Software Architecture: Evolution and Mobility

J.L.Fiadeiro

University of Leicester

Plan

- Motivation:
  - why **Evolution** and **Mobility** matter
  - **Social** vs **physiological** complexity
- **Software Architectures**
  - Usage vs interaction
  - Components, connectors and configurations
- **Coordination** primitives for evolution
  - Externalisation of **interaction** in connectors
- **Location** primitives for mobility
  - Externalisation of **distribution** in connectors

Contributions

- ATX Software
- Antónia Lopes @ University of Lisbon
- The AGILE consortium
  - Univ. Munich
  - Univ. Pisa
  - Univ. Florence
  - Univ. Warsaw
  - Univ. Lisbon
  - ATX Software
  - ISTI – CNR

IST-2001-32747
Architectures for Mobility
Jan 02 – Apr 05

Coping with change

- In Business Systems today, change is the rule of the game…

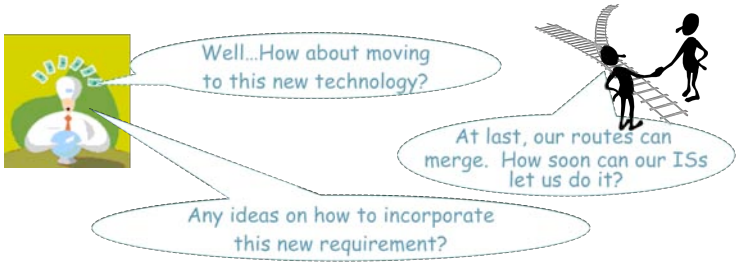
Coping with change

- In Business Systems today, change is the rule of the game…

> "… the ability to change is now more important than the ability to create [e-commerce] systems in the first place. Change becomes a first-class design goal and requires business and technology architecture whose components can be added, modified, replaced and reconfigured".
>
> P.Finger, "Component-Based Frameworks for E-Commerce", Communications of the ACM 43(10), 2000, 61-66.

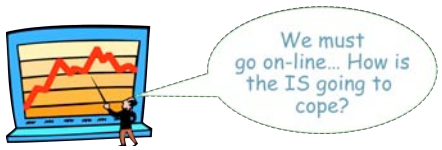
Coping with change

- In Business Systems today, change is the rule of the game…



Coping with change

- In Business Systems today, change is the rule of the game…
- The Web is only fuelling the rate of change… (B2C, B2B, P2P,…)



Coping with change

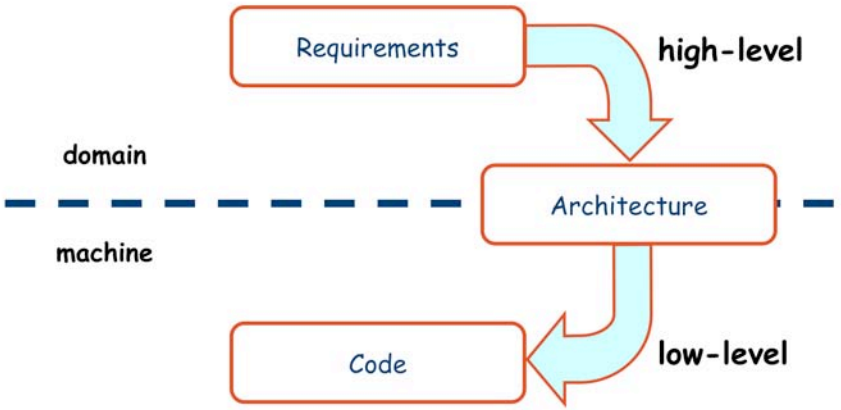
- In Business Systems today, change is the rule of the game…
- The Web is only fuelling the rate of change… (B2C, B2B, P2P,…)
- Critical infrastructures depend on the ability to react to failure by reconfiguring themselves (self-healing)…

Coping with change

- In Business Systems today, change is the rule of the game…
- The Web is only fuelling the rate of change… (B2C, B2B, P2P,…)
- Critical infrastructures depend on the ability to react to failure by reconfiguring themselves (self-healing)…
- The real-time economy…

Complexity is now on evolution…

Architectures in Software Design



What is it?

- A software architecture for a system is the structure or structures of the system, which comprise elements, their externally-visible behavior, and the relationships among them.

source: L.Bass, P.Clements and R.Kazman. Software Architecture in Practice. Addison-Wesley, 1998.

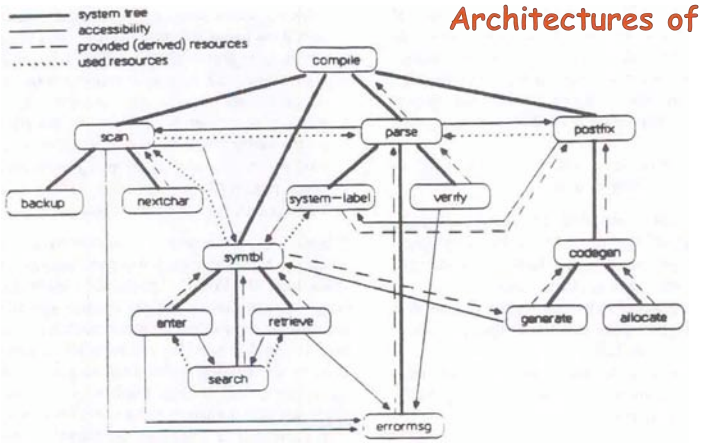
- There are many views, as there are many structures, each with its own purpose and focus in understanding the organisation of the system.

Module structures

- Code/implementation structures
- Address the global structure of a system in terms of
  - what its modules and resources are
  - how they fit together in the system
  - Definition/usage graphs
- **Modelling Interconnection Languages** for programming-in-the-large (DeRemer and Kron 75)

Module structures

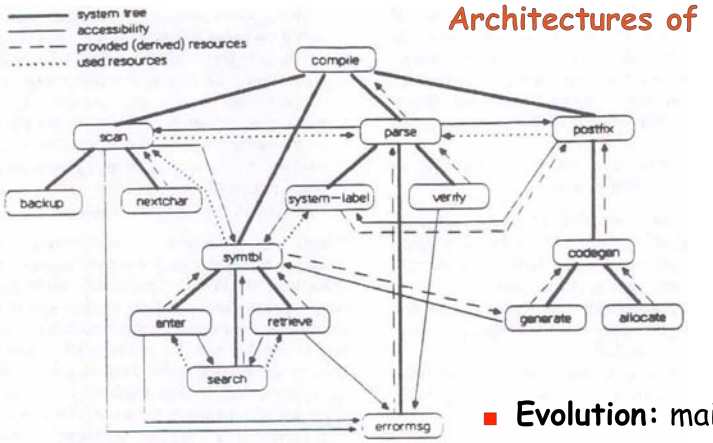
Architectures of Usage



Module structures

7

Architectures of Usage



- **Evolution:** maintenance

SEFM'04
University of Leicester

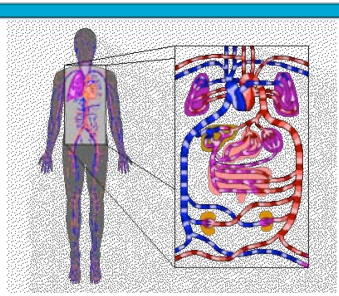
Two different relationships

- **Implements**
  - a given module is defined in terms of facilities provided by/to other modules;
  - composition mechanisms glue pieces together by indicating for each use of a facility where its corresponding definition is provided
- **Interacts**
  - components are treated as independent entities that may interact with each other along well defined lines of communication (connectors)

SEFM'04
University of Leicester

Two different notions of complexity

- **"Physiological" complexity**
  - static, linear interaction based on identities
  - compile or design time integration
  - **contracts of usage**



- **"Social" complexity**
  - dynamic, mobile and unpredictable interactions based on properties
  - "late" or "just-in-time" integration
  - **quality and trust**

SEFM'04
University of Leicester

Run-time Architectures

- The Components&Connectors view
- The "Interacts" relationship
- One generation later
  - Perry and Wolf (92)
  - Shaw and Garlan (96)
  - Bass, Clements, Kazman (98)
- Partly inspired by (civil) architects (Alexander)

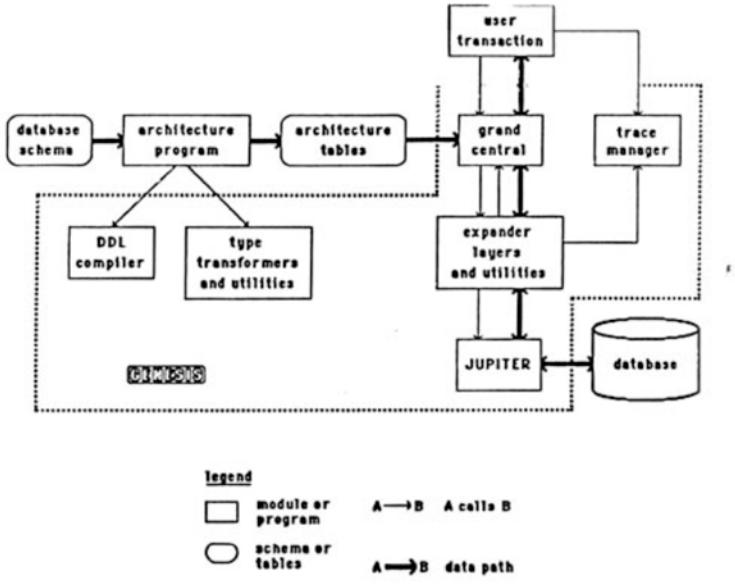


Figure 3.1 The Configuration of the GENESIS Prototype

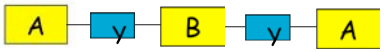
Genesis: A Reconfiguration Database Management System, D. S. Batory, J.R. Barnett, J.F. Garza, K.P. Smith, K. Tsukuda, B.C. Twichell, T.E. Wise, Department of Computer Sciences, University of Texas at Austin,

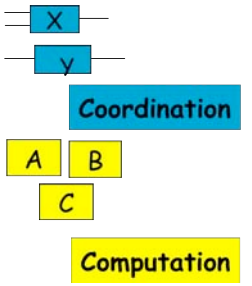
The challenge of evolution

- Reflect on the (run-time) architecture of the system the different levels of change that can take place in the application domain.
- Support evolution through dynamic reconfiguration, without interruption of service, minimising impact on the global system.

Summary

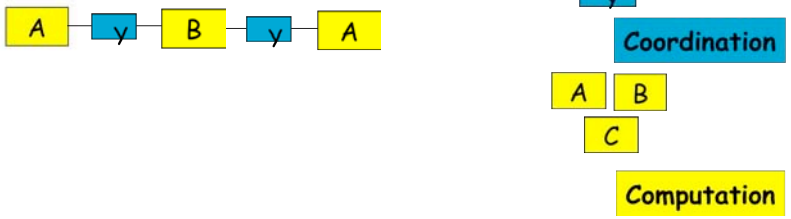
Architecture-based approaches





Summary

Architecture-based approaches

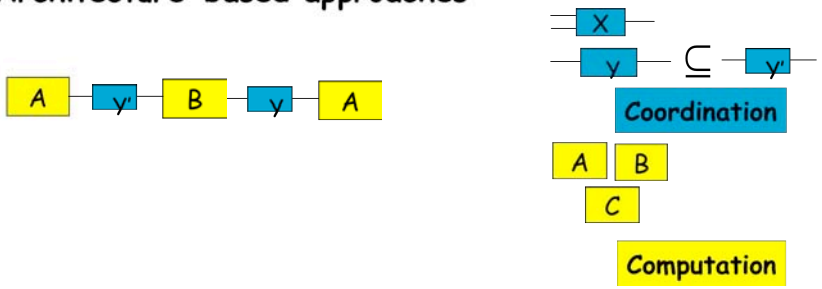


Compositionality wrt refinement

Summary

13

Architecture-based approaches



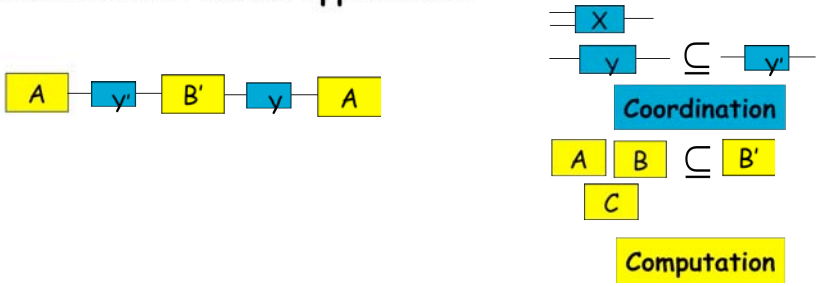
Compositionality wrt refinement

SEFM'04
University of Leicester

Summary

13

Architecture-based approaches



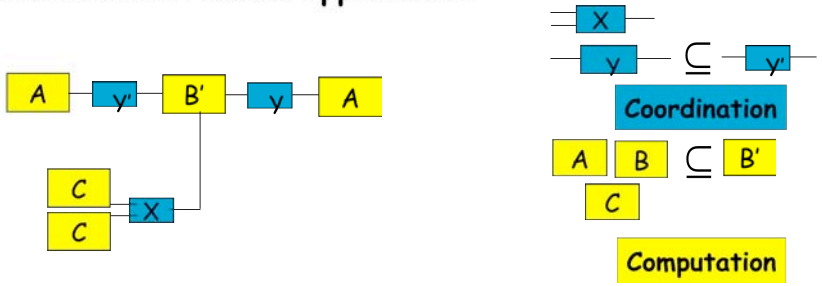
Compositionality wrt refinement

SEFM'04
University of Leicester

Summary

13

Architecture-based approaches



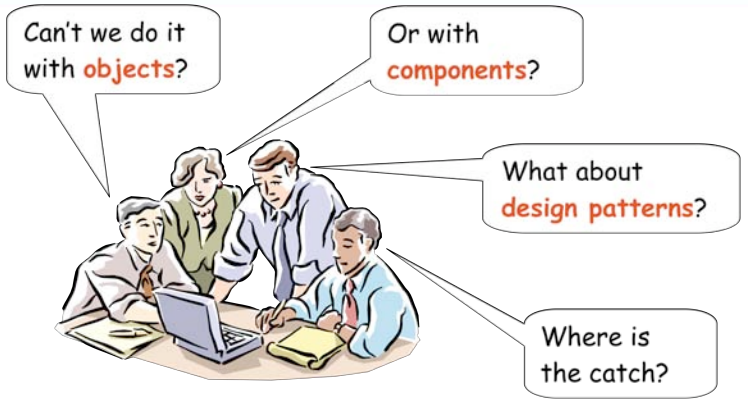
Compositionality wrt refinement
wrt evolution

SEFM'04
University of Leicester

Some FAQs...

Some FAQs...

14



SEFM'04 University of Leicester

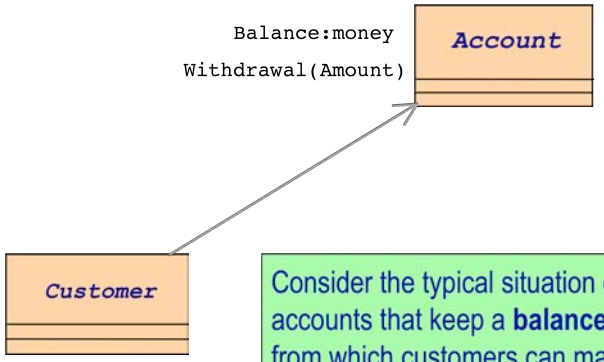
An example from Banking

Consider the typical situation of bank accounts that keep a **balance** and from which customers can make **withdrawals**.

SEFM'04 University of Leicester

An example from Banking

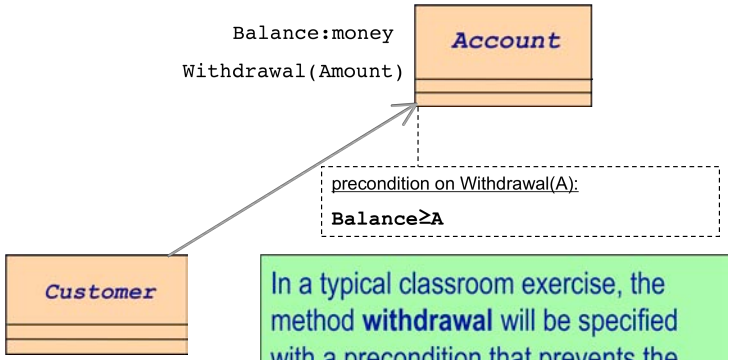
15



Consider the typical situation of bank accounts that keep a **balance** and from which customers can make **withdrawals**.

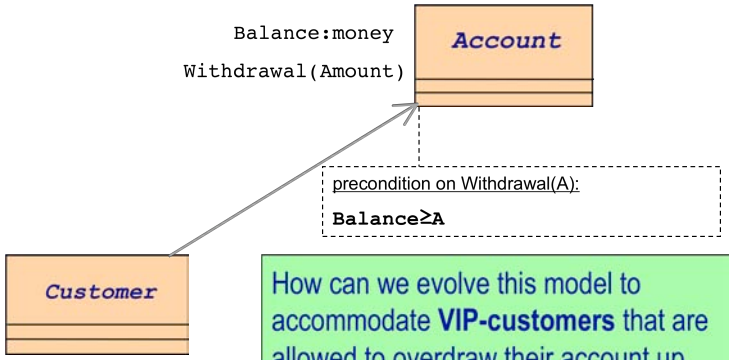
SEFM'04 University of Leicester

An example from Banking



In a typical classroom exercise, the method **withdrawal** will be specified with a precondition that prevents the account from being overdrawn.

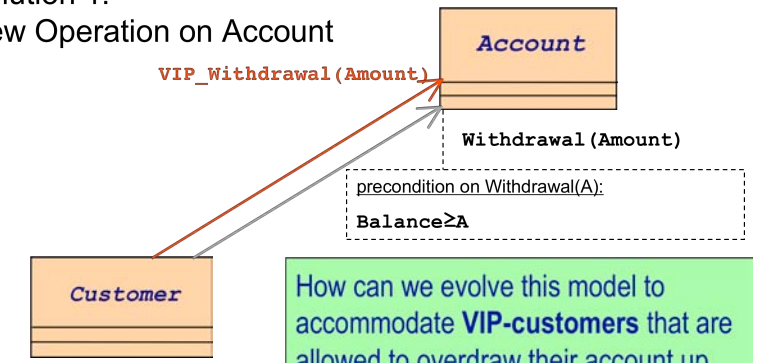
An example from Banking



How can we evolve this model to accommodate **VIP-customers** that are allowed to overdraw their account up to an agreed limit?

An example from Banking

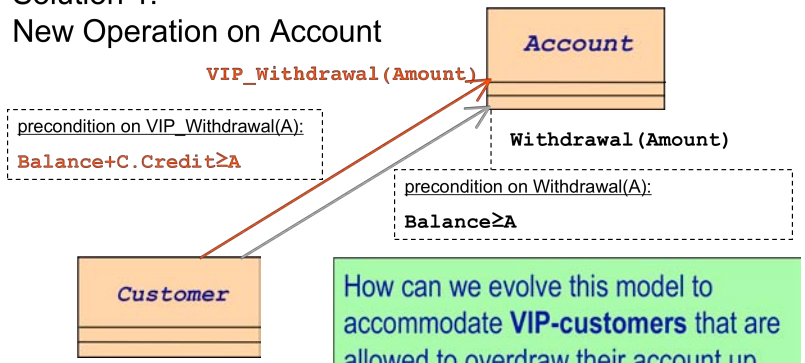
- Solution 1:
  New Operation on Account



How can we evolve this model to accommodate **VIP-customers** that are allowed to overdraw their account up to an agreed limit?

An example from Banking

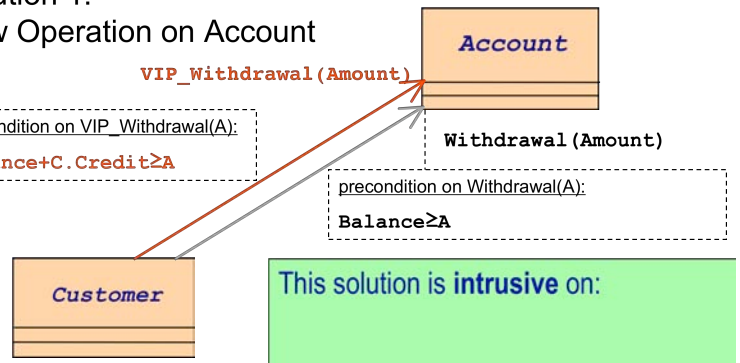
- Solution 1:
  New Operation on Account



How can we evolve this model to accommodate **VIP-customers** that are allowed to overdraw their account up to an agreed limit?

An example from Banking

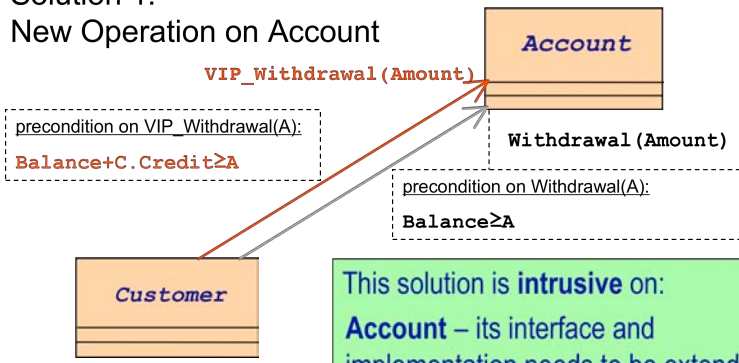
- Solution 1:
  New Operation on Account



This solution is **intrusive** on:

An example from Banking

- Solution 1:
  New Operation on Account

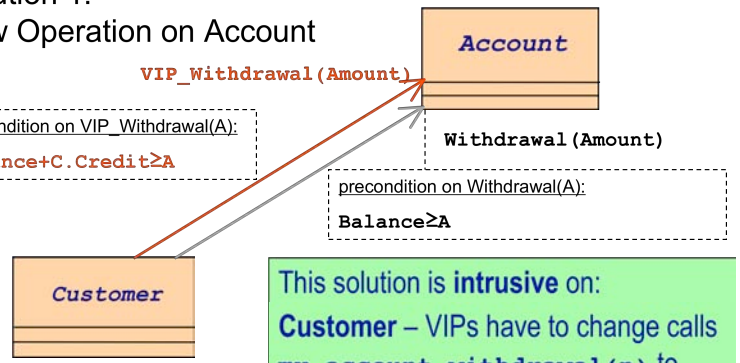


This solution is **intrusive** on:

Account – its interface and implementation needs to be extended with the new operation

An example from Banking

- Solution 1:
  New Operation on Account

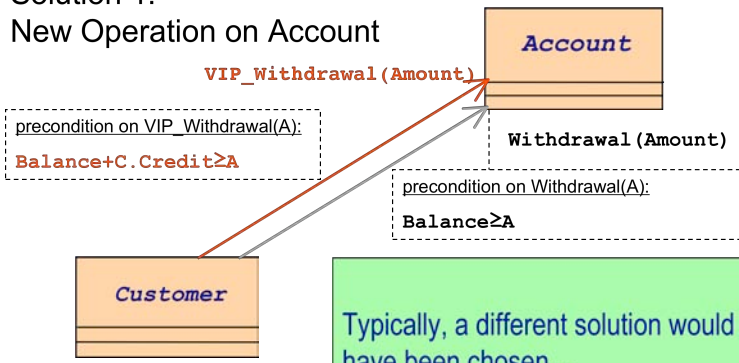


This solution is **intrusive** on:

Customer – VIPs have to change calls `my_account.withdrawal(n)` to `my_account.VIP_withdrawal(n)`

An example from Banking

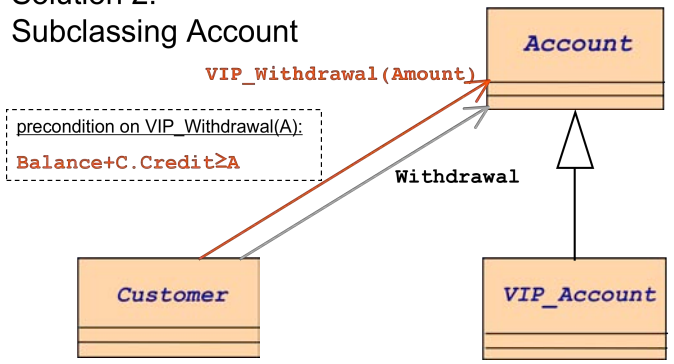
- Solution 1:
  New Operation on Account



Typically, a different solution would have been chosen…

An example from Banking

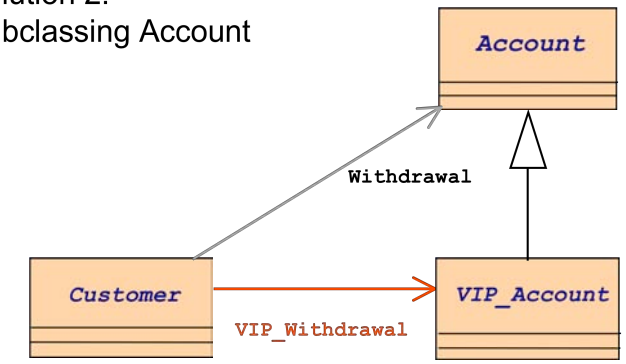
- Solution 2:
  Subclassing Account



An example from Banking

15

- Solution 2:
  Subclassing Account



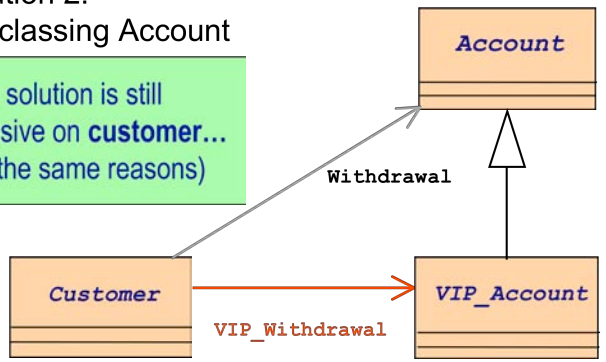
SEFM'04
University of Leicester

An example from Banking

15

- Solution 2:
  Subclassing Account

This solution is still intrusive on **customer**... (for the same reasons)



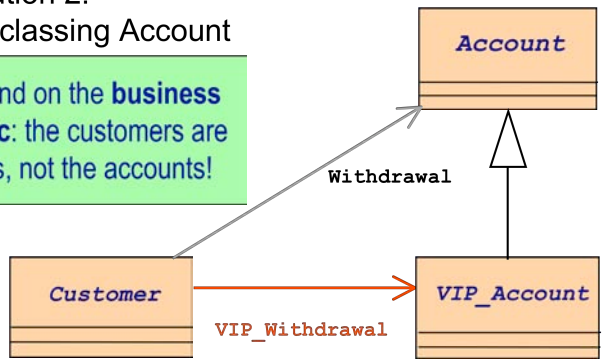
SEFM'04
University of Leicester

An example from Banking

15

- Solution 2:
  Subclassing Account

... and on the **business logic**: the customers are VIPs, not the accounts!



SEFM'04
University of Leicester

An example from Banking

- Problems with Solutions 1,2

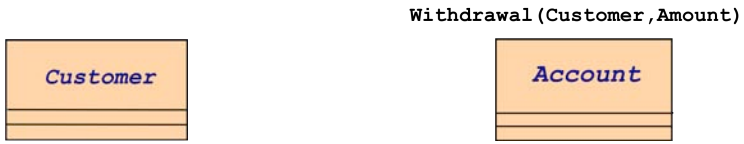
An example from Banking

16

- Problems with Solutions 1,2
  - They are intrusive on the code…
  - …and on the interconnections
  - …and on the business logic
- This is because, through clientship, business rules get **encoded** in the methods, and the methods reside in the server side.

SEFM'04 University of Leicester

A better solution

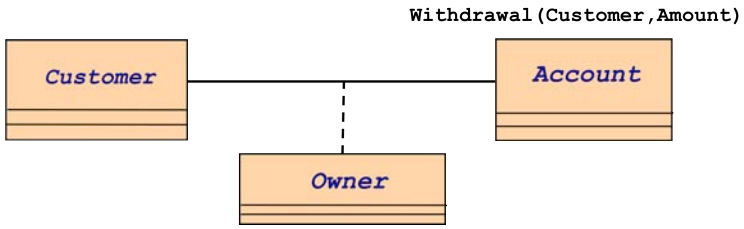


SEFM'04 University of Leicester

A better solution

17

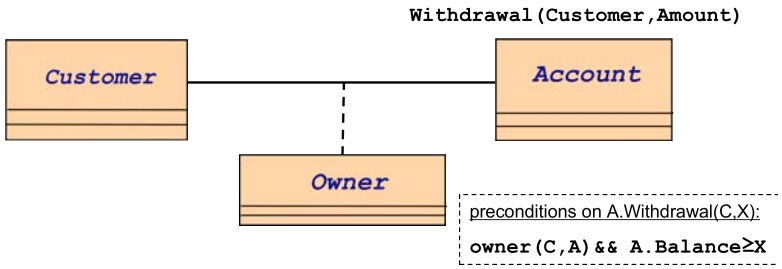
Model the relationship between customers and accounts as an **association class**…



SEFM'04 University of Leicester

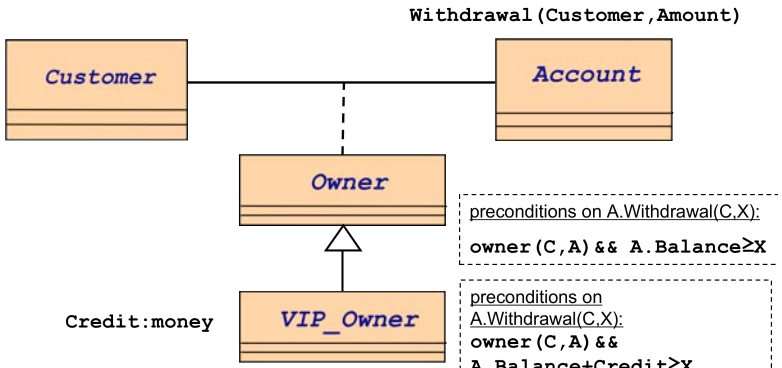
A better solution

… on which the "business rule" can be placed



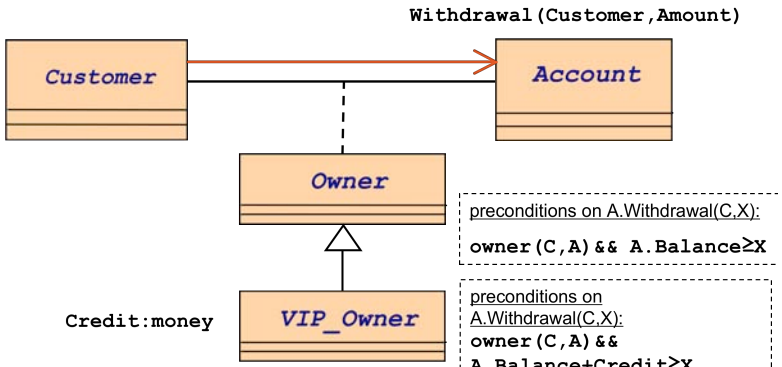
A better solution

… and specialise it to **evolve** the business rule



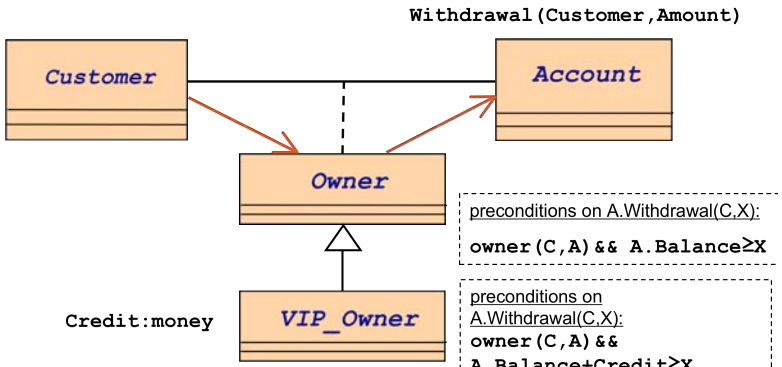
But still some problems

If the association is implemented through attributes and direct calls, we get the same problems as before…



But still some problems

A better way is to use a **mediator** through which the calls can be redirected and managed…



Mediator

- Problems with the mediator

Mediator

18

- Problems with the mediator
  - Each mediation is intrusive on the code…
  - …because it is managed explicitly by the objects involved
  - This also means that it can be interrupted
  - …and even by-passed
  - On the other hand, additional business rules means additional mediators and mediation between them…

SEFM'04 University of Leicester

OO is identity-based

- What is intrinsically "wrong" with OO:

SEFM'04 University of Leicester

OO is identity-based

19

- What is intrinsically "wrong" with OO:
  - Feature calling, the basic mechanism through which objects can interact, is **identity-based**: objects call specific features of specific objects (clientship);
  - As a result, any change on the interactions is **intrusive** on the code of the object.

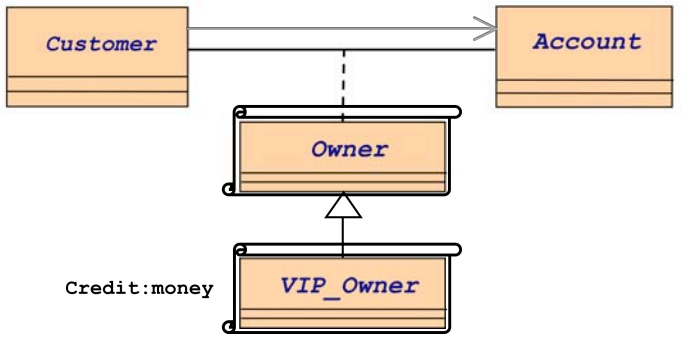
SEFM'04 University of Leicester

OO is identity-based

- What is intrinsically "wrong" with OO:
  - Feature calling, the basic mechanism through which objects can interact, is **identity-based**: objects call specific features of specific objects (clientship);
  - As a result, any change on the interactions is **intrusive** on the code of the object.
- We propose a way for interactions to be ***externalised*** and handled as first-class citizens.

Contracts for Change

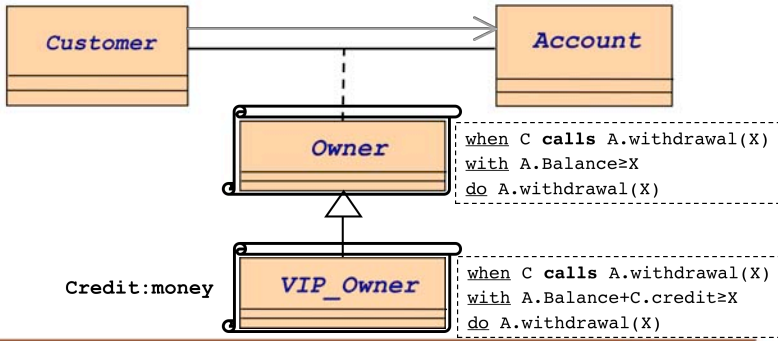
- A solution inspired on Architectural Connectors



Contracts for Change

- A solution inspired on Architectural Connectors

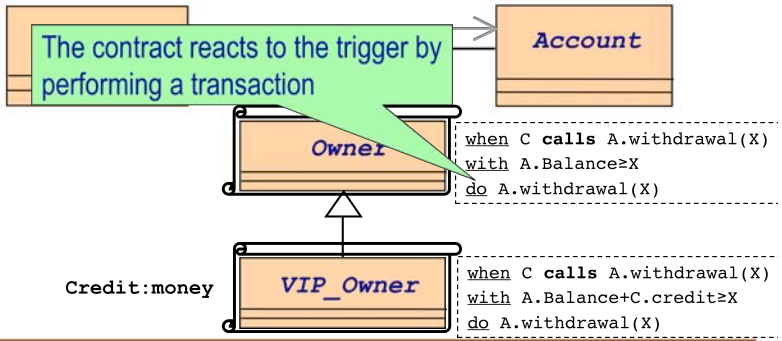
The customer still calls the account but the call is intercepted by the contract, without any of the parties being aware…



Contracts for Change

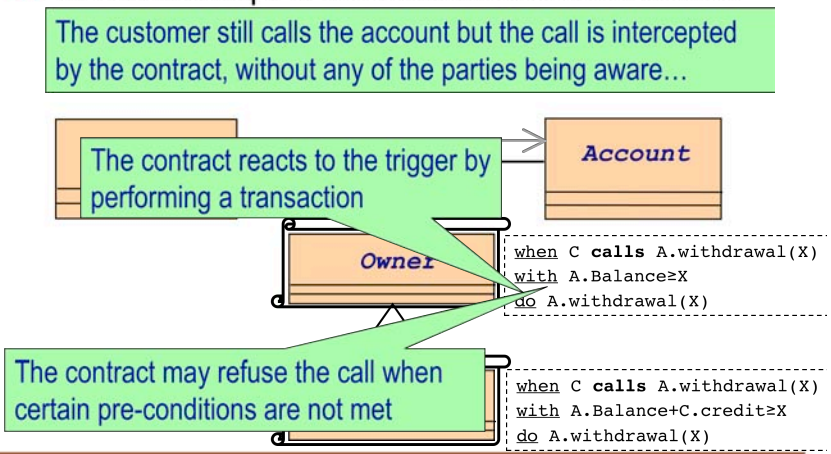
- A solution inspired on Architectural Connectors

The customer still calls the account but the call is intercepted by the contract, without any of the parties being aware…



Contracts for Change

- A solution inspired on Architectural Connectors



The CCC approach

- **A confluence of contributions from**
  - Coordination Languages and Models
    Separation between "computation" and "coordination"
  - Software Architectures
    Connectors as first-class citizens
  - Parallel Program Design
    Superposition
- **An Academia/Industry partnership**

ATX SOFTWARE ⟺ / University of Leicester

The CCC approach

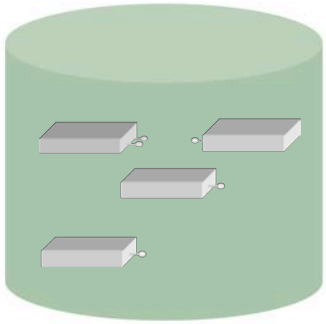
- **The Strategy**

The CCC approach

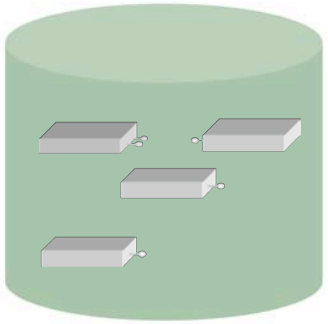
- **The Strategy**

- Recognize that change in the application domain occurs at different levels;

- Distinguish Computation Resources…

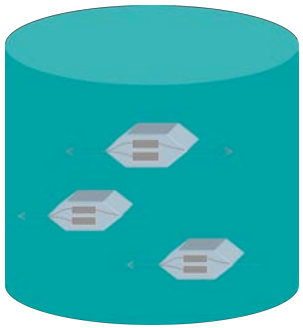


- Distinguish Computation Resources…

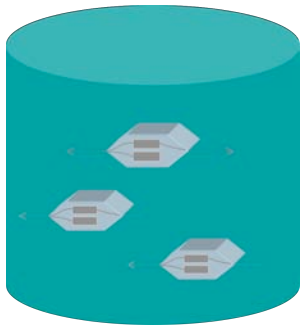


- Units that model **core** business/domain entities and provide services through computations performed **locally**
- These tend to be **stable** components, for which modifications imply major re-engineering

- …from Coordination Resources



- …from Coordination Resources



Units that model **volatile** "business" rules and processes and can be **superposed, at run time**, on the core units to…

- …**coordinate** their interactions
- …**regulate** their behaviour
- …**adapt** their behaviour
- …**monitor** their behaviour

The CCC approach

- The Strategy
  - Recognize that change in the application domain occurs at different levels;

The CCC approach

25

- The Strategy
  - Recognize that change in the application domain occurs at different levels;
  - Reflect these levels in the architecture of the system;

SEFM'04 University of Leicester

The CCC approach

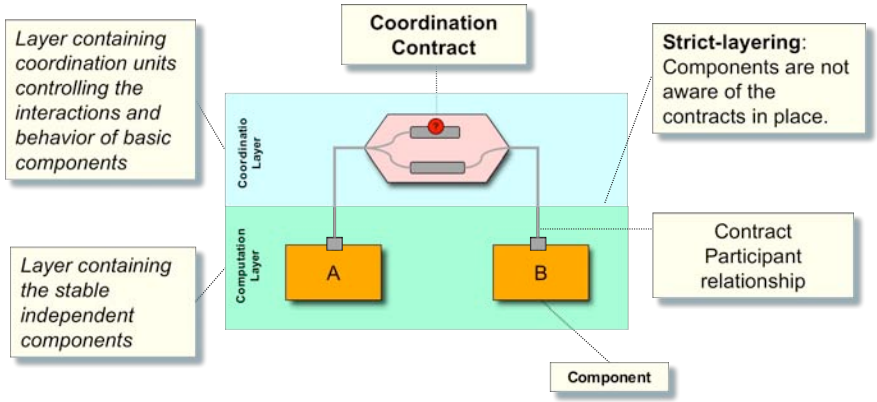
- Change-oriented layered architecture

SEFM'04 University of Leicester

The CCC approach

26

- Change-oriented layered architecture



SEFM'04 University of Leicester

The Strategy

- Recognize that change in the application domain occurs at different levels;
- Reflect these levels in the architecture of the system;

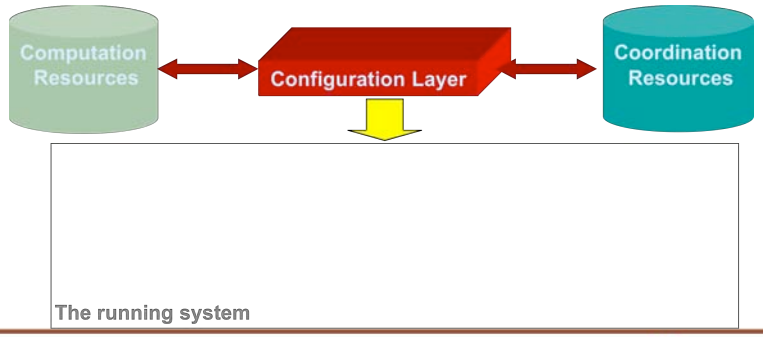
The Strategy

- Recognize that change in the application domain occurs at different levels;
- Reflect these levels in the architecture of the system;
- Manage evolution according to the architecture.

The Configuration Layer

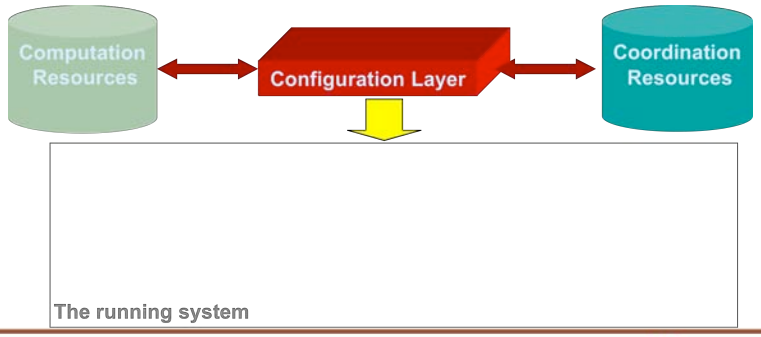
The Configuration Layer

Services that model business activities and through which the system can be configured, at run-time, to provide the required response.



- The Configuration Layer

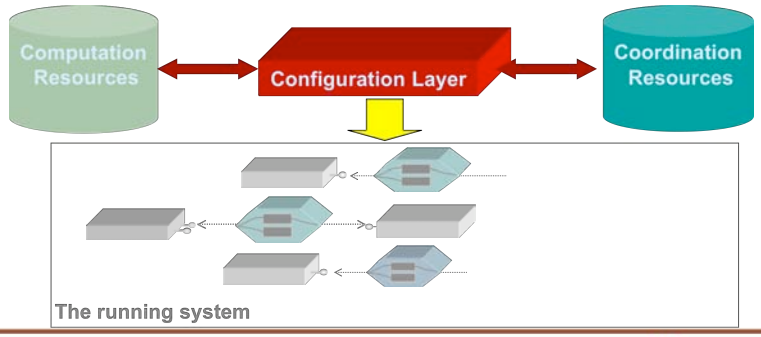
These services can be either invoked by authorized users or triggered by events (self-adaptation).



The CCC approach 28

- The Configuration Layer

These services can be either invoked by authorized users or triggered by events (self-adaptation).

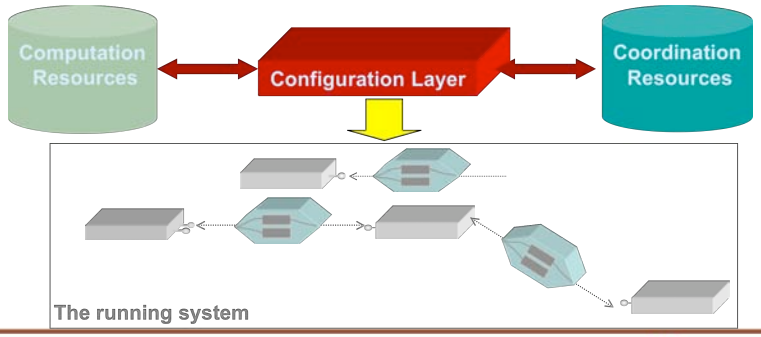


SEFM'04 University of Leicester

The CCC approach 28

- The Configuration Layer

These services can be either invoked by authorized users or triggered by events (self-adaptation).



SEFM'04 University of Leicester

SEFM'04 University of Leicester

The CCC approach

- Semantic primitives for **Coordination**
- Semantic primitives for **Configuration**
- Full mathematical **semantics – CommUnity**
- A **micro-architecture** for deployment over platforms for component-based development
- An instantiation of this micro-architecture for Java components – the **Coordination Development Environment (CDE)**

Semantic primitives for coordination

- **Coordination laws** that provide abstract models of services in terms of reactions to be performed upon detection of triggers.
- **Coordination interfaces** that identify the types of components that can instantiate the service as a law.

Overview


```
coordination law standard-withdrawal
partners
     a:account-debit;
     c:customer-withdrawal
rules
          when  c.withdrawal(n,a)
          with  a.balance()≥n & c.owns(a)
          do    a.debit(n);
end law
```

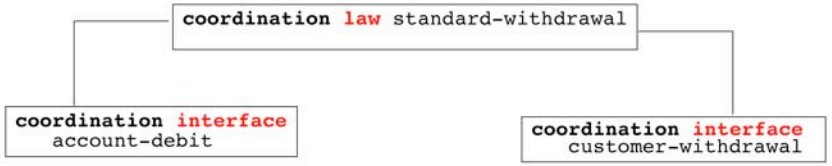

Overview


```
coordination law standard-withdrawal
partners
     a:account-debit;
     c:customer-withdrawal
rules
          when  c.withdrawal(n,a)
          with  a.balance()≥n & c.owns(a)
          do    a.debit(n);
end law
```

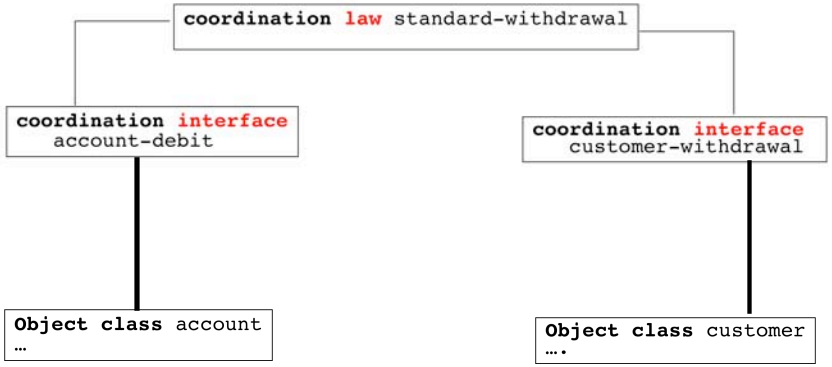
```
coordination interface
   account-debit
import types
   money;
services
   balance():money;
   debit(a:money): post balance()
       = old balance()-a
end interface
```

```
coordination interface
   customer-withdrawal
import types
   money, account;
services
   owns(a:account):Boolean
events
   withdrawal(n:money;a:account)
end interface
```

Just-in-time integration

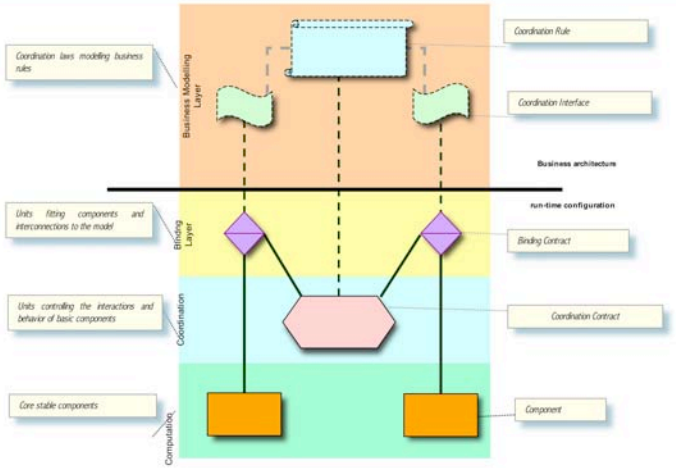


Just-in-time integration



Binding may require **adaptation**…

Instantiation



Coordination interfaces

- **Coordination interfaces** correspond to the roles of architectural connectors.
- They identify types of components according to services and events:
  - ***services*** identify operations that components that are instances of the interface need to provide for a contract to operate according to the law;
  - ***events*** identify situations produced during the execution of the components that are required to be detected as triggers for the contract to react and activate a *coordination rule* as discussed below.

Coordination interfaces


```
coordination interface customer-withdrawal
import types     money, account
services    owns(a:account):Boolean
events    withdrawal(n:money;a:account)
end interface
```

- We require to detect as triggers events that consist of customers performing withdrawals, and be provided with services that query about the account ownership relation
- In traditional object-oriented modelling, typical events are feature calls: a withdrawal would normally be modelled as a direct call to the debit operation of the corresponding account – a.debit(n).

Coordination interfaces


```
coordination interface account-debit
import types     money
services
       balance():money
       debit(n:money)  post balance()= old balance – a
end interface
```

- The inclusion of properties, e.g. pre and post-conditions on services, provide means for requirements to be specified on the components that can be bound to the interface.
- A special section `properties` may be used for other kinds of requirements.

Coordination laws


```
coordination law standard-withdrawal
partners     a:account-debit; c:customer-withdrawal
rules  when   c.withdrawal(n,a)
       with   a.balance()≥n and c.owns(a)
       do     a.debit(n)
end law
```

- A coordination law corresponds to a connector (type).
- The partners are logical parameters typed by coordination interfaces and correspond to the connector's roles.
- The coordination rules provide the glue of the connector.

Coordination rules


```
coordination law standard-withdrawal
partners     a:account-debit; c:customer-withdrawal
rules  when   c.withdrawal(n,a)
       with   a.balance()≥n and c.owns(a)
       do     a.debit(n)
end law
```

- Each coordination rule identifies, under `when`, a trigger to which the contracts that instantiate the law will react – a request by the customer for a withdrawal in the case at hand.
- The trigger can be just an event observed directly over one of the partners or a more complex condition built from one or more events.

Coordination rules


```
coordination law standard-withdrawal
partners     a:account-debit; c:customer-withdrawal
rules  when   c.withdrawal(n,a)
       with   a.balance()≥n and c.owns(a)
       do     a.debit(n)
end law
```

- Under `with` we include conditions (guards) that should be observed for the reaction to be performed.
- If any of the conditions fails, the reaction is not performed and the occurrence of the trigger fails.
- Failure is handled through whatever mechanisms are provided by the language used for deployment.

Coordination rules


```
coordination law standard-withdrawal
partners     a:account-debit; c:customer-withdrawal
rules  when   c.withdrawal(n,a)
       with   a.balance()≥n and c.owns(a)
       do     a.debit(n)
end law
```

- The reaction to be performed to occurrences of the trigger is identified under `do` as a set of operations – a debit for the amount and on the account identified in the trigger.
- This set may include services provided by one or more of the partners as well as operations that are proper to the law itself.
- The whole interaction is handled as a single transaction.

Example: VIP-withdrawal


```
coordination law VIP-withdrawal
partners     a:account-debit; c:customer-withdrawal
operation    credit():money
rules  when   c.withdrawal(n,a)
       with   a.balance()+credit()≥n and c.owns(a)
       do     a.debit(n)
end law
```

- The credit-limit is assigned to the law itself rather than the customer or the account. This is because we may want to be able to assign different credit limits to the same customer but for different accounts, or for the same account but for different owners.
- Would it make sense to have a separate partner of the law providing the credit?

Interfacing with external events

- Coordination interfaces can also act as useful abstractions for either events or services that lie outside the system, or global phenomena that cannot be localised in specific components.
- In the case of events, this allows for the definition of reactions that the system should be able to perform to triggers that are either global (e.g. a deadline) or are detected outside the system.
- In the case of reactions, this allows us to identify services that should be procured externally. This is particularly useful for B2B operations and the modelling of Web-services.

Example: interfaces for transfers


```
coordination interface external-transfer
import types    money, account, transfer-id
events   transfer(n:money;a:account;t:transfer-id)
end interface
```

```
coordination interface account-credit
import types    money
Services     credit(n:money)
end interface
```


Example: law for transfers


```
coordination law external-transfer-handler
partners    a:account-credit; t:external-transfer
operation   ackn(t:transfer-id)
rules  when  transfer(n,a,t)
       with  a.exists
       do    n≥1000:a.credit(n-100)
             and n<1000:a.credit(n*0.9)
             and ackn(t)
end law
```


Monitoring behaviour

- Assume that new legislation is passed that requires credits over a certain amount to be reported to the central bank – e.g. as a means of detecting money laundering.
- Rather than revise the implementation of credits to take care of this new requirement, it is better to superpose a contract over every account to perform the required monitoring activity.

Example: monitoring big credits


```
coordination law report-big-credits
partners      a:account-credit-event
operation     big():money;
              report(n:money);
              set-big(n:money) post big()=n
rules  when   a.credit(n) and n≥big()
       do     report(n)
end law
```

```
coordination interface account-credit-event
import types    money
events         credit(n:money)
end interface
```

Regulating behaviour

- Contracts can also be used for superposing *regulators* over certain components of the system.
- For instance, consider the situation in which the bank decides to penalise customers who fail to keep a given minimum average balance by charging a monthly commission.

Example: commission on balances


```
coordination law commission-on-low-balance
partners      a:average-balance
operation     minimum(), charge():money
rules
       when   end-of-month
       do     minimum()>a.average():a.debit(charge())
end law
```

```
coordination interface average-balance
import types     money
services  debit(n:money); average():money
end interface
```


Example: openplan

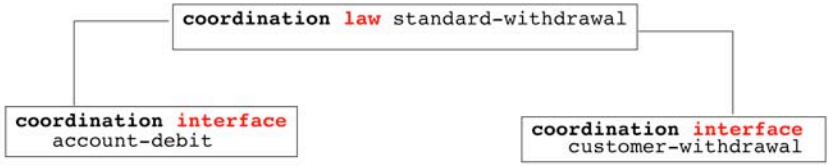

```
coordination law flexible-package
partners      c,s:account-debit&credit
operation     minimum(), maximum():money
rules
       when   c.balance()<minimum()
       do     let N=min(s.balance(),maximum()-c.balance())
              in s.debit(N)and c.credit(N)

       when   c.balance()>maximum()
       do     let N=c.balance()-maximum()
              in s.credit(N)and c.debit(N)
end law
```

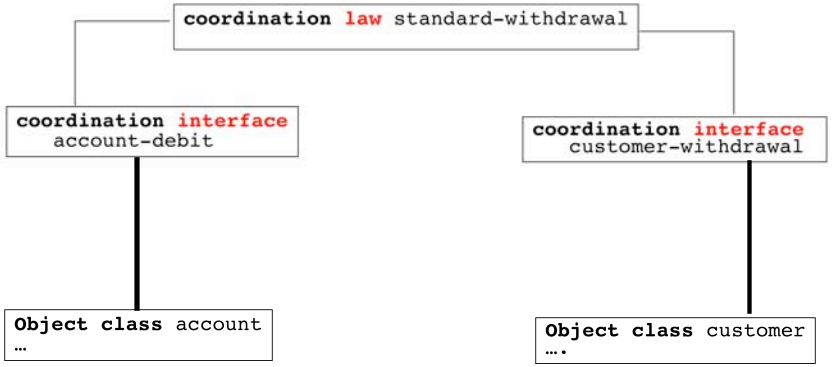

Example: openplan


```
coordination interface account-debit&credit
import types money
events        balance():money
services      debit(a:money);
              credit(a:money);
              balance():money
properties
       balance() after debit(a) is balance()-a;
       balance() after credit(a) is balance()+a
end interface
```

Instantiation



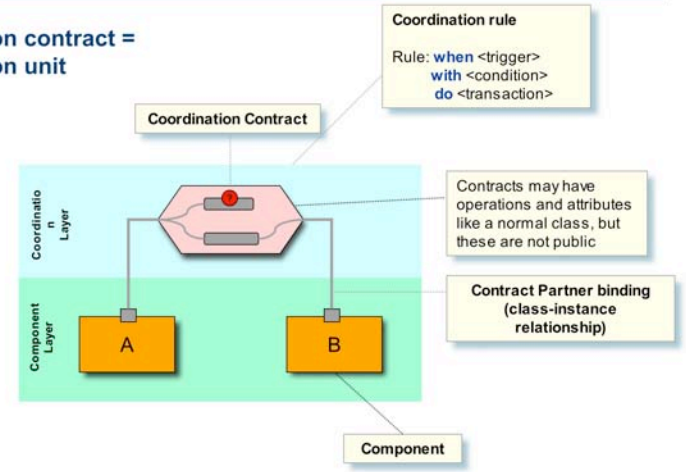
Instantiation



The result of the instantiation is a coordination contract

Design primitives for coordination

Coordination contract =
Coordination unit



Coordination contracts

- **Coordination contracts** instantiate coordination laws for particular kinds of components.
- For instance, in the case of **OO programming environments**,
  - **services** are provided through methods of the classes that instantiate the partners (coordination interfaces)
  - **events** are provided by object(instance) method calls or system/class method calls

Notation


```
coordination contract standard-withdrawal
  partners x : Account; y : Customer;
  coordination
            when    y ->> x.withdrawal(z)
            with    x.balance()>z and y.owns(x)
            do      call x.withdrawal(z)
end contract
```


Notation


```
coordination contract VIP package
  partners x : Account; y : Customer;
  attributes Credit: Integer;
  coordination
      when   y ->> x.withdrawal(z)
      with   x.balance()+Credit()>z and y.owns(x)
      do     x.withdrawal(z)
end contract
```


Intuitive semantics

- Instead of interacting with a mediator that delegates execution on the supplier, the client calls directly the supplier (the partners in the contract are not aware that they are being coordinated by a third party);
- The contract "*intercepts*" the call and superposes whatever forms of behaviour are prescribed;
- This means that it is not possible to bypass the coordination being imposed through the contract.

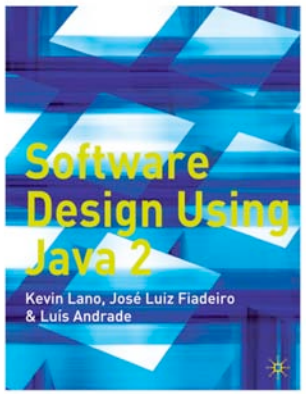
Another FAQ...

The answer to the FAQ

- Coordination Contracts reduce dependencies between objects
- Coordination Contracts place relationships together with associated interaction behavior.
- Coordination Contracts provide much more runtime flexibility
- Coordination Contracts are a simple, abstract and intuitive specification primitive
- Coordination Contracts avoid some complexity in models resulting from using design patterns and design pattern composition

Coordination runs in Java…

Coordination runs in Java…



The
Coordination Development Environment
can be downloaded from
www.atxsoftware.com/CDE

However

- **Mobility**

  A new factor of complexity in the development of software systems (Web, mobile communication) that cannot be relegated to lower level design.

  As components move across a network, the connectors in place may no longer ensure the required interactions.

However

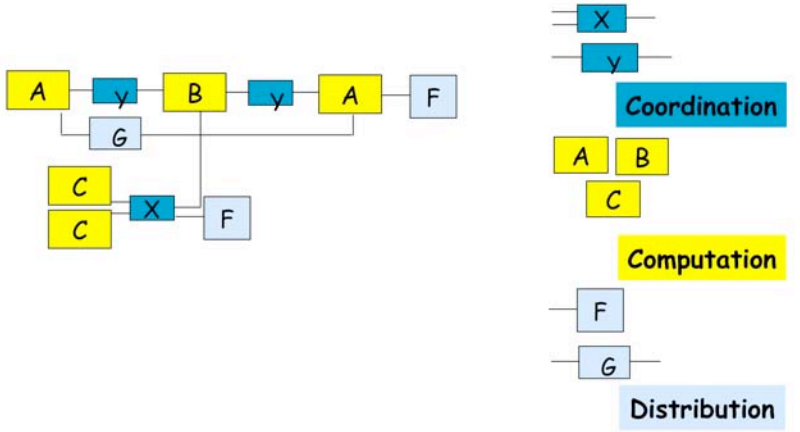
- **Business**

  Many businesses are selling the same services through different channels, each of which has specific features.

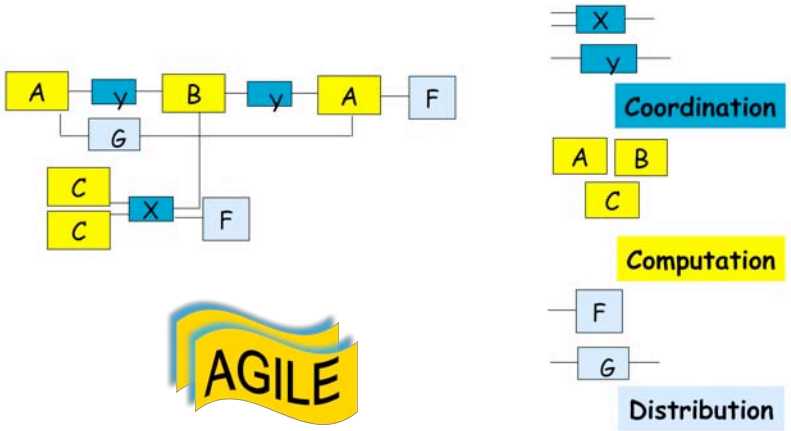
  For instance, withdrawing money is subject to location-specific rules in addition to the coordination-related ones. It is different to withdraw money at your local branch, another branch, an ATM…

  The system should self-adapt to changes of location without interfering with the coordination business rules.

Architectures for Mobility



Architectures for Mobility



Location laws


```
location law ATM-withdrawal
locations    atm:ATMW-LI; bank:BANKW-LI
rules  when  atm.withdrawal(n) and BT(atm,bank)
       with  n≤bank.maxatm() and n≤atm.cash()
       do    atm.give(n)
       when  atm.withdrawal(n) and not BT(atm,bank)
                      and REACH(atm,bank)
       let   N=min(atm.default(),n) in
       with  N≤atm.cash()
       do    atm.give(N)
       mv    bank.internal(N,atn.acco())
end law
```

Be in touch


```
rules  when  atm.withdrawal(n) and BT(atm,bank)
       with  n≤bank.maxatm() and n≤atm.cash()
       do    atm.give(n)
```

- **BT** indicates whether the two locations are **"in touch"** meaning that they can communicate and synchronise actions at both locations;
- If they are, coordination laws apply to the partners that are located there. The guards (*with* conditions) of coordination and location rules apply, and the reactions of both are performed atomically.

Reach


```
when  atm.withdrawal(n) and not BT(atm,bank)
                        and REACH(atm,bank)
      let   N=min(atm.default(),n) in
      with  N≤atm.cash()
      do    atm.give(N)
      mv    bank.internal(N,atm.acco())
```

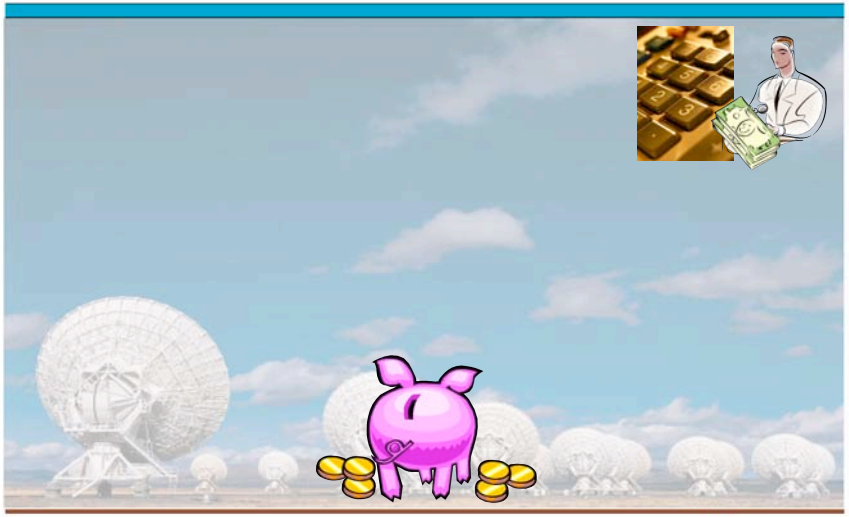
- **REACH** indicates that one location can be **"reached"** from the other, meaning that services can move across;
- **mv** indicates that the service is sent for execution at the other location.

Location interfaces

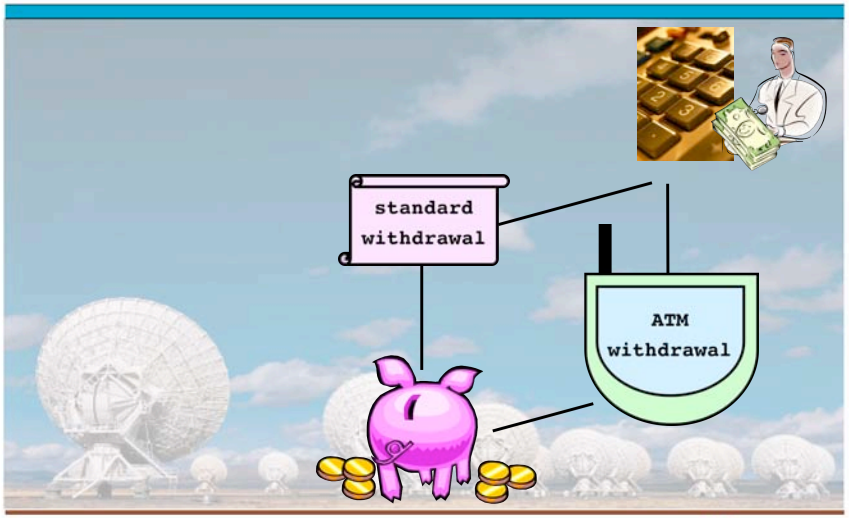

```
location interface BANKW-LI
location type    BANK
services      internal(n:money,a:ACCOUNT)
              maxatm():money
end interface
```

```
location interface ATMW-LI
location type    ATM
services      default(),cash():money,acco():ACCOUNT
              give(n:money) post cash()=oldcash()-n
events        withdraw(n:money)
end interface
```

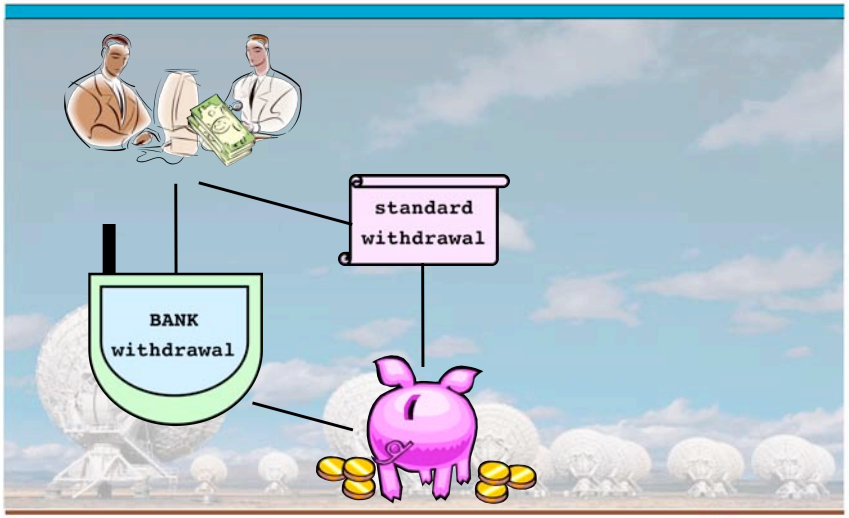

Instantiation



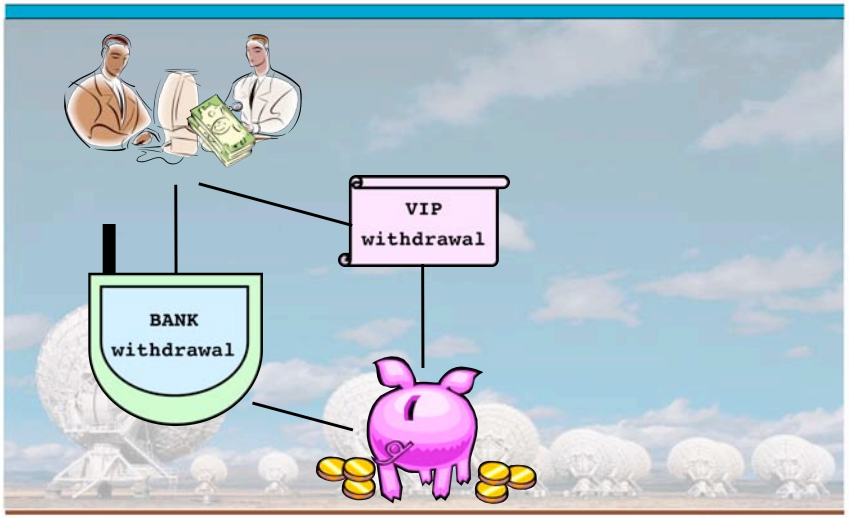
Instantiation



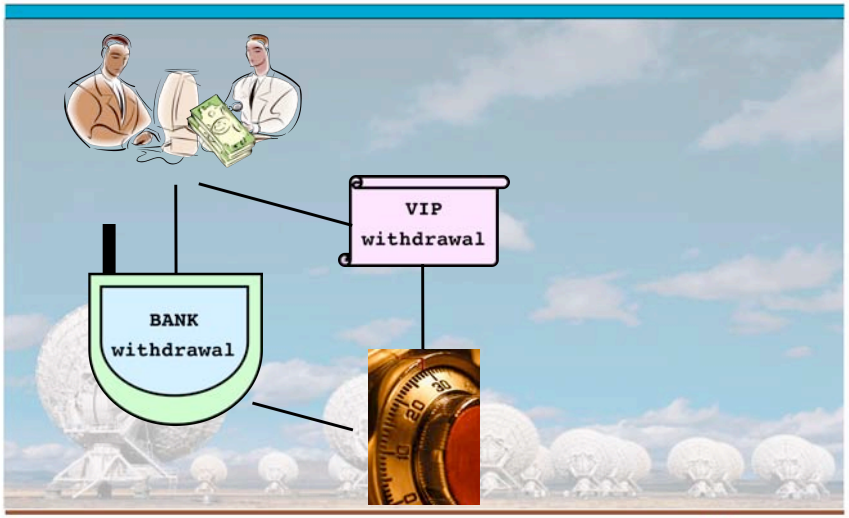
Instantiation



Instantiation



Instantiation



Other Work

- Semantic primitives for location/mobility in the UML
- Operational semantics in KLAIM
- Reconfiguration with Hypergraph Rewriting
- Refinement in Tile Logic
- Model-checking
- Architectures in CASL

Other Work

68

- Semantic primitives for location/mobility in the UML
- Operational semantics in KLAIM
- Reconfiguration with Hypergraph Rewriting
- Refinement in Tile Logic
- Model-checking
- Architectures in CASL

SEFM'04 University of Leicester

Other Work

- Mathematical semantics in MMONITY
- Requirements with Problem Frames
- Domain specific extensions (embedded, collaborative, …)
- Context-awareness

SEFM'04 University of Leicester

Join us…

www.cs.le.ac.uk/SoftSD

Postgraduate (MSc,PhD)
Postdoc
Sabbatical leaves
Research visits

SEFM'04 University of Leicester

Join us...

www.cs.le.ac.uk/SoftSD

Postgraduate (MSc,PhD)
Postdoc
Sabbatical leaves
Research visits

IST-2001-32747
Architectures for
Mobility

WEE-NET
Network of Excellence in Web Engineering

αlfa
Latin America-Academic training

RELEASE

Scientific Network
on Software Evolution

Ad-hoc
Web Applications
Nuffield Foundation