (1)

- Joinpoints are identified using pointcut designators
- Methods and constructors
  - call(Signature), execution(Signature), initialization(Signature)
  - Example:
    call(public * Account.get*(..))
- Exeption handling
  - handler(TypePattern)
  - Example: handler(TransactionException+)
- Field accesses
  - get(Signature), set(Signature)
  - Example: get(private * Account+.*)

AspectJ Joinpoints (2)

- Objects
  - this(TypePattern), target(TypePattern), args(TypePattern, …)
  - Example:
    call(public * Account.get*(..)) && this(AccountManager)
- Lexical extent
  - within(TypePattern), withincode(Signature)
  - Example:
    call(public * Account.get*(..)) &&
    withincode(public void AccountManager.transfer())
- Based on control flow
  - cflow(Pointcut), cflowbelow(Pointcut)
  - Example:
  - cflow(public void AccountManager.transfer()) && call(public * Account.get*(..))

AspectJ Joinpoints (3)

- Conditional
  - if(Expression)
  - Example:
    if(debugEnabled) &&
    call (public * Account.*(..))
- Combination
  - !, &&, || and ()

AspectJ Pointcuts

- Pointcuts group together a set of joinpoints, and can pass on values from the execution context
- Examples:

```
pointcut PublicCallsToAccount(Account a) :
call(public * Account.*(..)) && target(a);

pointcut SettingIntegerFields(int newValue) :
set(* int Account.*) && args(newValue);
```

AspectJ Advice

- Add behavior *before*, *after* or *around* a joinpoint
- Example:

```
around PublicCallsToAccount(Account a) {
  if (a.blocked) {
    throw new AccountBlockedException();
  } else {
    proceed();
  }
}
```

AspectJ Introduction

- Even the account class does not provide the “block” functionality, we can add it through introduction
- Example:

```
private boolean Account.blocked = false;
public void Account.block() {
  blocked = true;
}
public void Account.unblock() {
  blocked = false;
}
```

AspectJ Aspects

- Aspects group everything relevant for implementing a particular concern

```
public aspect BlockableAccounts {
  pointcut PublicCallsToAccount (Account a) :
    call(public * Account.*(..)) && target(a);
  private boolean Account.blocked = false;
  public void Account.block() {
   blocked = true;
  }
  around PublicCallsToAccount(Account a) {
    if (a.blocked) {
      throw new AccountBlockedException();
    } else {
      proceed();
    }
  }
}
```

Aspect-Oriented Bank with Security

```
class Bank {
  void transfer(Account a, Account b) {
    a.withdraw(100);
    b.deposit(100);
  }
}
```

```
class Account {
  int balance;
  public void withdraw(int amount) {
    balance -= amount;
  }
  public void deposit(int amount) {
     balance += amount;
  }
}
```

```
aspect AccountSecurity {
  boolean Account authorized = false;
  int Account PIN = ...;
  void Account authorize() {
    p = GUI.askForPIN();
    if (p == PIN) authorize = true;
  }
  before (call public * Account+.*(..)
    && target(currentAccount)) {
    if (!currentAccount.authorized)
      throw new SecurityException();
  }
  after (call public * Account+.*(..)
    && target(currentAccount)) {
    currentAccount.authorized = false;
}
```

The security concern is nicely modularized within the *AccountSecurity* aspect

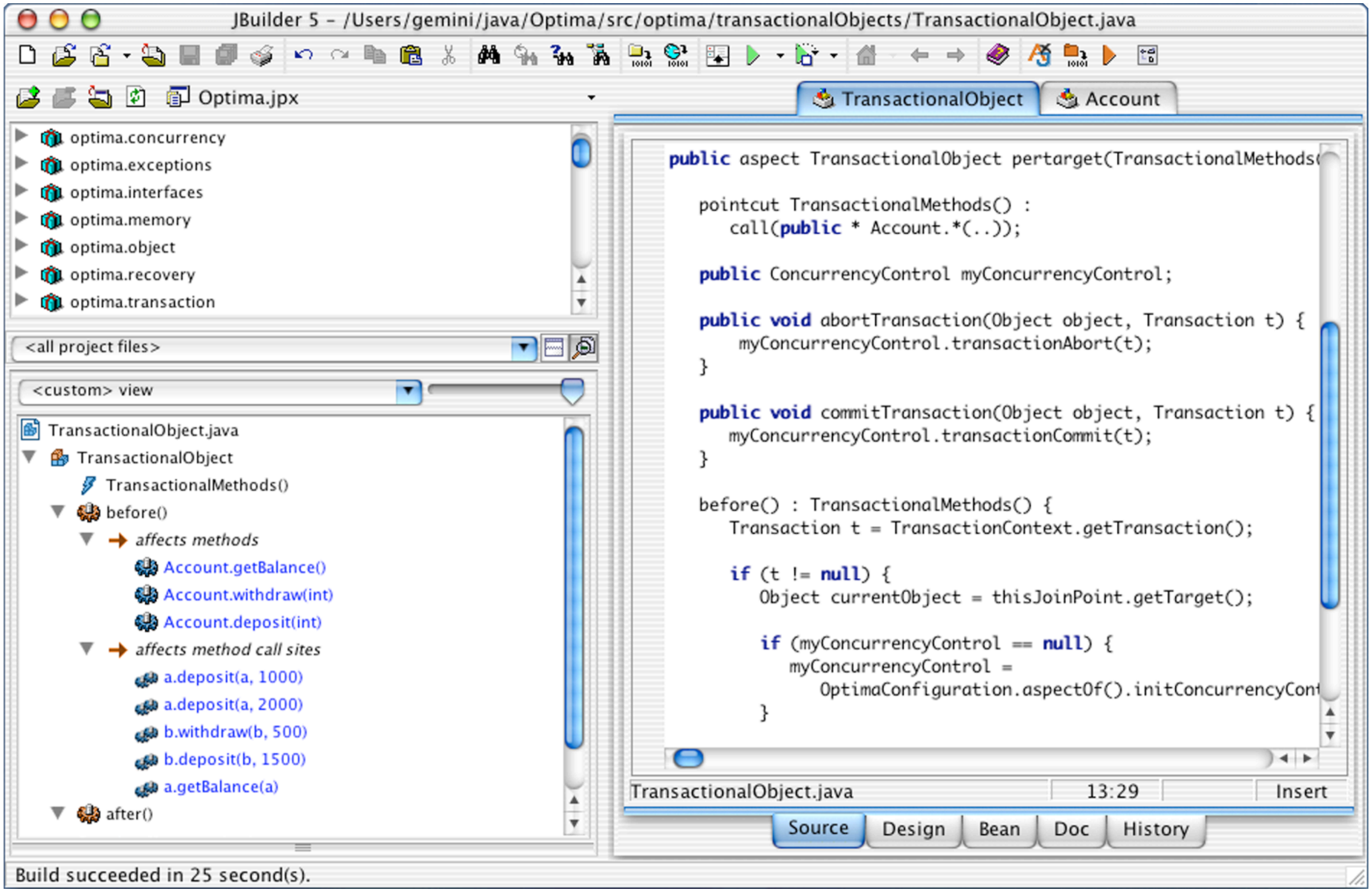
Advanced AspectJ

- Abstract aspects and pointcuts
- Implementing interfaces / inheritance
  - **`declare parents`** `: Account` **`implements`** `Blockable;`
  - **`declare parents`** `: Point` **`extends`** `GeometricObject;`
- Compile-time checking, e.g. for verifying programming conventions
  - **`declare error`** `: Pointcut : String;`
  - **`declare warning`** `: Pointcut : String;`
- Reflective access to run-time information through the static `thisJoinPoint` variable

AspectJ, How to Get Started

- http://www.eclipse.org/aspectj/
  or http://www.aspectbench.org
- AspectJ plug-ins exist for
  - Eclipse
  - Emacs
  - Sun Forte
  - JBuilder
- Version 1.0: Compile-time weaving
  Version 1.1: Byte-code weaving
  Current version: AspectJ 1.6.12 supports Java 1.6
  AspectJ 1.7.0M1 supports Java 1.7

AspectJ Plug-in for JBuilder



References (1)

- [Goo75]
  Goodenough, J. B.: “Exception Handling: Issues and a Proposed Notation”, Communications of the ACM 18(12), December 1975, pp. 683 – 696.
- [IY91]
  Ichisugi, Y.; Yonezawa, A.: “Exception Handling and Real-Time Features in Object-Oriented Concurrent Languages”, in Concurrency: Theory, Language and Architecture, pp. 92 – 109, Lecture Notes in Computer Science 491, Springer Verlag, 1991.
- [TLP93]
  Thomsen, B.; Leth, L.; Prasad, S.; Kuo, T.-M.; Kramer, A.; Knabe, F. C.; Giacalone, A.: “Facile Antigua Release – Programming Guide”. Technical Report ECRC-93-20, European Computer Industry Research Centre, Munich, Germany, December 1993.

References (2)

- [Mae87]
  Maes, P.: “Concepts and Experiments in Computational Reflection”, ACM SIGPLAN Notices 22(12), December 1987, pp. 147 – 155.
- [KdB91]
  Kiczales, G.; des Rivieres, J.; Bobrow, D. G.: The Art of the Meta-Object Protocol. MIT Press, Cambridge (MA), USA, 1991.
- [T+99]
  Tarr, P. L., et al.: “N Degrees of Separation: Multi-Dimensional Separation of Concerns”. In Proceedings of the 21st International Conference on Software Engineering (ICSE’1999), pp. 107-119, IEEE Computer Society Press / ACM Press, 1999.
- [RSB+98]
  Riehle, D.; Siberski, W.; Bäumer, D.; Megert, D.; Züllighoven, H.: “Serializer”, in Martin, R.; Riehle, D.; Buschmann, F. (Eds.): Pattern Languages of Program Design 3. Addison–Wesley, Reading, MA, USA, 1998, pp. 293 – 312.

References (3)

- [K+97]
  Kiczales, G.; Lamping, J.; Mendhekar, A.; Maeda, C.; Lopes, C.; Loingtier, J.-M.; Irwin, J. : “Aspect-Oriented Programming”. In Proceedings of the 11th European Conference on Object–Oriented Programming (ECOOP ’97), pp. 220 – 242, Jyváskylá, Finland, 1997, Lecture Notes in Computer Science 1241, Springer Verlag.
- [K+01]
  Kiczales, G.; Hilsdale, E.; Hugunin, J.; Kersen, M.; Palm, J.; Griswold, W. G.: “An Overview of AspectJ”. In Proceedings of the 15th European Conference on Object–Oriented Programming (ECOOP 2001), pp. 327 – 357, June 18–22, 2001, Budapest, Hungary, 2001, Lecture Notes in Computer Science 2072, Springer Verlag, 2001.