Adaptive Plug and Play Components for Evolutionary Software Development

Based on paper by Mira Mezini and Karl Lieberherr
appcs.ps in my ftp directory

The Problem

Object-oriented languages do not provide adequate constructs to capture collaborations between several classes.

Has been recognized in different forms in the object-oriented community:

❶ OO accommodates the addition of new variants of data types better than procedural programming

but, the opposite is true when new operations on existing data types are needed

❷ visitor pattern: the matter of concern -- definition of new operations on an existing object structure (aggregation is involved besides inheritance)

❸ several works complain the lack of constructs for expressing ***collaboration-based designs***

Collaboration(Role)-Based Designs

A methodology for decomposing object-oriented applications into a set of **classes** and a set of **collaborations**.

Collaboration --

a distinct (relatively independent aspect of an application that involves several participants, or roles

roles played by application classes

each class may play different roles in different collaborations

each role embodies a separate aspect of the overall class behavior

Collaboration(Role)-Based Designs

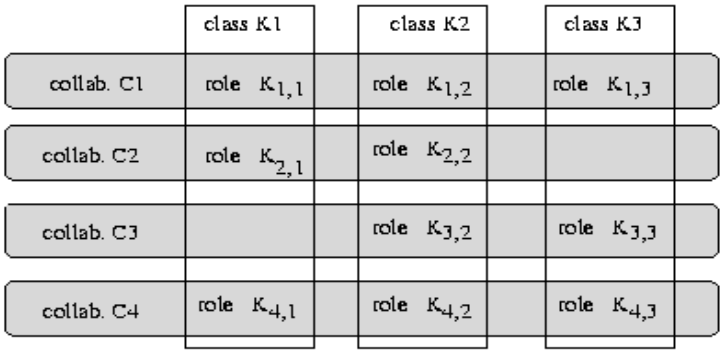
Require to view oo applications in two different ways:

(a) in terms of participants or types involved
(b) in terms of the tasks or concerns of the design

Collaboration(Role)-Based Designs

Require to view oo applications in two different ways:

(a) in terms of participants or types involved
(b) in terms of the tasks or concerns of the design



Not supported at the language level ==> gap between implementation and design

Implications for Testing, Verification and Validation

- Collaborations should be tested too, not just classes
- Collaborations have own invariants

Collaboration(Role)-Based Designs

Why do we need language constructs that capture collaborations:

unit of reuse is generally not a class, but a slice of behavior affecting several classes

this is the core of application framework.s but:

"because frameworks are described with programming languages, it is hard for developers to learn the collaborative patterns of a framework by reading it … it might be better to improve oo languages so that they can express patterns of collaboration more clearly"

[R. Johnson, CACM, Sep. '97]

Collaboration(Role)-Based Designs

Why do we need language constructs that capture collaborations:

single methods often make sense in a larger context

"oo technology can be a burden to the maintainer because functionality is often spread over several methods which must all be traced to get the "big picture"."

[Wilde at al., Software, Jan '93]

"object-oriented technology has not met its expectations when applied to real business applications and argues that this is partly due to the fact that there is no natural place where to put higher-level operations (such as business processes) which affect several objects. … if built into the classes involved, it is impossible to get an overview of the control flow. It is like reading a road map through a soda straw''

[Lauesen Software, April '98]

Language Constructs for Expressing Collaboration

Requirements on the design:

❶ orthogonal to the standard object-oriented models - not substitute, rather complement classes

❷ support a decomposition granularity that lies between classes and package modules

❸ support parameterization of collaborations with class graph information

❹ flexible composition mechanisms to support reusing existing collaborations to build more complex collaborations

Implications for Testing, Verification and Validation

- Parameterized collaborations allow for reuse of testing information
- Compositions of collaborations allow for reuse of testing information

Pricing Policies Example

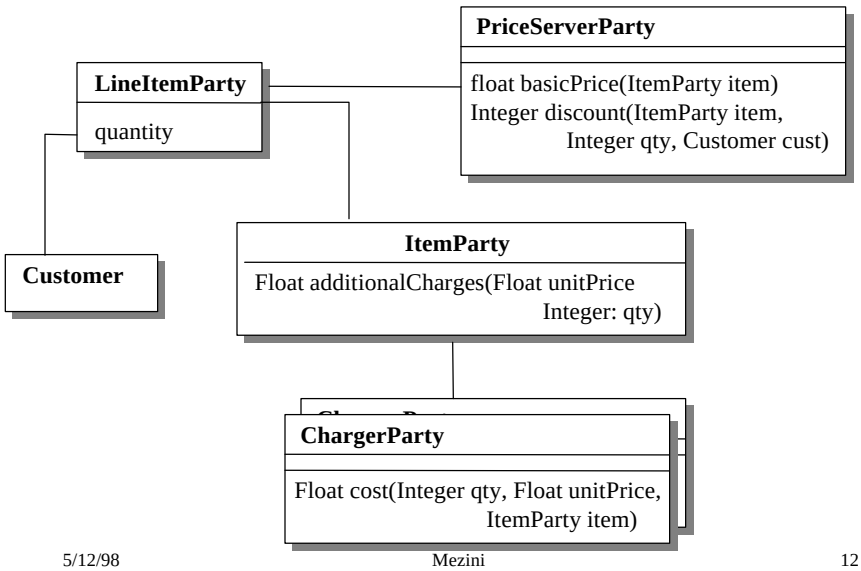
- example taken from Ian Holland's thesis
- comes from the domain of order entry systems
- originated from an application system generator developed at IBM ('70) called

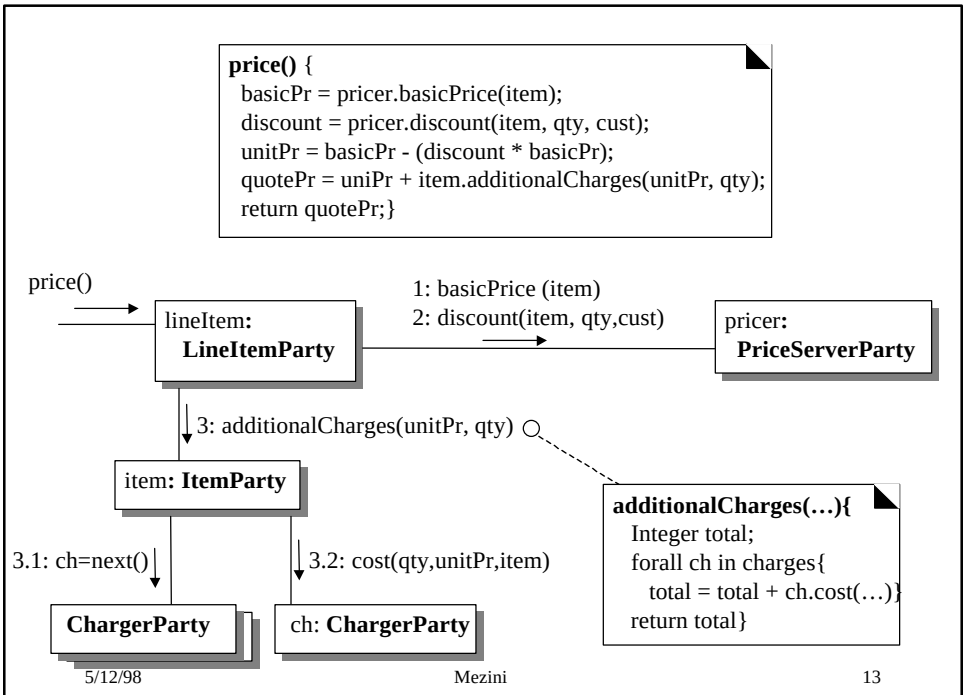
  Hardgoods Distributors Management Accounting System

  goal: encode a generic design for order entry systems which could be subsequently customized to produce an application meeting a customer's specific needs

- customer's specific requirements were recorded using a questionnaire
- the installation guide supplied with the questionnaire described the options and the consequences associated with questions on the questionnaire
- we consider only the pricing component of this application generator

A Collaboration Diagram for the Pricing Component





Pricing Policies Example

design is fairly simple

complexity is a problem with this application generator's component, though:

- the pricing component is described in nearly twenty pages of the installation guide
- the complexity results from numerous, and arbitrary, pricing schemes in use by industry and by the representation of these schemes in the system

Pricing Policies with APPCs

The price of an item may depend on:

- the type of the customer (government, educational, regular, cash, etc.),
- the time of the year (high/low demand season,
- whether cost-plus or discounting applies
- whether prior price negotiated prices involved,
- extra charges for th items such as taxes, deposits or surcharges
- … etc.

Language Constructs for Expressing Collaboration

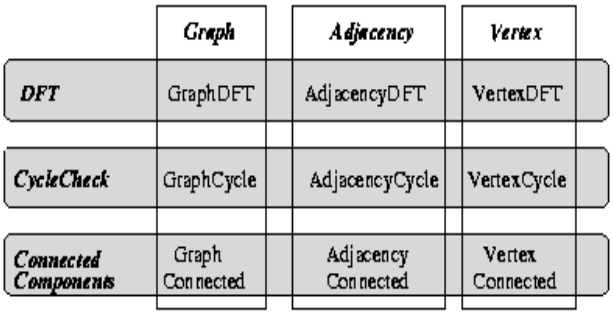
Requirements on the design:

❸ support parameterization of collaborations with class graph information

- ➢ Generic specification of the collaboration with respect to the class structure it will be applied to. This serves two purposes: (a) allow the same component to be used with several different concrete applications, and (b) allow a collaborative component to be mapped in different ways, i.e. with different class-to-participant mappings, into the same class structure.
- ➢ Loose coupling of behavior to structure to make collaborative components robust under changing class structures and thus better support maintenance

Language Constructs for Expressing Collaboration

- flexible composition mechanisms to support reusing existing collaborations to build more complex collaborations. Why?



Language Constructs for Expressing Collaboration

- flexible composition mechanisms to support reusing existing collaborations to build more complex collaborations. Why?

  Loose coupling among collaborations in the sense that their definition do

  The aim is to facilitate putting the same components into several

  - A composition mechanism that maintains the ``encapsulation" and

    other components. The aim is to avoid name conflicts and allow simultaneous execution of several collaborations even if these may

Pricing Policies with APPCs

APPC Pricing

Interface-to-Class-Structure

```
Interface-to-Class-Structure:
    s1 = from lineItem: LineItemParty to item: ItemParty to charges: ChargesParty;
    s2 = from lineItem: LineItemParty to pricer: PricerParty;
    s3 = from lineItem: LineItemParty to customer: Customer;

    PricerParty [
            Float basicPrice(ItemParty item);
            Integer discount(ItemParty item, Integer qty, Customer: customer); ]

    ChargesPart [
            Float cost(Integer qty, Float unitP, ItemParty: item ); ]
```


Pricing Policies with APPCs

APPC Pricing

Behavior

```
LineItemParty {
    public Float price (Integer qty ){
            Float basicPrice, unitPrice;
            Integer discount;
            basicPrice = pricer.basicPrice();
            discount = pricer.discount(item, qty, customer);
            unitPrice  = basicPrice - (discount * basicPrice);
            return (unitprice + additionalCharges(unitPrice, qty)); }

    Float additionalCharges(float unitP, Integer qty) {
            Interger total = 0;
            during s1 {
                ChargesParty{total += cost(qty, unitP, item); }
            return total;} }
    }
}
```

Pricing Policies with APPCs

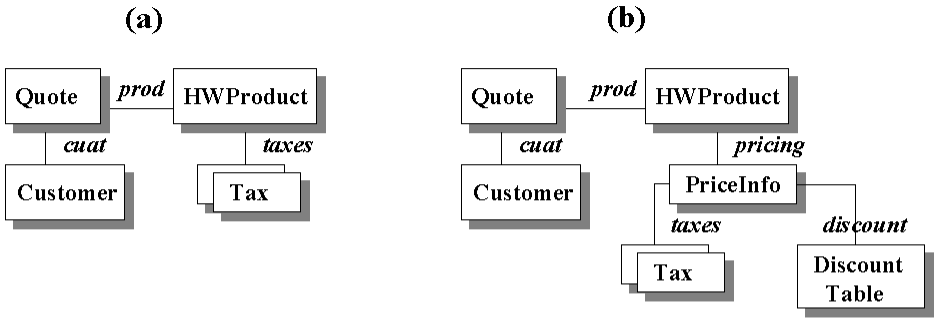
Appl.cd

HWProduct: <price> float <salePrice> float <taxes> {Tax} <discountTable> Table
Tax: <percentage> float;
Quote: <prod> HWProduct <quanttity> Integer <cust> Customer;
Customer: <name> String …;

Pricing Policies with APPCs

Appl.beh

```
class HWProduct {
   float salePrice() {return salePrice};
   float saleDiscount(Integer qty Customer c) {return 0};
   float regular-price() {return price};
   float regDiscount(HWProduct prod Integer qty Customer c)
                                                {return discountTable.lookUp(qty)};
 }

class Tax {
    float taxChange(Integer qty, float unitP  HWProduct p) {unitPrice * percentage /100}}

class Quote {
        integer quantity() {return quantity};

class Customer {
        float negProdPrice(HWProduct p) {…};
        float regProdDiscount(HWProduct p Integer qty Customer c) {…} }
```



Pricing Policies with APPCs

Let us generate different pricing schemes out of the generic pricing component specified by the pricing adjuster …

- Scheme 1: Regular Pricing

  each product has a base price which can be discounted depending on the number of the units ordered

- Scheme 2: Negotiated Pricing:

  A customer may have negotiated certain prices and discounts for particular items

Pricing Policies with APPCs

Scheme 1: Regular Price

```
Quote::+ {float regPrice() = Pricing with {
            LineItemParty = Quote;
            PriceParty = HWProduct
                              [basicPrice = regPrice;
                                discount = regDiscount;]
            ItemParty = HWProduct;
            ChargesParty = Tax
                              [cost = taxCharge]
          }
         }
```

Let see what this is supposed to generate:

Pricing Policies with APPCs

```
class Quote {
  ….
  public regPrice() {
            RegularPriceVisitor v = RegularPriceVisitor();
            return {v.price (this);}

  ….
            }
```

Pricing Policies with APPCs

```
class RegularPriceVisitor {

  public price (Quote host) {
            float basicPrice, quotePrice;
            Integer discount;
            Integer qty;
            qty = host.quantity();
            basicPrice = host.prod.regPrice();
            discount = host.prod.regDiscount(host.prod, qty, host.customer);
            unitPrice  = basicPrice - (discount * basicPrice);
            return (unitPrice +.additionalCharges(unitPrice, qty);
            }

  private additionalCharges(float unitPrice, Integer qty)
            { float total = 0;
              for all tax in host.prod.taxes
                        total = total + tax.taxCharge(float unitPrice, Integer qty)
            }
 }
```


Pricing Policies with APPCs

Scheme 2: Negotiated Price

```
Quote::+ {Float negPrice() = Pricing with {
            LineItemParty = Quote;
             PriceParty = Customer
                                [basicPrice = negProdPrice;
                                 discount = negProdDiscount;]
             ItemParty = HWProduct; \\
             ChargesParty = Tax\\
                                [cost = taxCharge]
             }
            }
```

Composing APPCs

```
APPC Marking {

    Interface
        s = from Graph to Adjacency to  Vertex to Adjacency
    Behavior
        Adjacency {
                bool marked = false;
                myRole() {
                    bool visited = marked;
                     if (marked == false) { marked = true; next()};
                     return visited;}
        }
      }
```

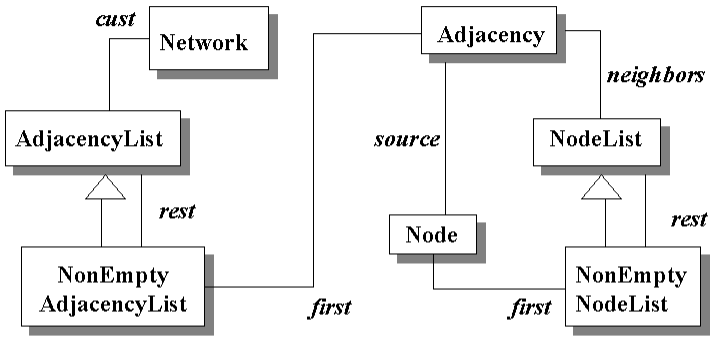

Composing APPCs

```
APPC Connectivity {

    Interface
        s: from Graph to-stop Adjacency
    Behavior
        Integer count = 0;
        return count;
        Adjacency {
                myRole() {
                        if ( next() == false ) { count += 1; } }
        }
  }
```

Composing APPCs

```
APPC DGCycleCheck {

    Interface
        s = from Graph to Adjacency to  Vertex to Adjacency
    Behavior
        Stack  stack = Stack new();
        Adjacency {
                myRole() {
                        if (stack.includes(this)) {
                                System.out.printIn(``cycle" + stack.print) }
                        else  { stack.add(this); }
                        next();
                        stack.remove(this); }
                }
        }
```


Composing APPCs

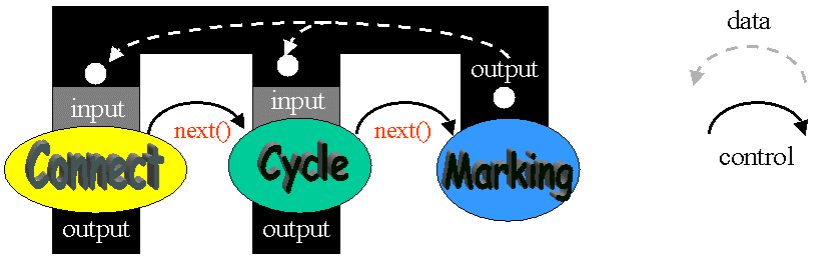


Want to do connectivity and cycle check simultaneously

Instantiating APPCs Compositions

```
ConnectivityAndCycleCheck =
        (Connectivity compose DGCycleCheck) (Marking)

s = Network via Adjacency through neighbors via Node through <-source
                                                    to Adjacency
Network ::+ {connectivityAndCycleCheck()
                = ConnectivityAndCycleCheck(s}
                        with {Network = Graph; Node = Vertex; }
```


APPCs Compositions



Conclusions

- Collaborations are a natural abstraction for designing systems
- Collaborations help with testing and validation
  - reuse of collaborations leads to reuse of testing information
    - parameterization with class graph information
    - composition of components