OO Design Principles & Metrics

Jason Gorman

OO Design Goals

- Make software easier to change when we *want* to
  - We might *want* to change a class or package to add new functionality, change business rules or improve the design
  - We might *have* to change a class or package because of a change to another class or package it depends on (e.g., a change to a method signature)
  - Manage dependencies between classes and packages of classes to minimise impact of change on other parts of the software
  - Minimise reasons that modules or packages might be forced to change because of a change in a module or package it depends upon

parlez|uml

Cat Poetry

Cat Poetry

Sohsasfs gshsdgof dfidffvv08vwowv
Sfvsfv fogodgfgosdf fgvdfgodfvn awtre
Qwqwwdasvs srvouowvrv sfdifwifd
Sdvsvwvfvsvspvsvs
Sosd ty bicei rohrtgneoesvs woo sdn
Vovwvodosv osdssos kerrfwdca
Oooodcnoos wrfrohfowrofh woffwofwo
Eoegeor cdscseuowfuhf cocecncnc

Cat Poetry Between Well-defined Interfaces

I wandered lonely as a clowd
Sohsasfs gshsdgof dfidffvv08vwowv
Sfvsfv fogodgfgosdf fgvdfgodfvn awtrev
Qwqwwdasvs srvouowvrv sfdifwifd
Sdvsvwvfvsvspvsvs
Sosd ty bicei rohrtgneoesvs woo sdn wp
Vovwvodosv osdssos kerrfwdca
Oooodcnoos wrfrohfowrofh woffwofwoeohfow
Eoegeor cdscseuowfuhf cocecncnc
Continuous as the stars that shine

A Hazelnut In Every Bite…

- Much of OO design is about managing dependencies
- It is very difficult to write OO code without creating a dependency on something
- => <u>99.9% of lines of code contain at least one significant design decision</u>
- => <u>Anyone who writes a line of code is defining the design</u>

parlez|uml

Every Programmer Is A Designer!

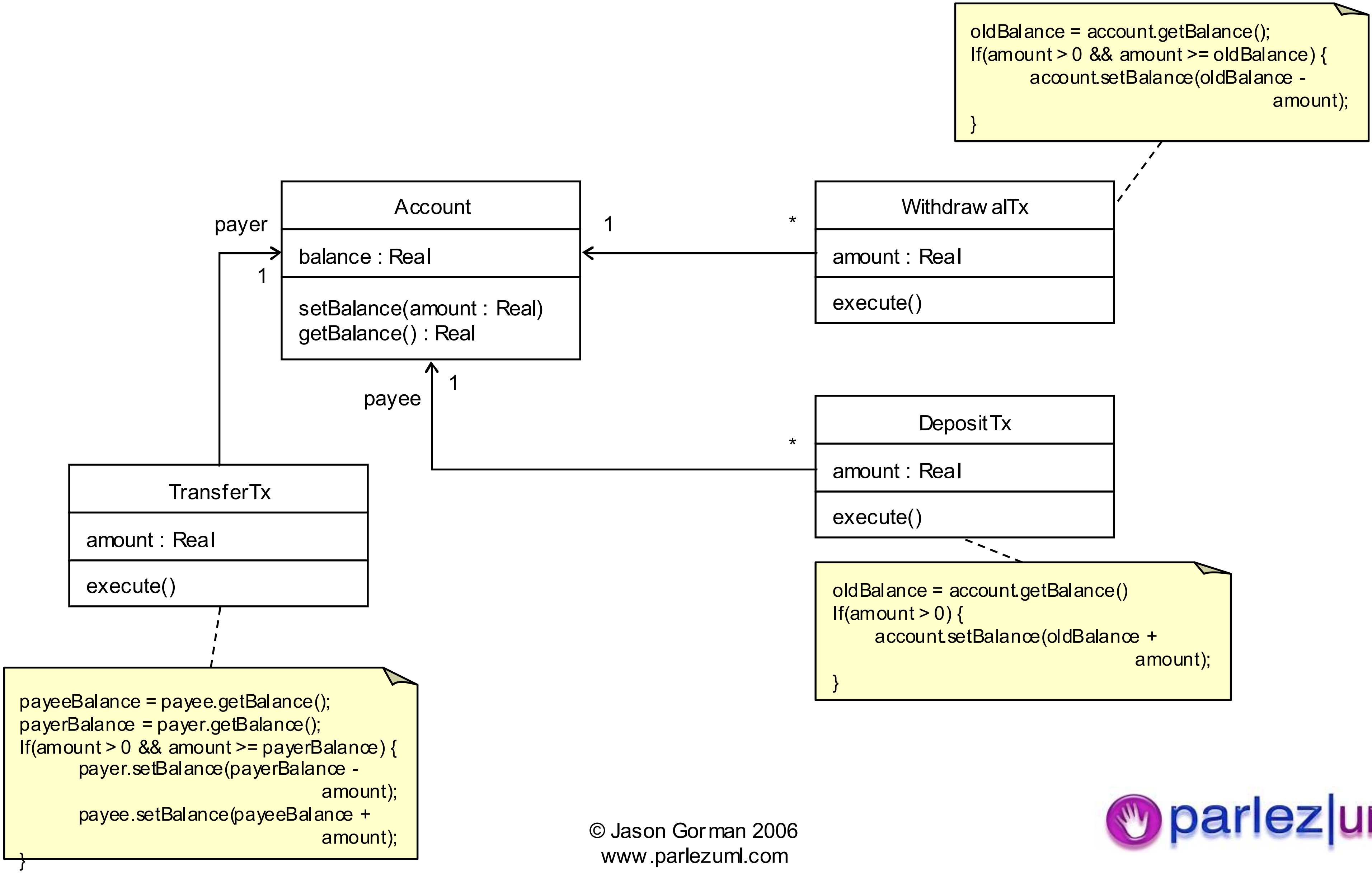
Class Design

- Class Cohesion
- Open-Closed
- Single Responsibility
- Interface Segregation
- Dependency Inversion
- Liskov Substitution
- Law of Demeter
- Reused Abstractions

Class Cohesion

Reasoning:

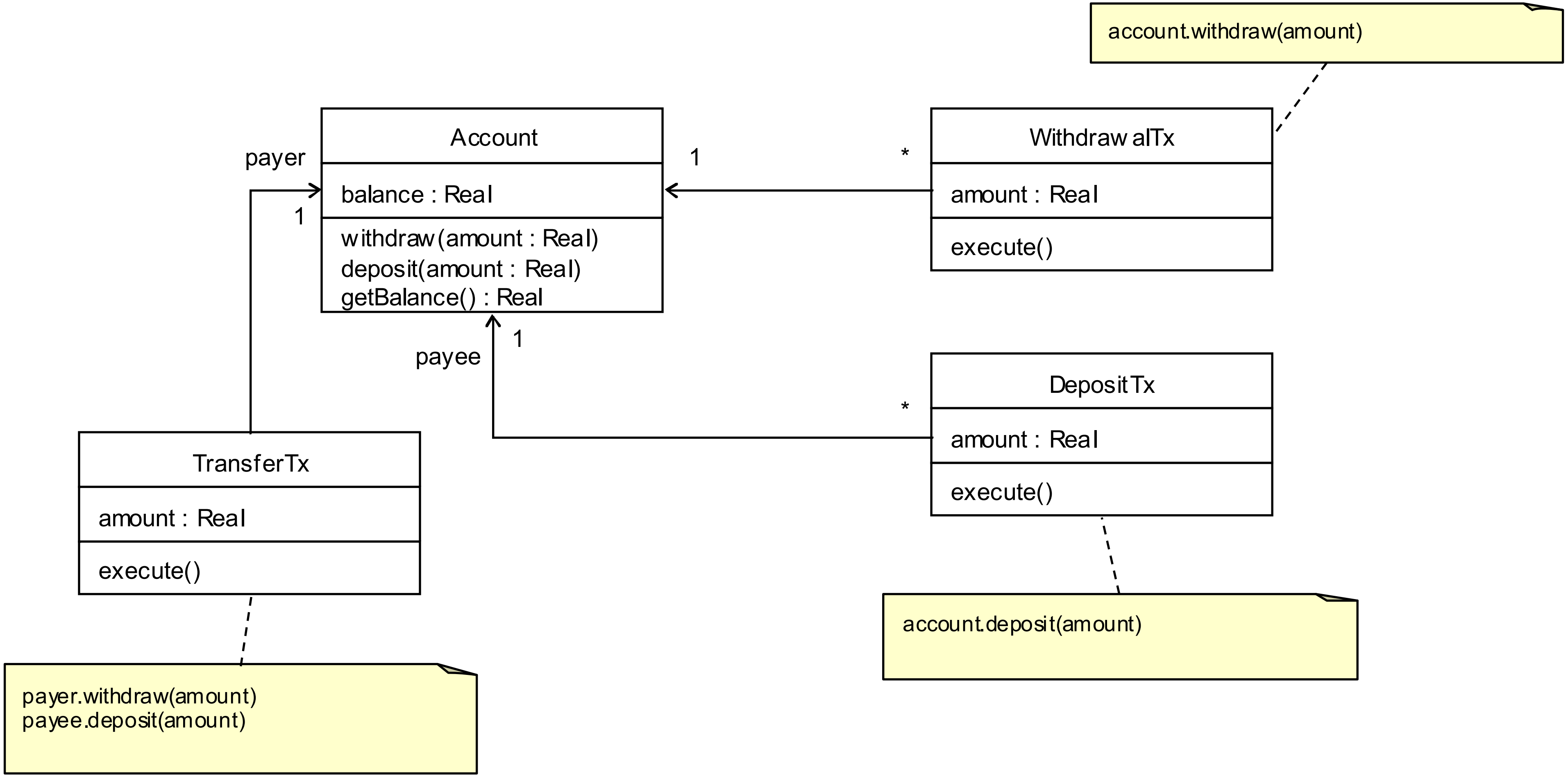
Class design should reduce the need to edit multiple classes when making changes to application logic. A fundamental goal of OO design is to place the behaviour (methods) as close to the data they operate on (attributes) as possible, so that changes are less likely to propagate across multiple classes



Class Cohesion - Refactored

Reasoning:

Class design should reduce the need to edit multiple classes when making changes to application logic. A fundamental goal of OO design is to place the behaviour (methods) as close to the data they operate on (attributes), so that changes are less likely to propagate across multiple classes



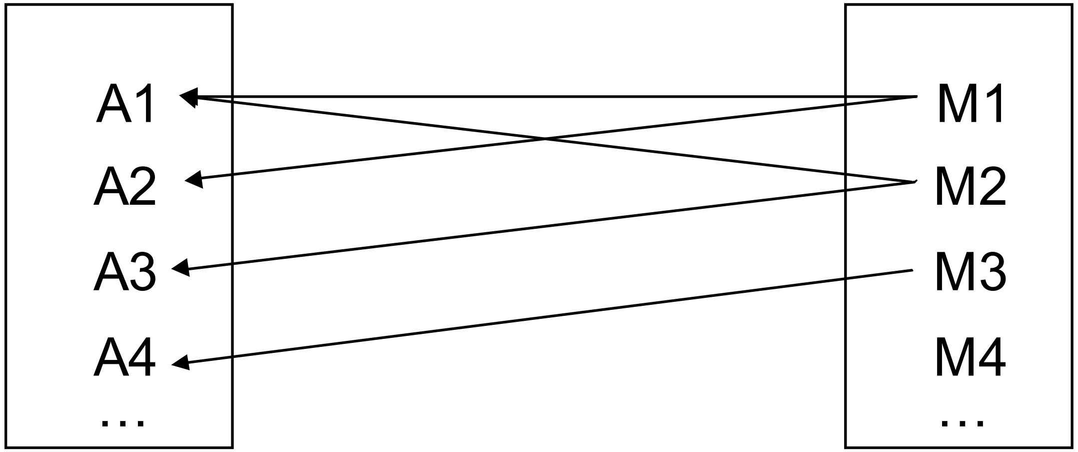
Class Cohesion Metrics

Lack of Cohesion of Methods (LCOM) = $\frac{((\sum R(A))/A) - M}{1 - M}$

"The average number of methods that access each attribute"

Each attribute A is accessed by R(A) methods

automation possible



Class has A attributes

Class has M methods

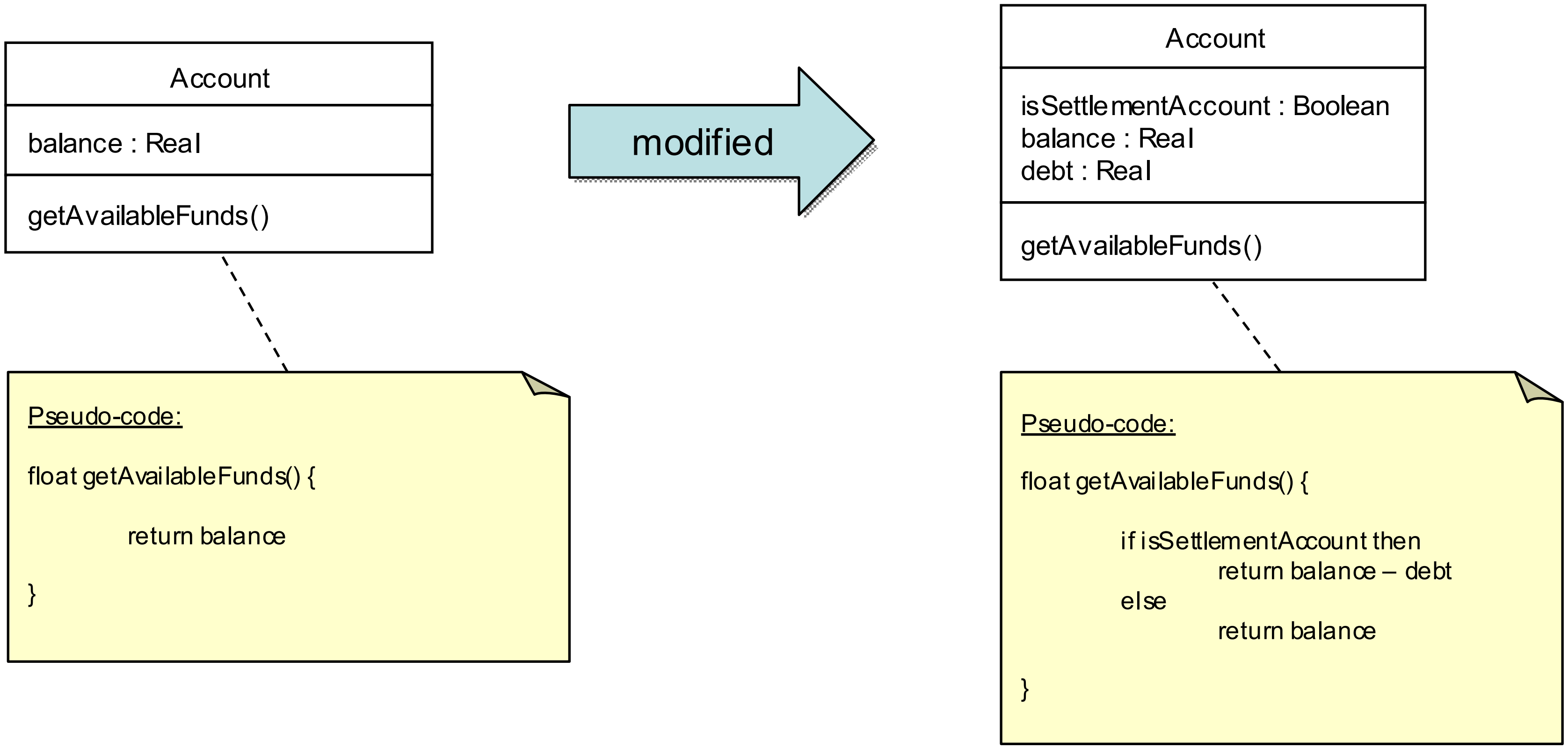
TIP: An reference to an associated object is equivalent to an attribute

Open-Closed

Reasoning:

Once a class is tested and working, modifying its code can introduce new bugs. We avoid this by extending the class, leaving its code unchanged, to add new behaviour.

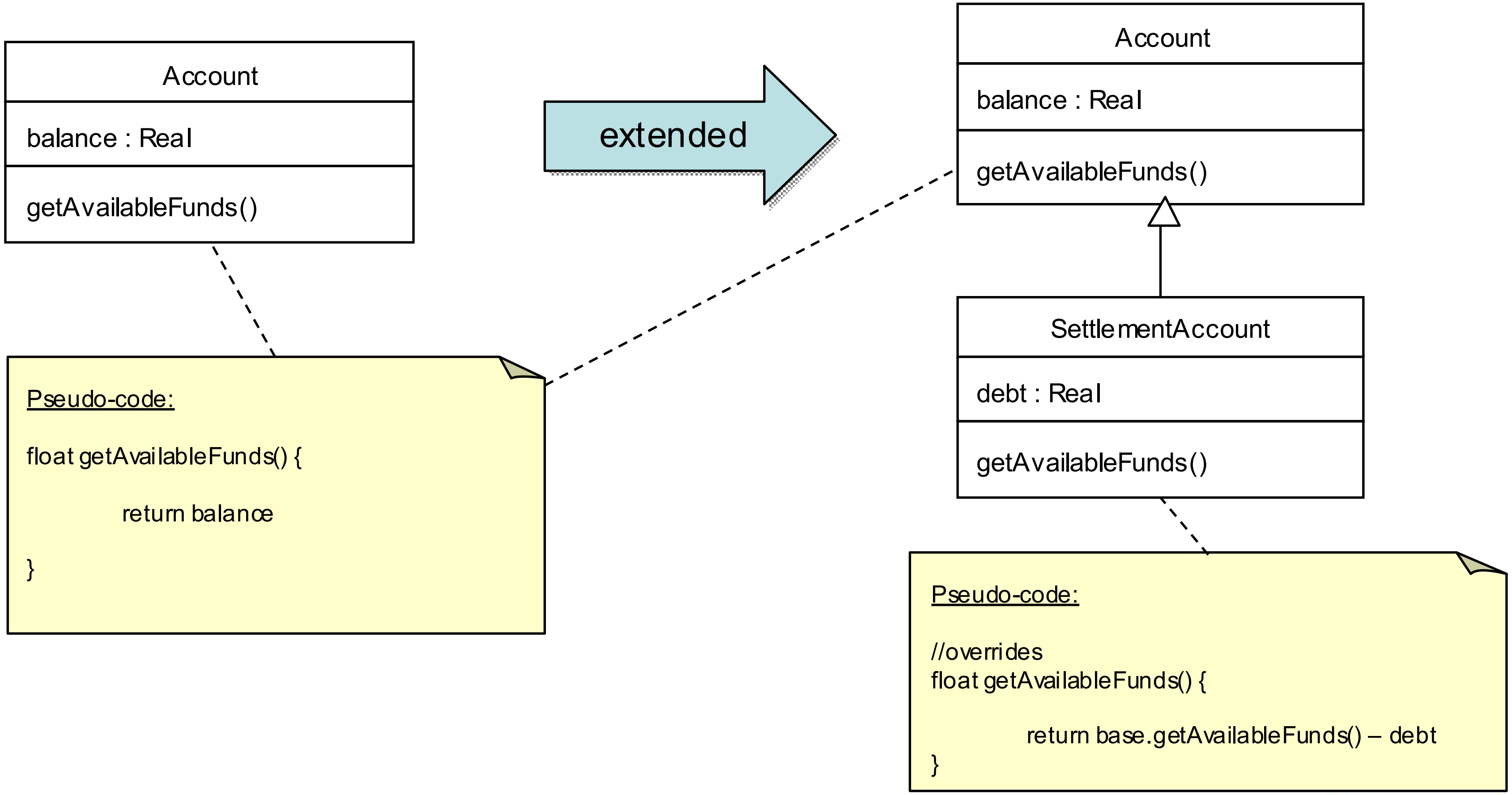
Classes should be open to extension, but closed to modification



Open-Closed - Refactored

Reasoning:

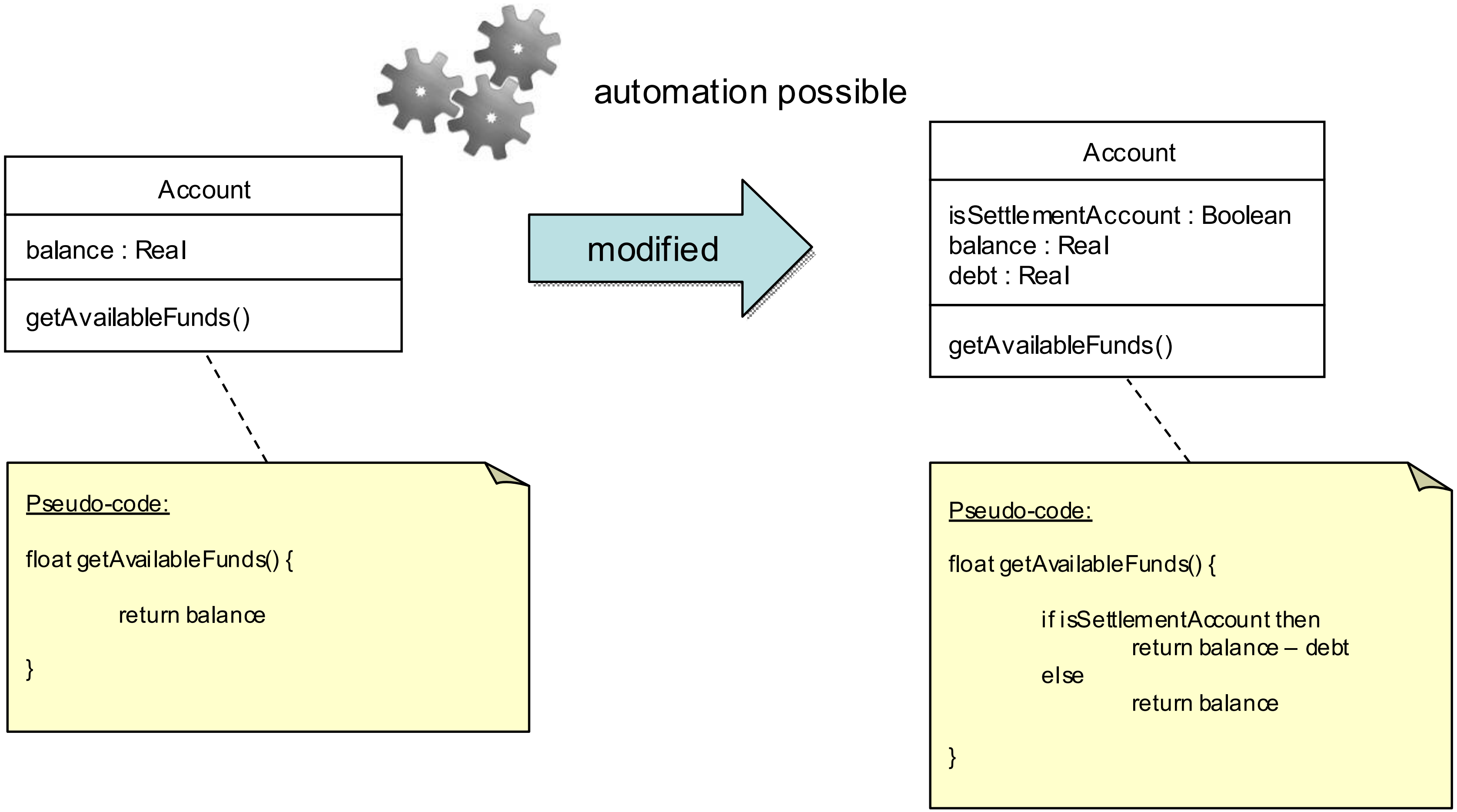
Once a class is tested and working, modifying its code can introduce new bugs. We avoid this by extending the class, leaving its code unchanged, to add new behaviour. Classes should be open to extension, but closed to modification



Open-Closed - Metrics

⇒Per successful check-in

⇒ classes extended and not modified / classes extended and/or modified

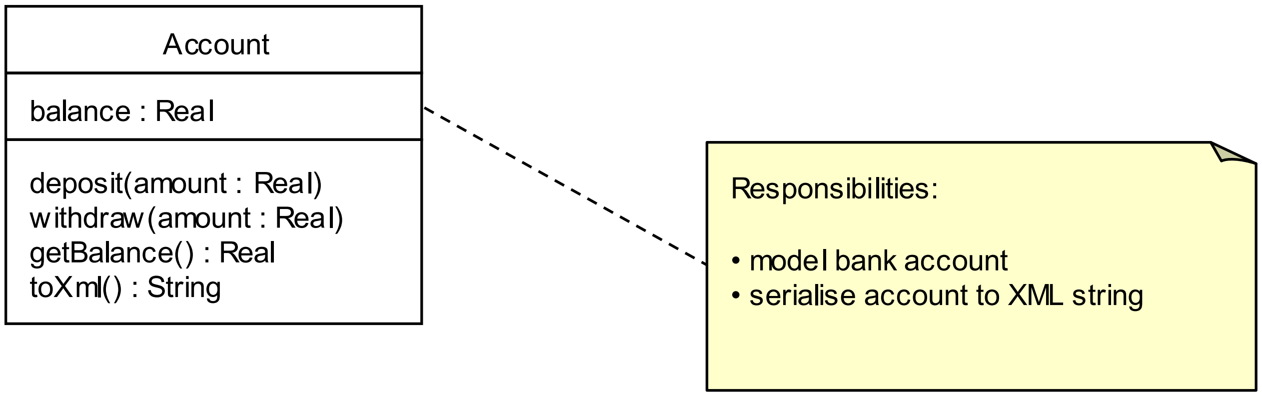


parlez|uml

Single Responsibility

<u>Reasoning:</u>

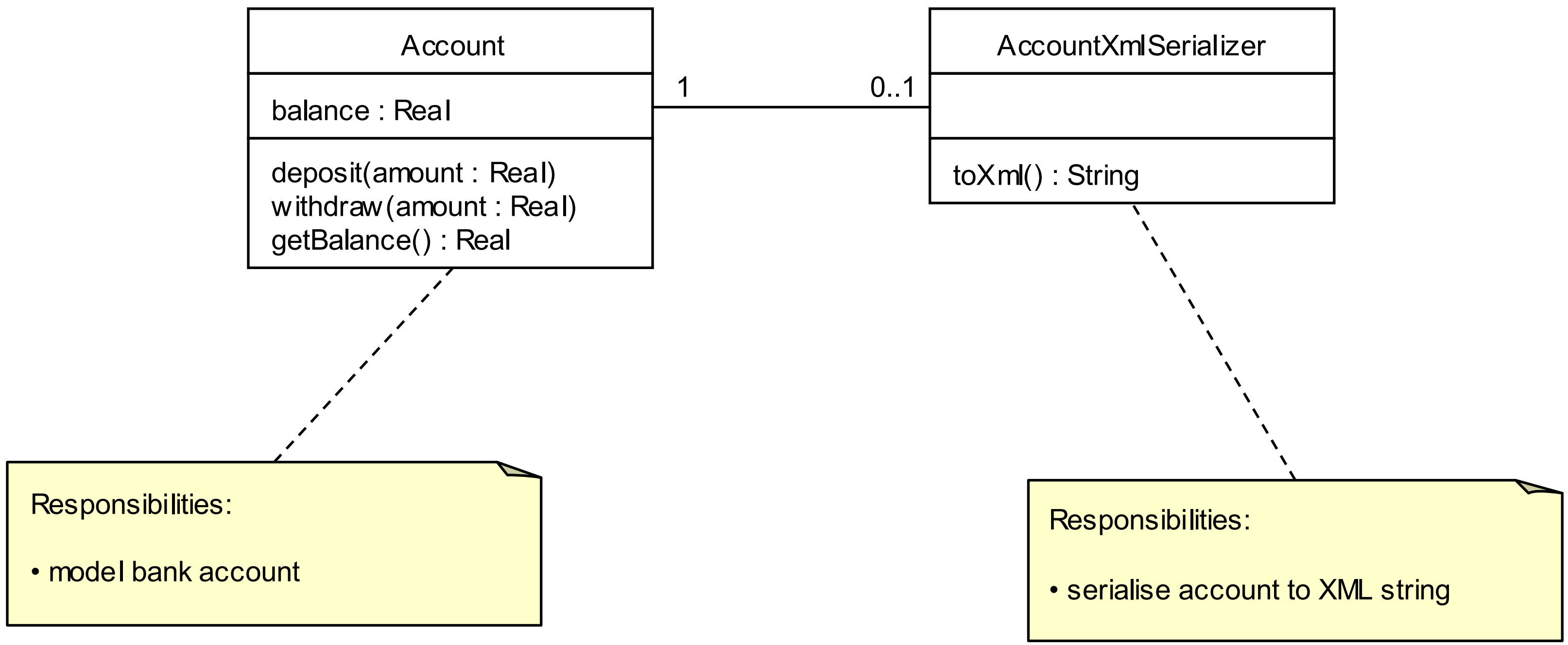
Changing code in a tested class can introduce new bugs. We seek to minimise the reasons why a class might need to change. The more different things a class does, the more reasons it might have to change.



<u>Two</u> reasons why this class might need to change

- changes to domain logic
- changes to XML format

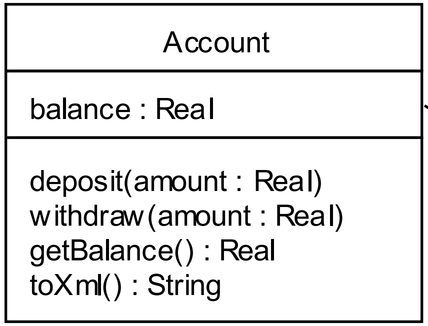
Single Responsibility - Refactored



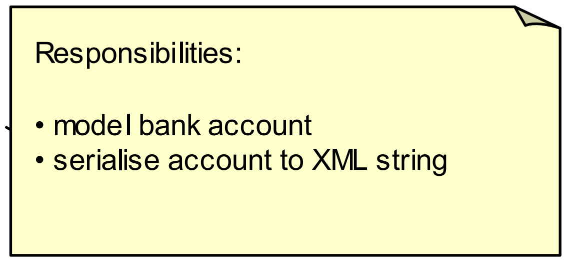
parlez|uml

Single Responsibility - Metrics

=> responsibilities / class



by code/design inspection & statistical sample



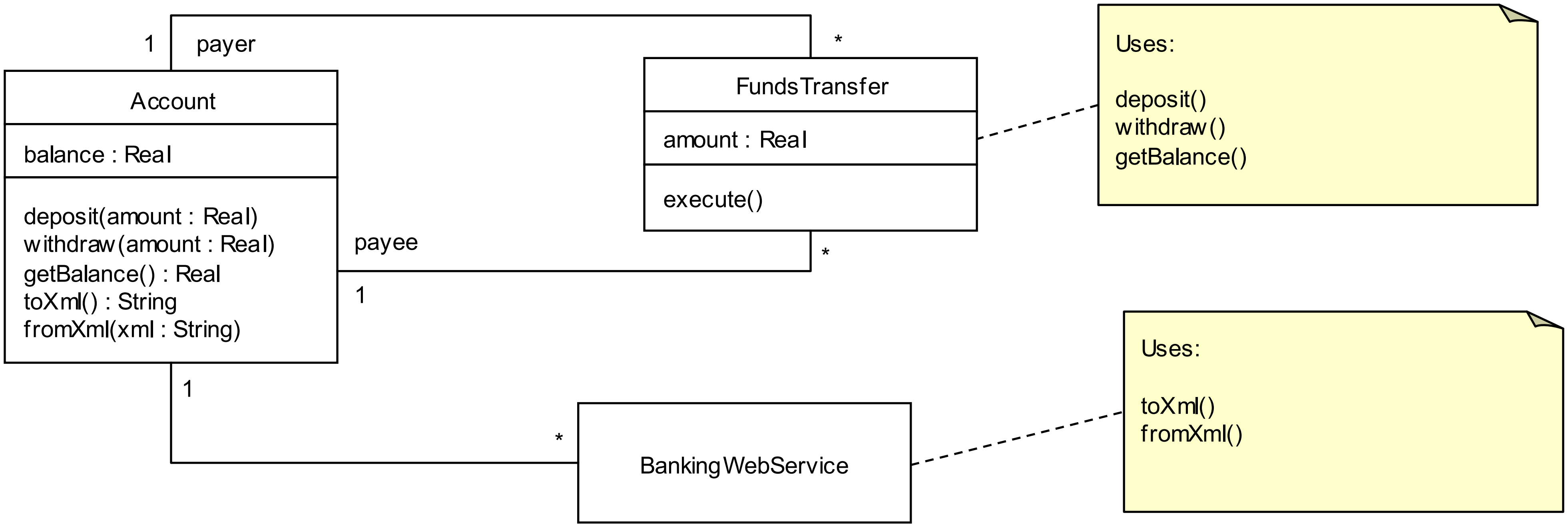
Q: What is a “responsibility”?

parlez|uml

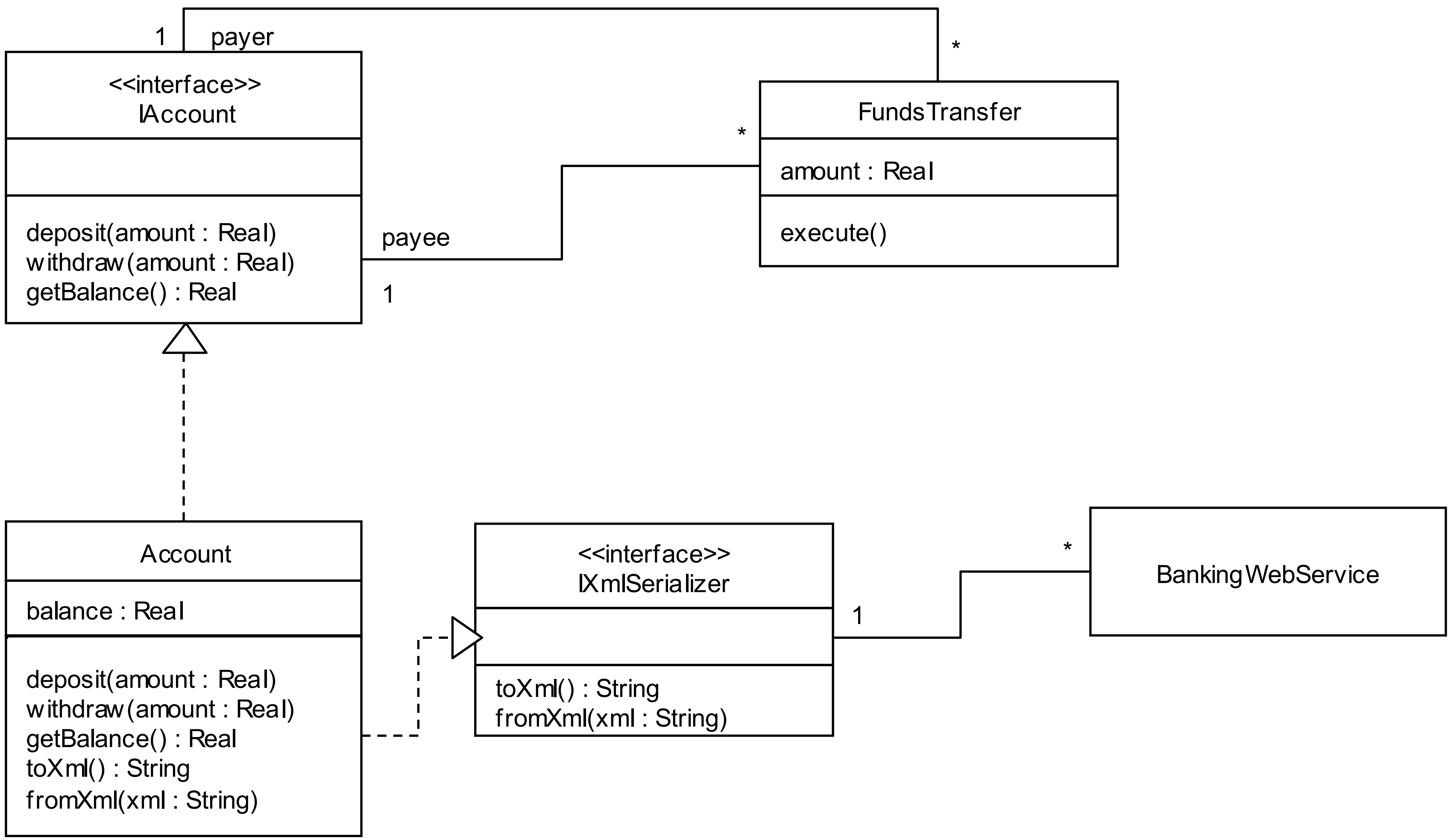
Interface Segregation

Reasoning:

If different clients depend on different methods of the same class, then a change to one method might require a recompile and redeployment of other clients who use different methods. Creating several client-specific interfaces, one for each type of client, with the methods that type of client requires, reduces this problem significantly.



Interface Segregation - Refactored

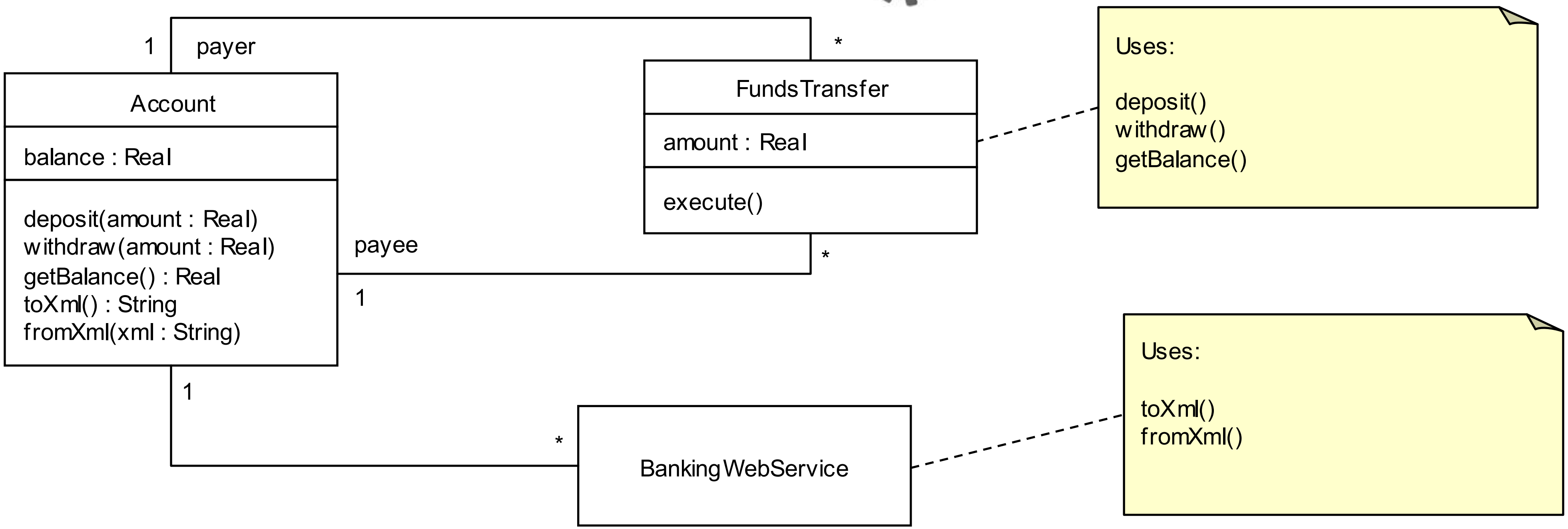


Interface Segregation - Metrics

If type T exposes N methods, and client C uses n of them, then T's interface is n/N specific with respect to C.

=> Average n/N for all clients of T

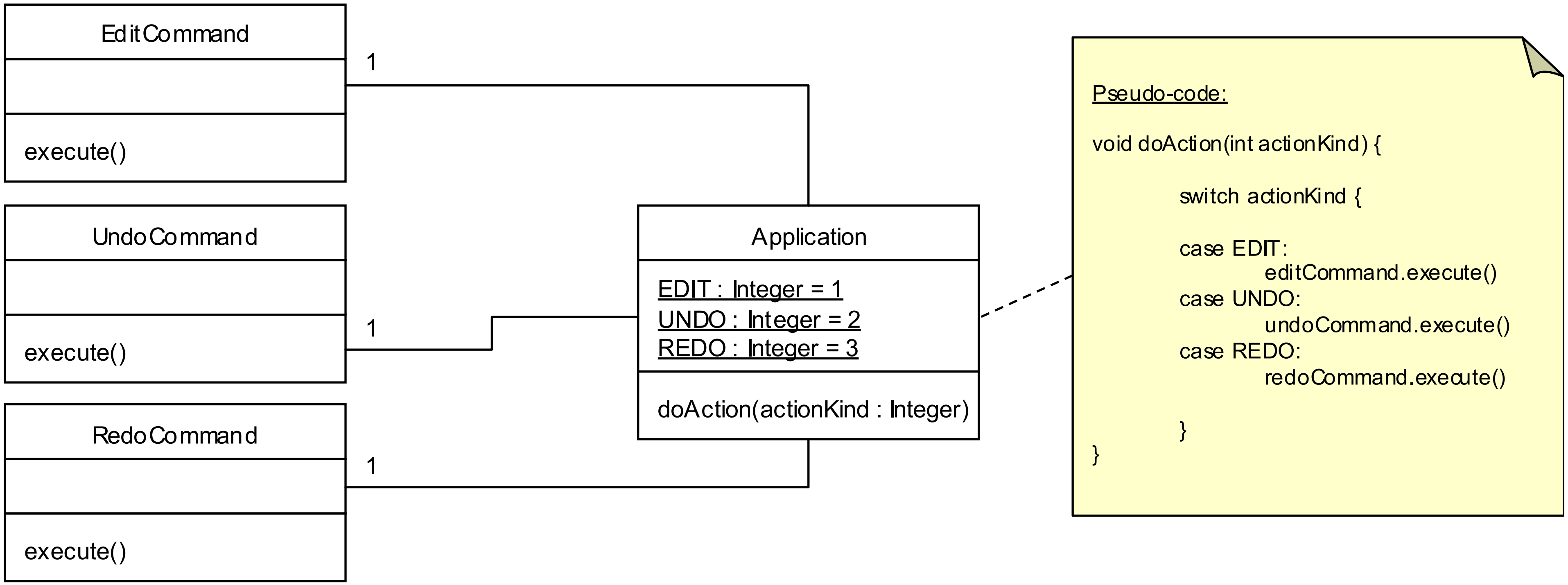
automation possible



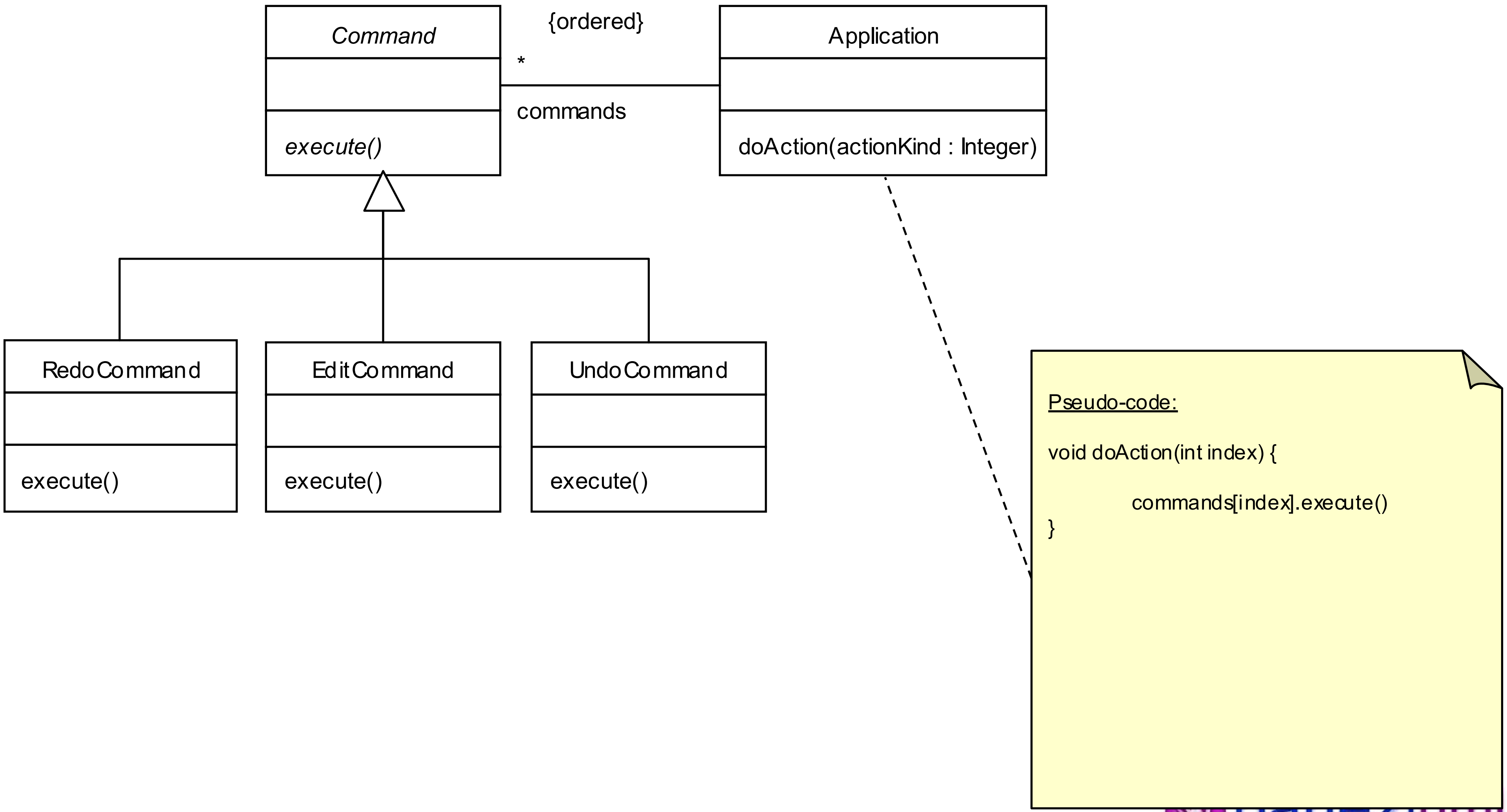
Dependency Inversion

Reasoning:

Much of the duplication in code comes from client objects knowing about all sorts of specialised suppliers, that – from the client's perspective – do similar things but in different ways. Polymorphism is a powerful mechanism that underpins OO design. It allows us to bind to an abstraction, and then we don't need to know what concrete classes we are collaborating with. This makes it much easier to plug in new components with no need to change the client code.



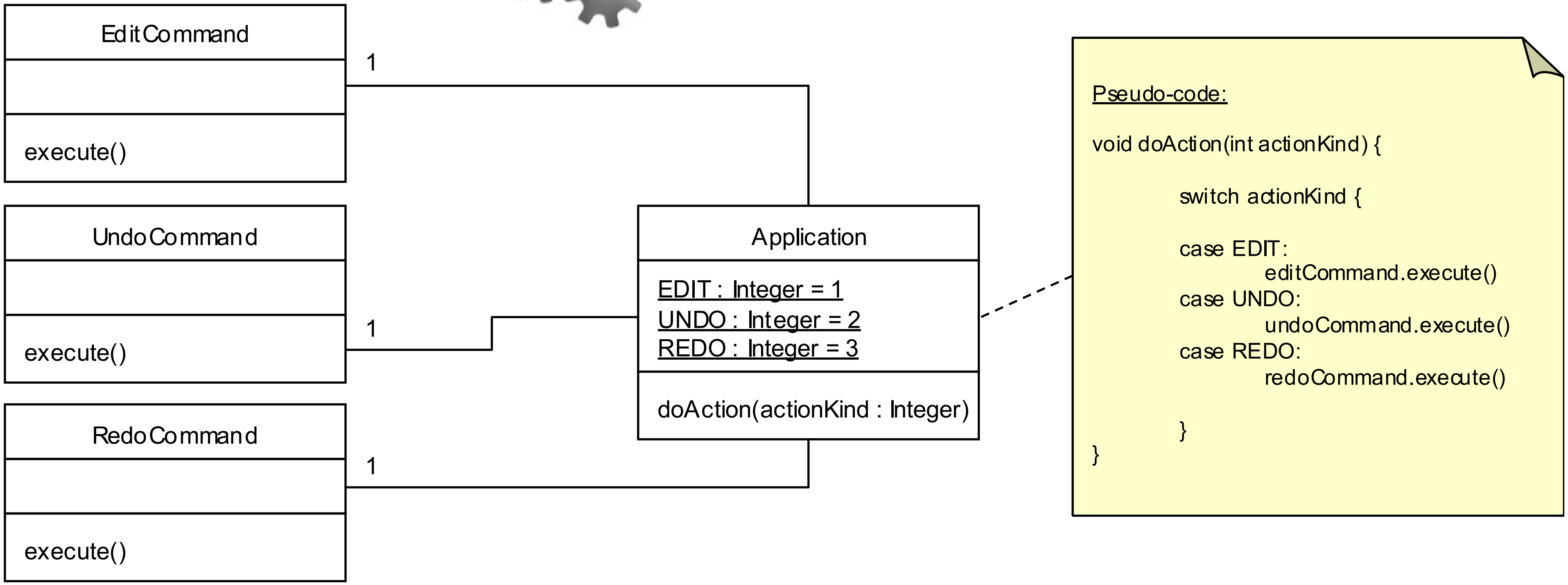
Dependency Inversion - Refactored



Dependency Inversion - Metrics

=> dependencies on abstractions / total dependencies

automation possible

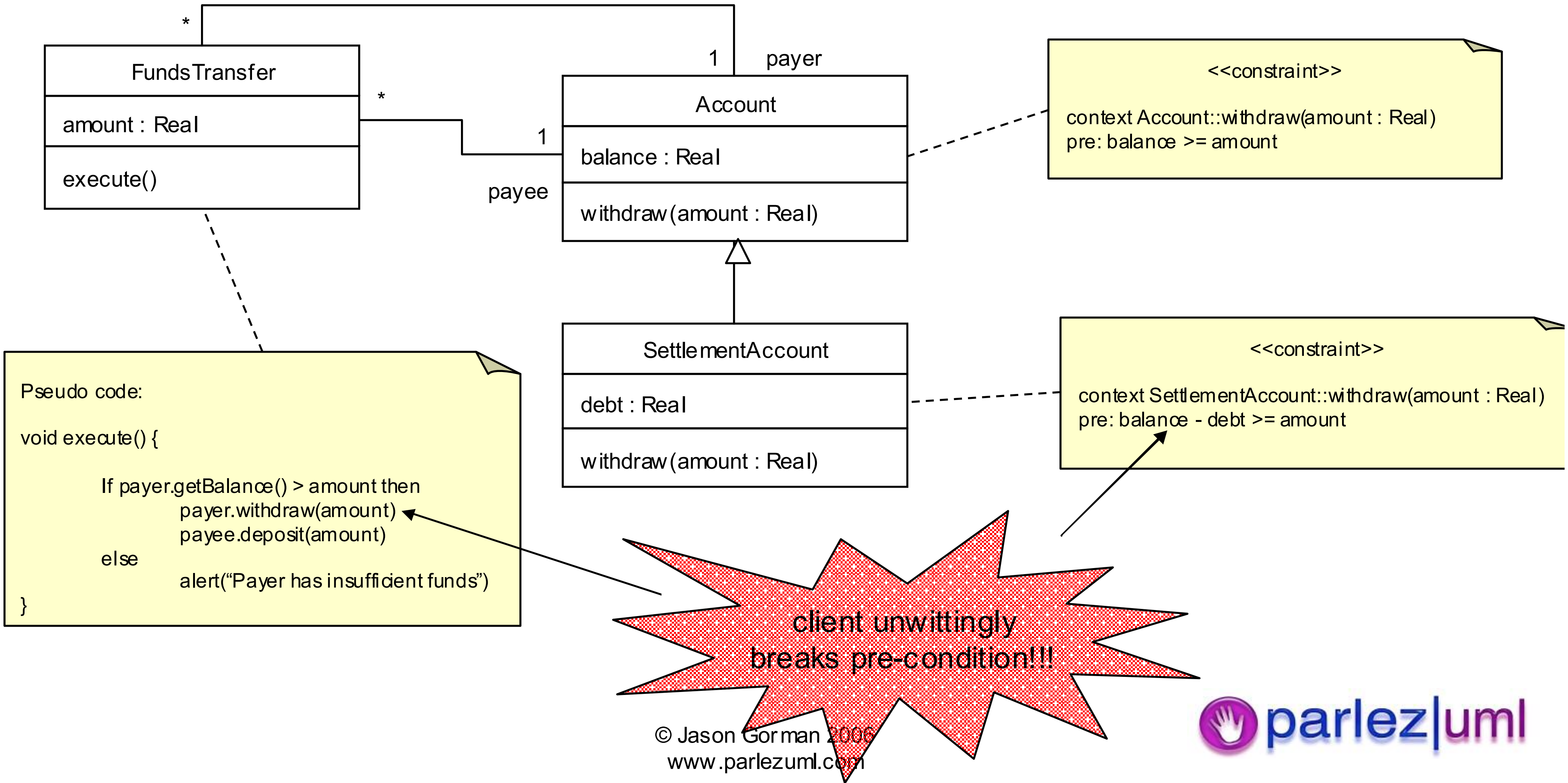


parlez|uml

Liskov Substitution

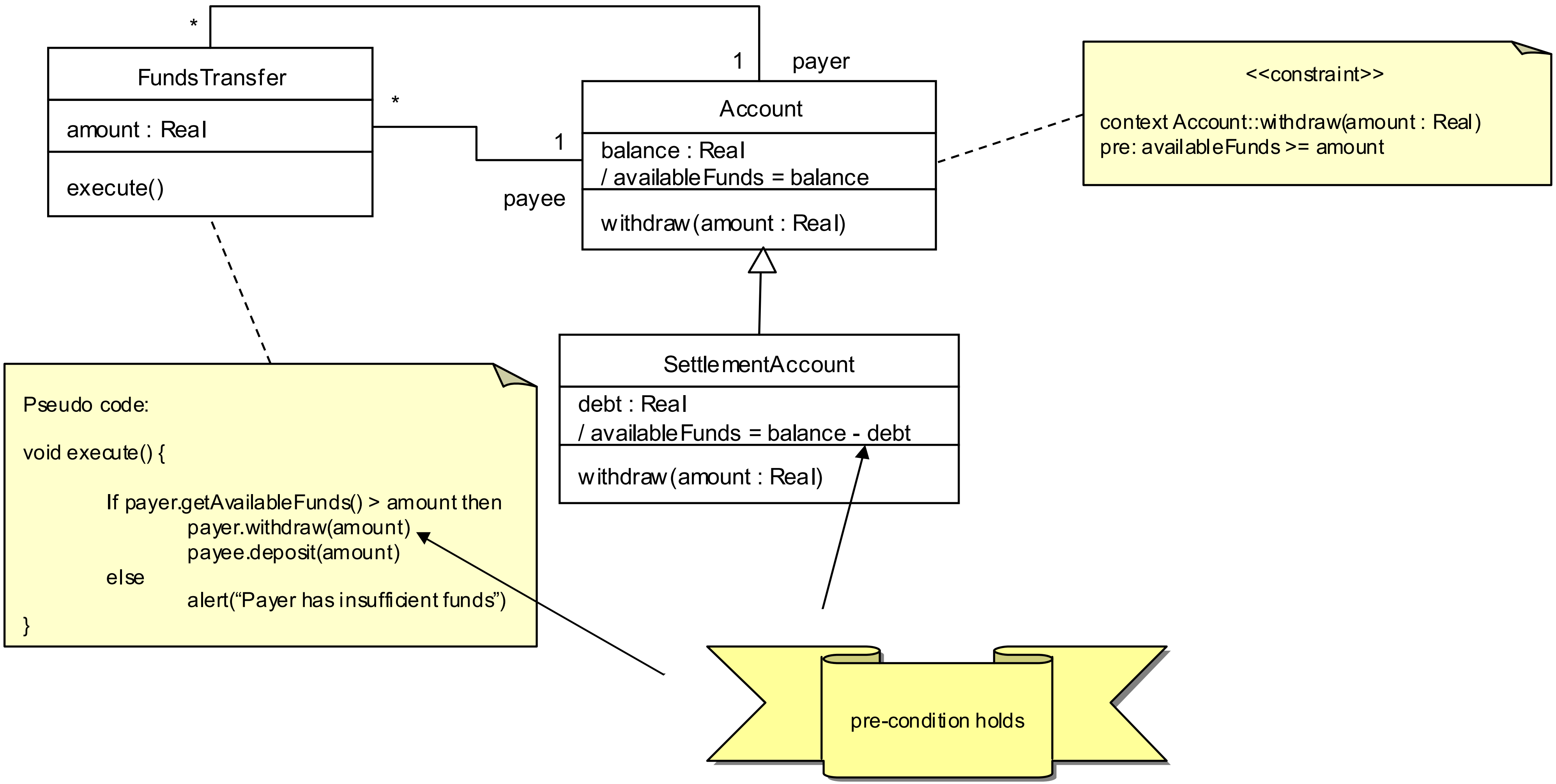
Reasoning:

Dynamic polymorphism is a powerful mechanism that allows us to invert dependencies, reducing duplication and making change much easier. All OO design principles depend upon polymorphism, but we must ensure that any type can be substituted for any of its subtypes at run-time without having any adverse effect on the client. Subtypes must obey all of the rules that apply to their super-types – pre-conditions for calling methods, post-conditions of methods called, and invariants that always apply between method calls.



parlez|uml

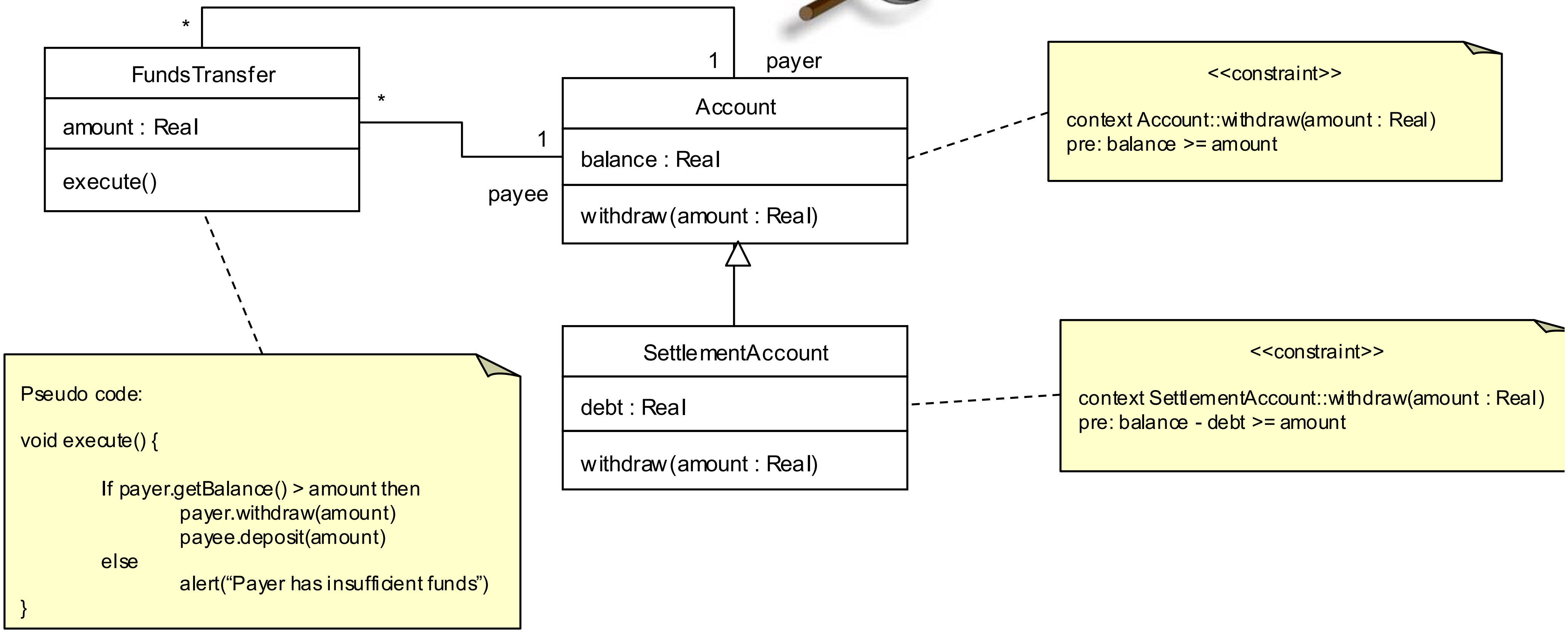
Liskov Substitution - Refactored



Liskov Substitution - Metrics

=> every class should pass all of the unit tests for all of its super-types

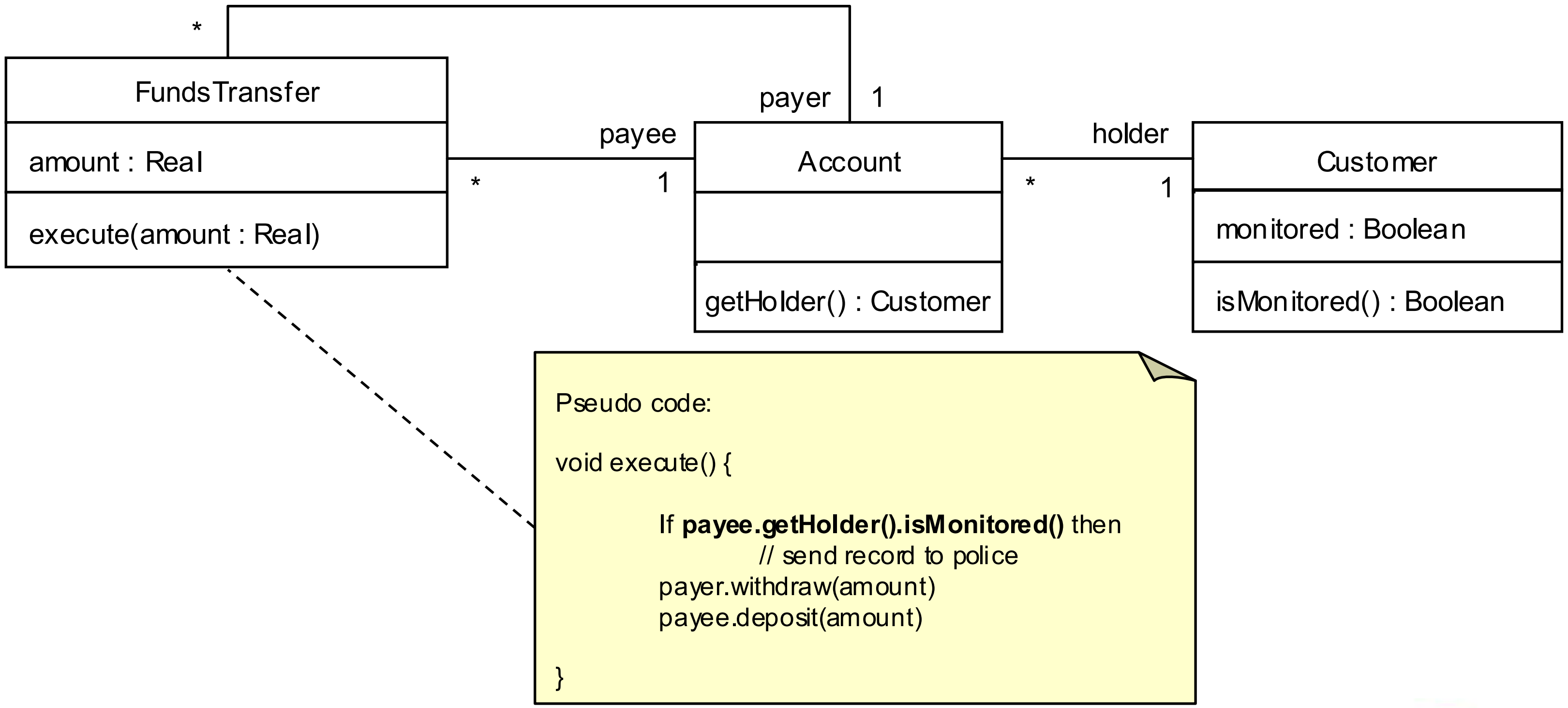
by code/design inspection &
statistical sample



Law of Demeter

<u>Reasoning:</u>

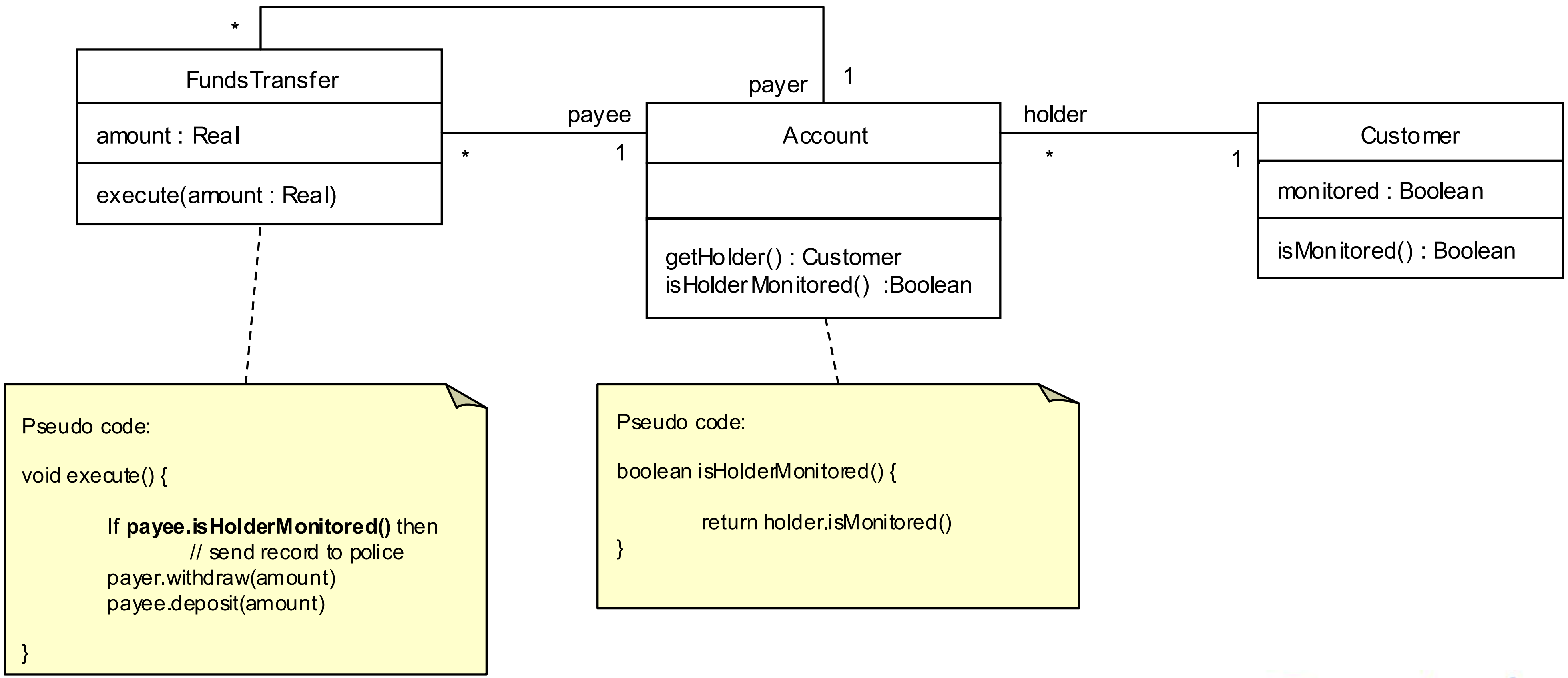
Objects should only collaborate with their nearest neighbours – the less they depend on the interfaces of "friends of a friend", the less reasons they might have to have to change. This means avoiding long navigations and deferring knowledge of interactions with objects that aren't directly related to your nearest neighbours.



Law of Demeter - Refactored

Reasoning:

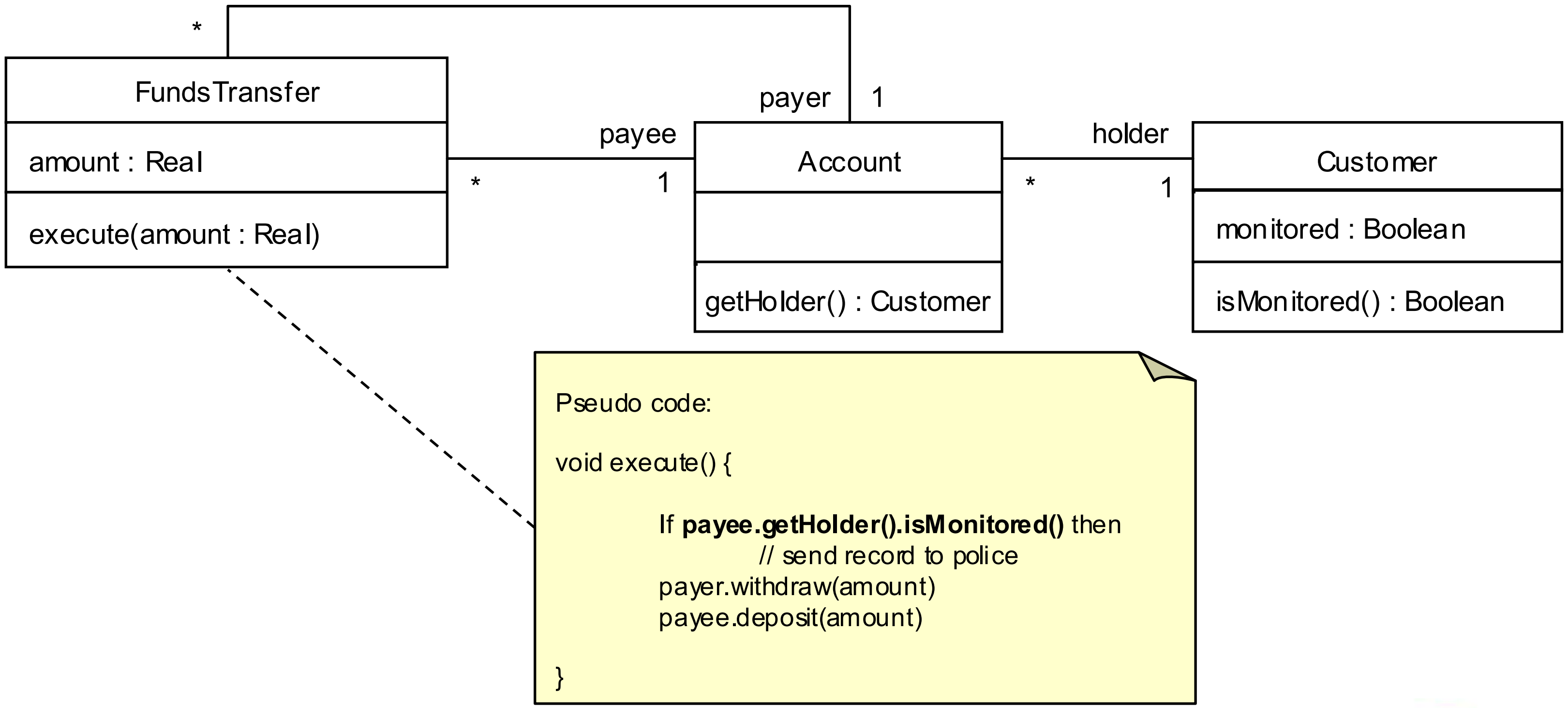
Objects should only collaborate with their nearest neighbours – the less they depend on the interfaces of “friends of a friend”, the less reasons they might have to have to change. This means avoiding long navigations and deferring knowledge of interactions with objects that aren’t directly related to your nearest neighbours.



Law of Demeter - Metrics

⇒ average depth of navigation

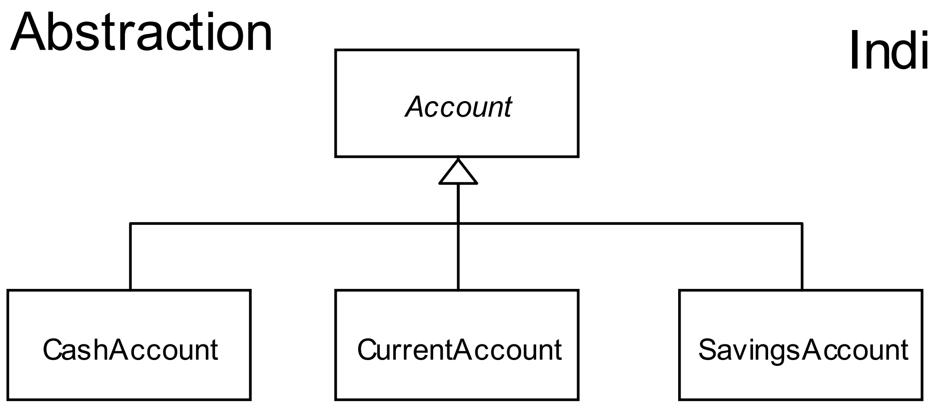
automation possible

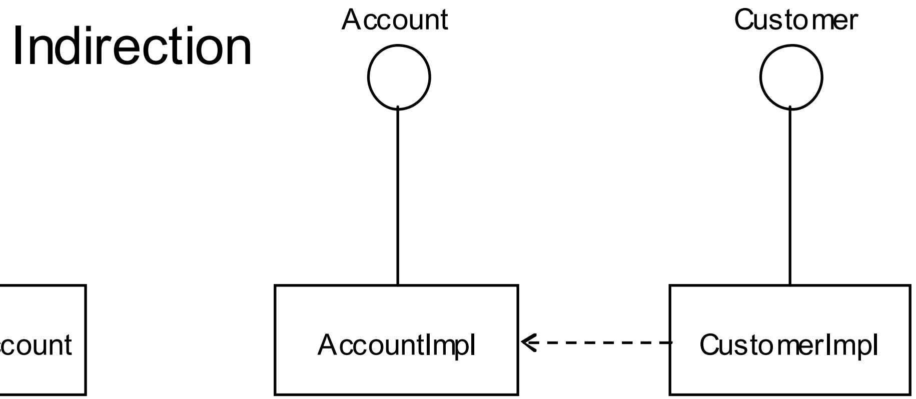


Reused Abstractions

Reasoning:
In test-driven development, abstractions are discovered by looking for similarities between classes or interfaces. Designers should distinguish between bone fide abstractions and indirection. A bone fide abstraction incorporates shared elements of two or more types into a single, shared abstraction to which both types conform. When we create arbitrary abstractions (e.g., interfaces for mock object tests), we create an extra maintenance burden with no pay off in term so removal of duplication.

In simpler terms, abstractions should be extended or implemented by more than one class.





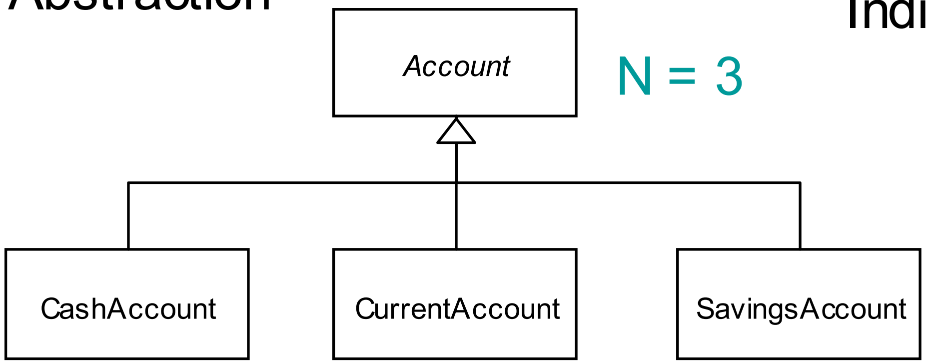
parlez|uml

Reused Abstractions - Metrics

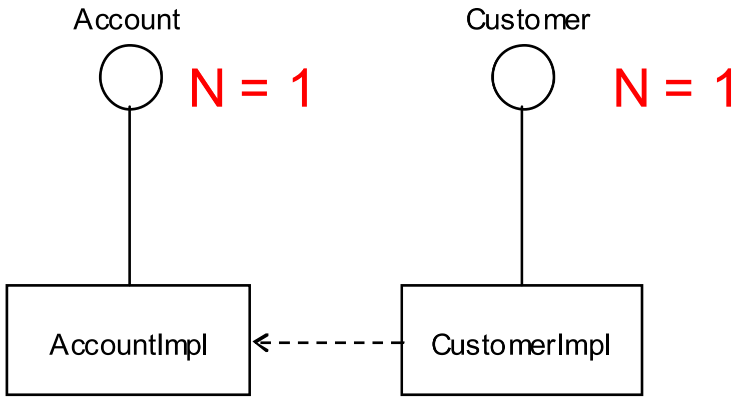
For an abstract class or interface T, which is extended or implemented by N classes or interfaces, such that **N > 1**

automation possible

Abstraction



Indirection



Package Design

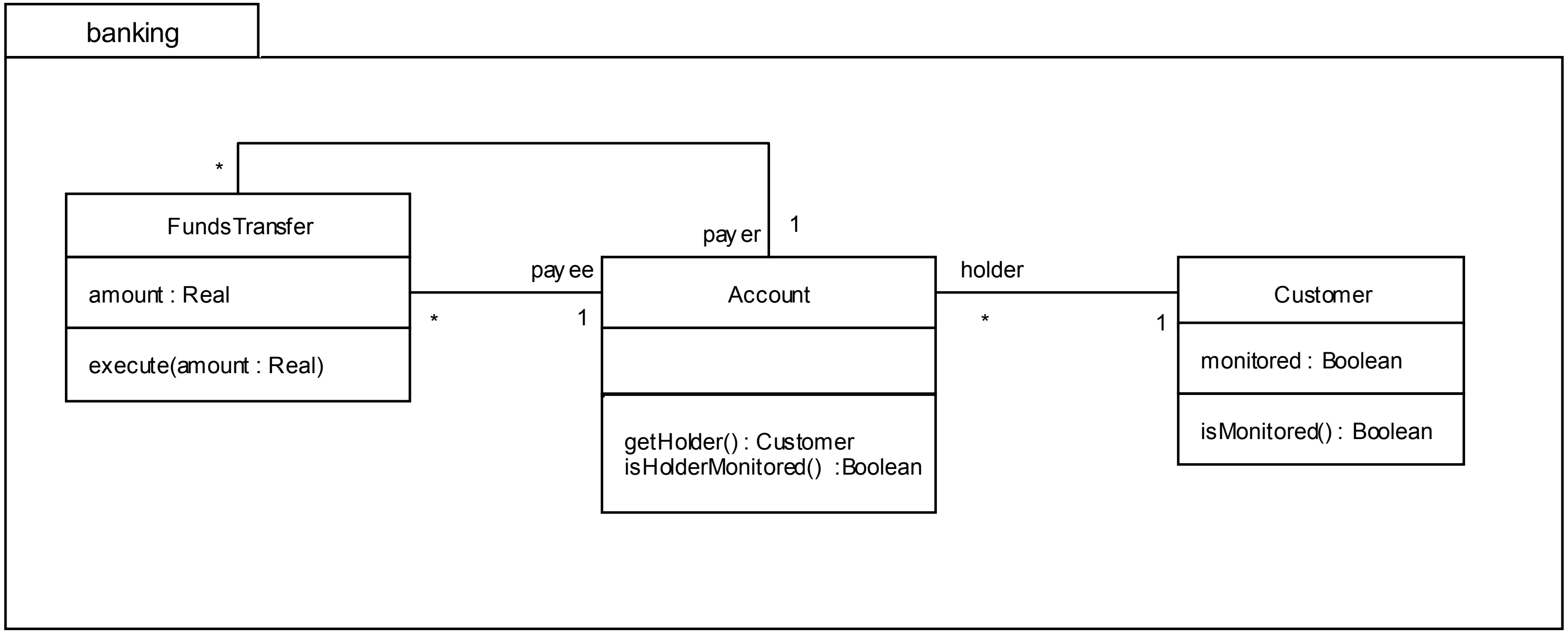
- Cohesion
  - Release-Reuse Equivalency
  - Common Closure
  - Common Reuse
- Coupling
  - Acyclic Dependencies
  - Stable Dependencies
  - Stable Abstractions

Release-Reuse Equivalency

Reasoning:

When developers reuse* a class, they do not want to have to recompile their code every time that class changes. There must be a controlled release process through which the class can be reused. In .NET, the unit of release is the assembly, so the unit of reuse is the assembly. It is for this reason that classes that are highly dependent – and therefore will be reused together - must be packaged in the same assembly.

The unit of reuse is the unit of release

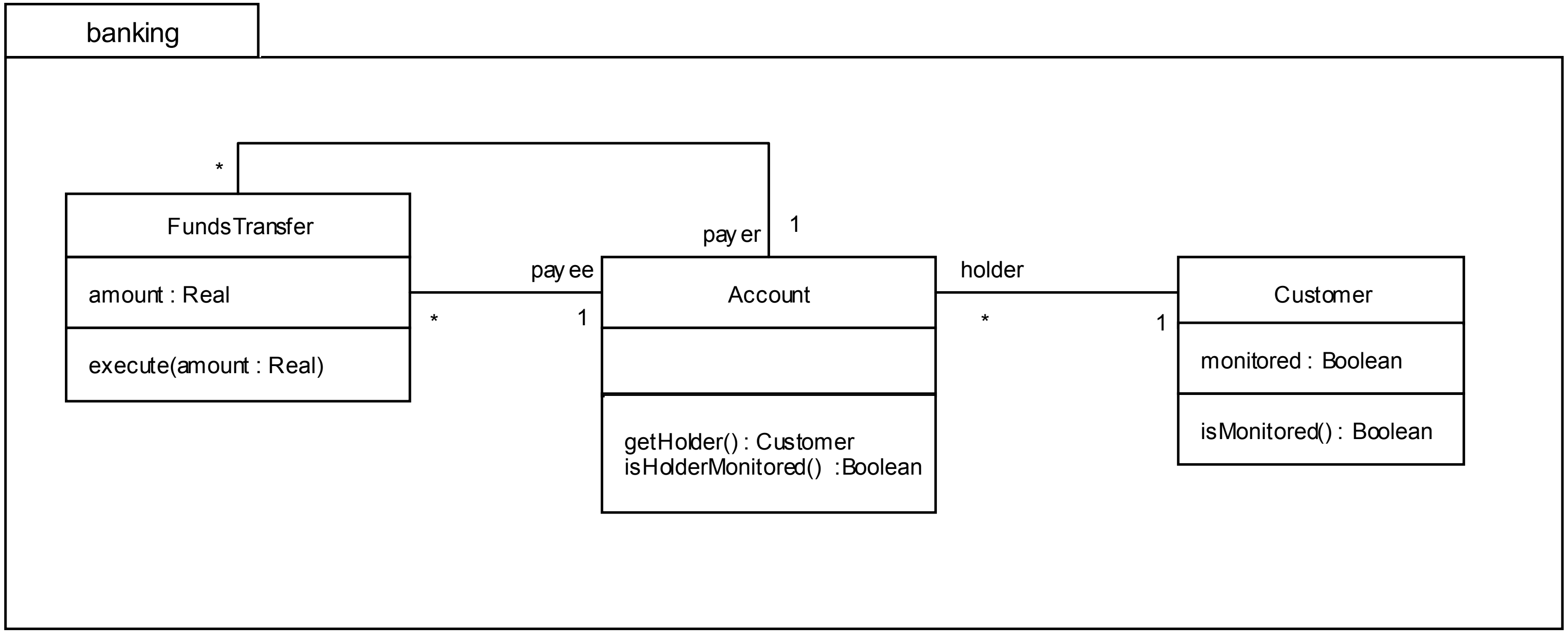


parlez|uml

Common Closure

<u>Reasoning:</u>
A software application will be made up of many packages, and a change in one package can force changes to other packages. This increases the overhead of the build and release cycle, so seek to minimise package dependencies by grouping dependent classes together.

Classes that change together, belong together.



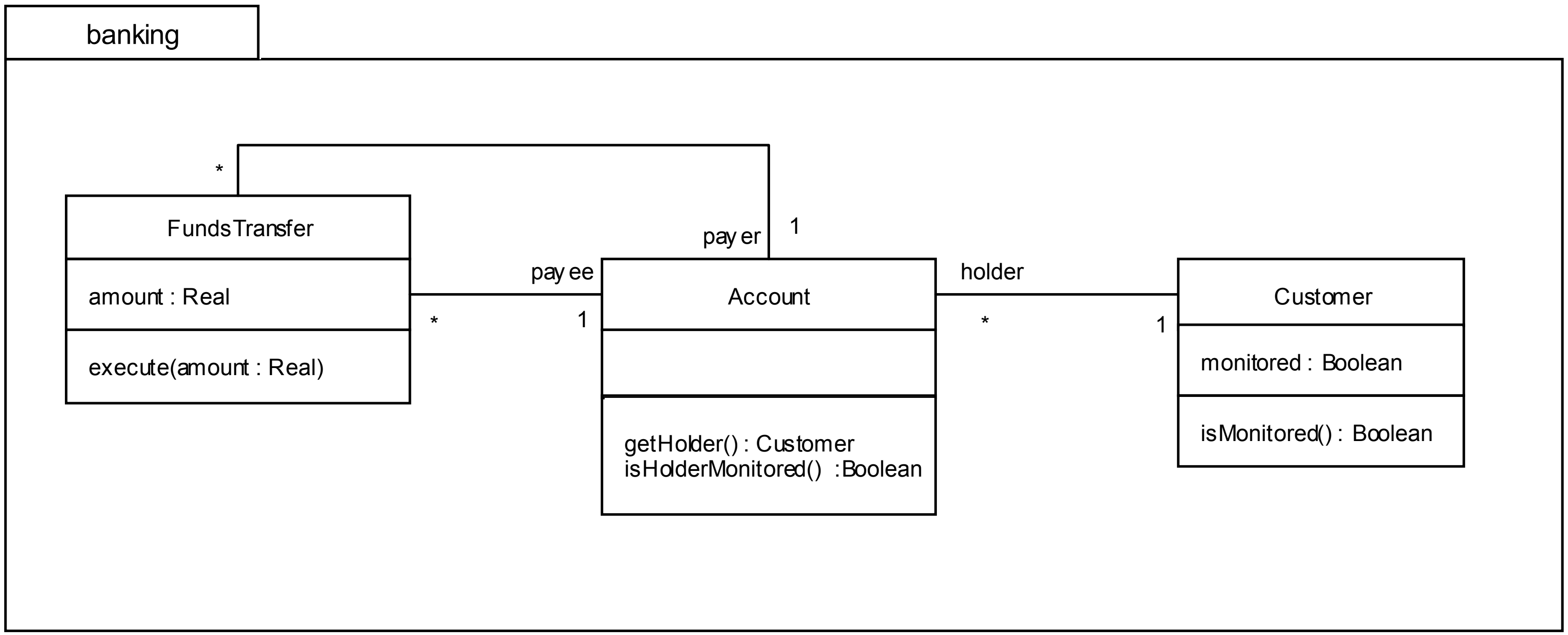
parlez|uml

Common Reuse

<u>Reasoning:</u>

If packages are highly cohesive then a dependency on a package is a dependency on every class in that package.

Classes that aren't reused together, don't belong together.

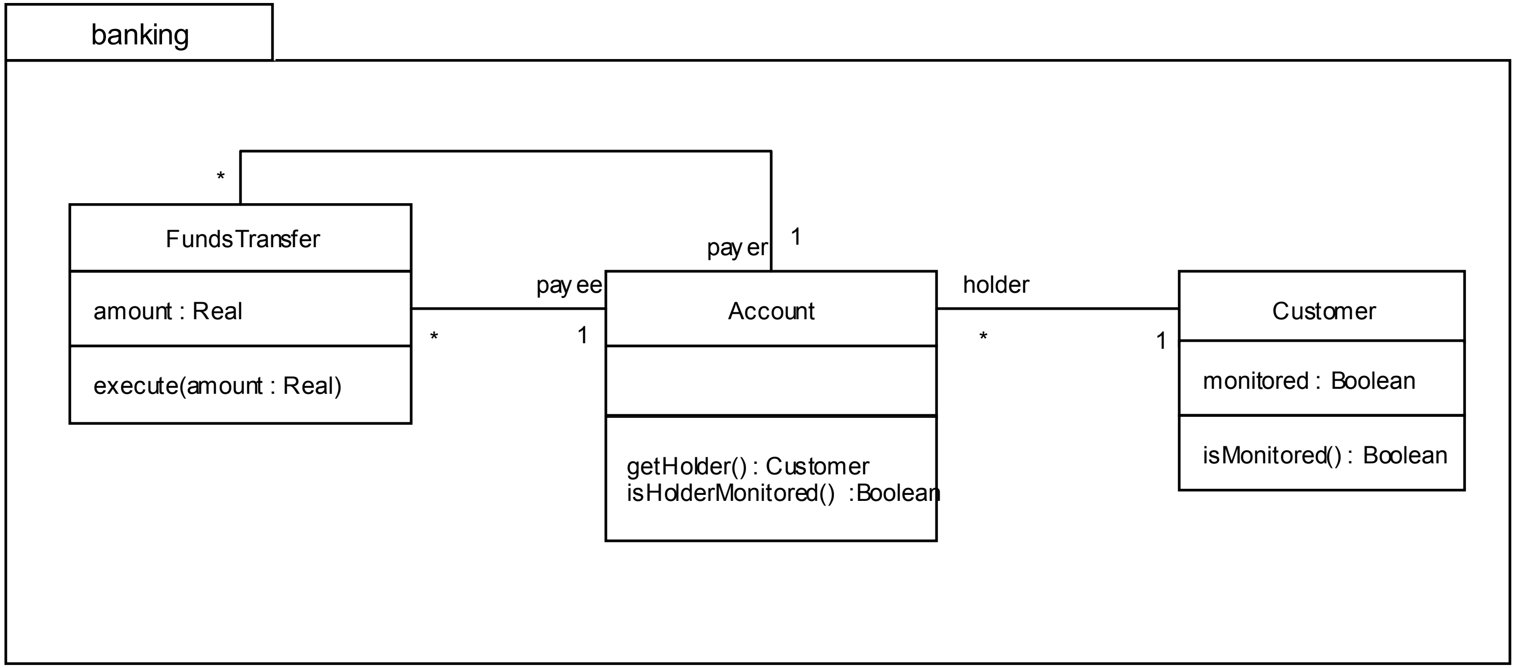


Package Cohesion Metrics

⇒ Class C depends directly or indirectly on N classes in the same package P
⇒ There are M classes in P
⇒ Common reuse & common closure with respect to C is N/(M – 1)
⇒ Package cohesion for P is the average of N/(M – 1) across all classes in P

Except when M <= 1, in which case package cohesion is zero (as opposed to 0 / (1 – 1) which would be *undefined*!)

automation possible

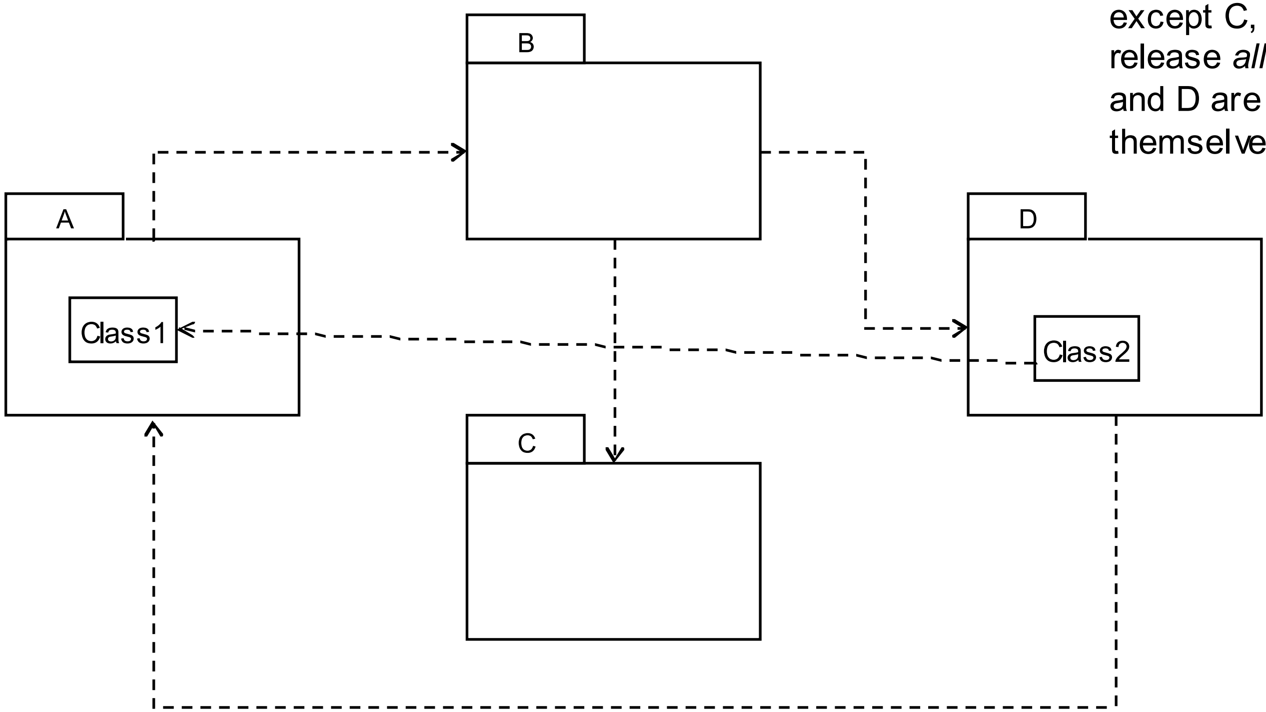


Acyclic Dependencies

Reasoning:

To build and release a package, we must first build and release the packages it depends on. If somehow the package depends indirectly on itself, then you create a potentially much longer build and release cycle.

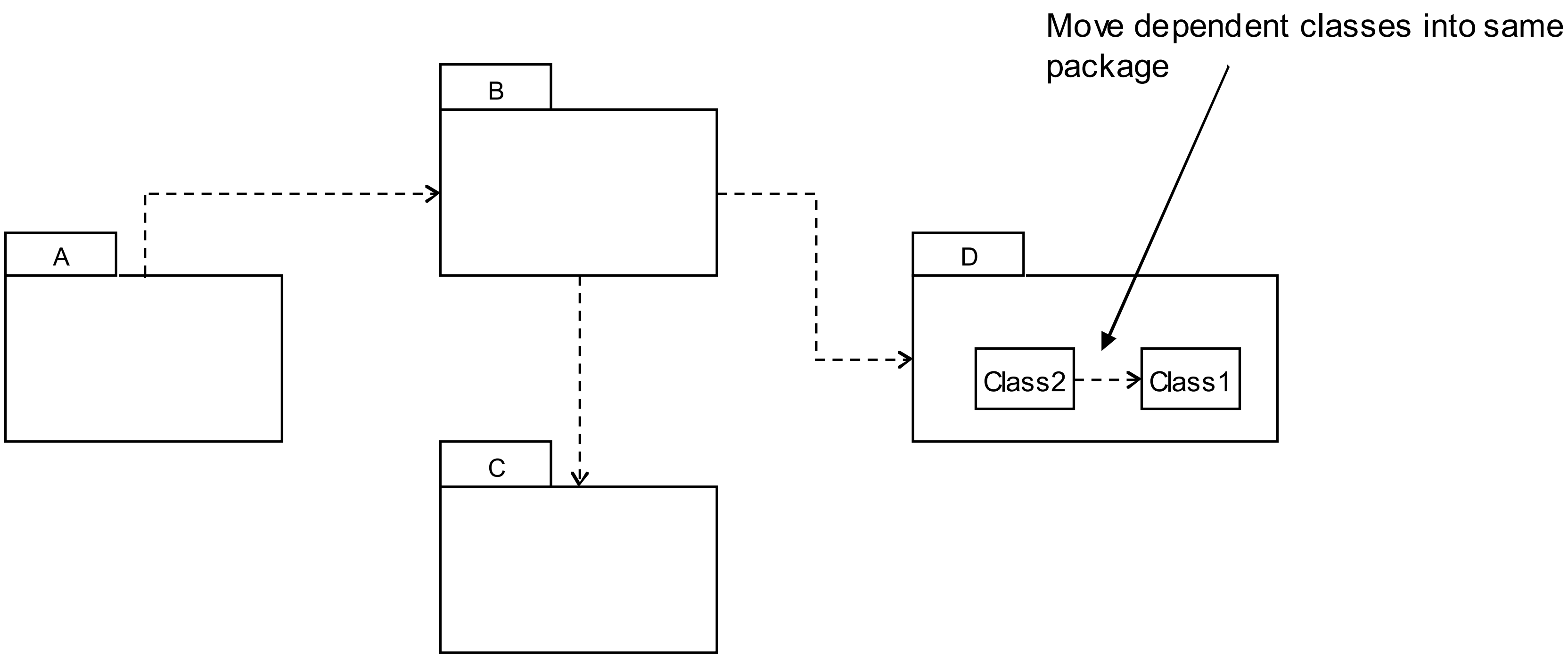
Packages must not be indirectly dependent on themselves



To build and release any package except C, we have to build and release *all* packages because A, B and D are indirectly dependent on themselves

Acyclic Dependencies - Refactored

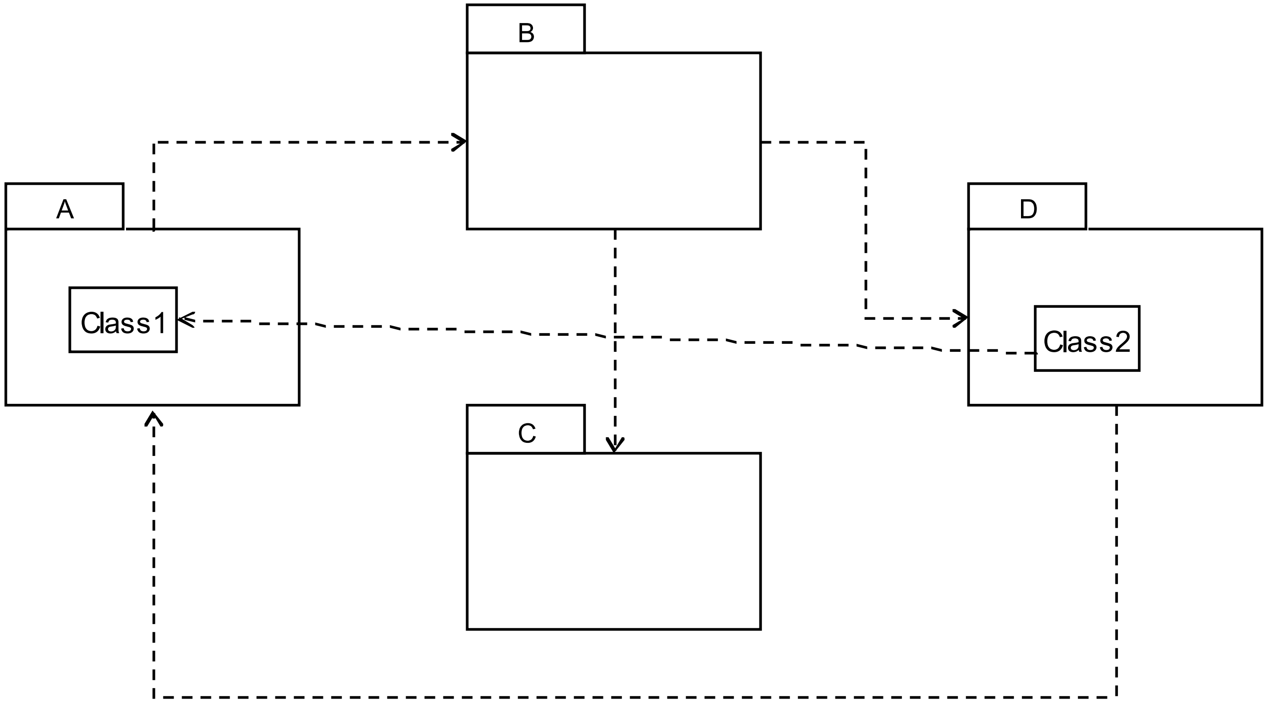
Every package structure must be a **Directed Acyclic Graph**



Acyclic Dependencies - Metrics

No. of cycles in package graph
=> Should not be > 0

automation possible



parlez|uml

Stable Dependencies

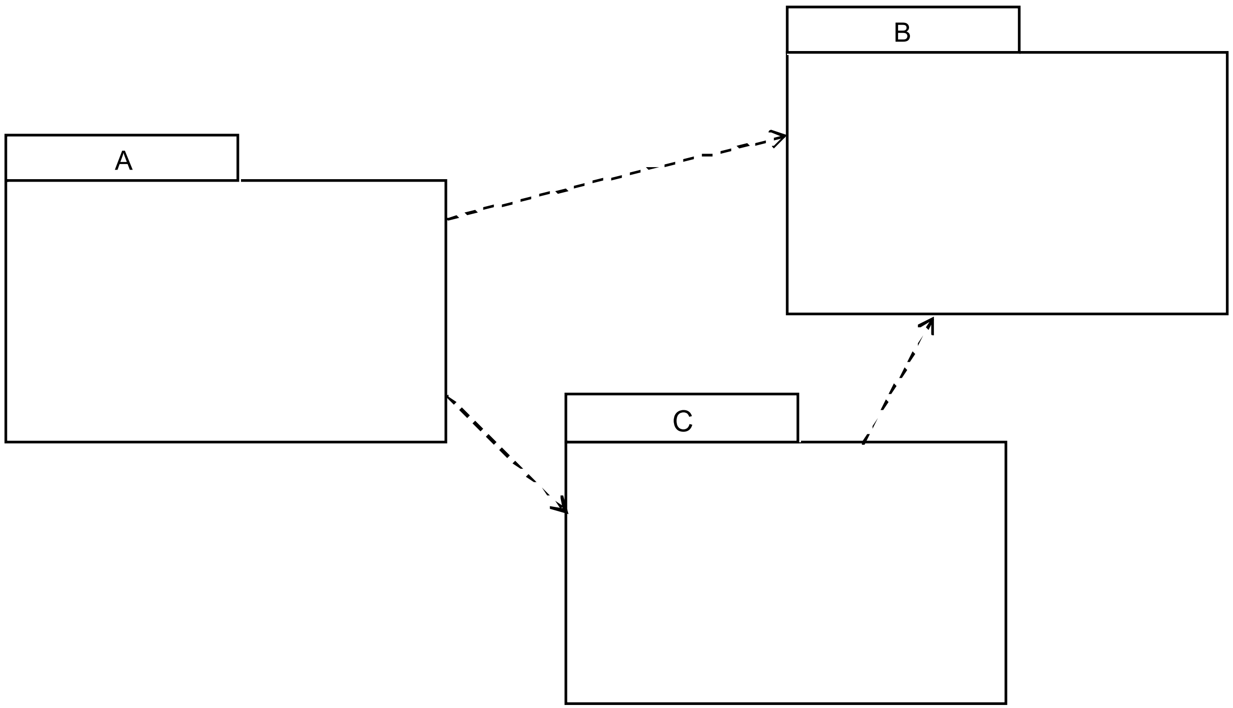
Reasoning:

There are two reasons why we might need to change code in a package.

1. Because we want to (because the logic or design changes)
2. Because changes in another package force us to

   It is for that reason that packages should depend on packages that are more *stable*. Package B has two other packages depending upon it. A change in package B might force us to make changes in A and C. We say that B is *stable* because the effort required to change code in B will be higher, and therefore we're less likely to do it. Package A depends on B and C, and therefore is more likely to have to change because of changes in those packages. We say that B is *instable*.

Packages must depend on more stable packages



Stable Dependencies

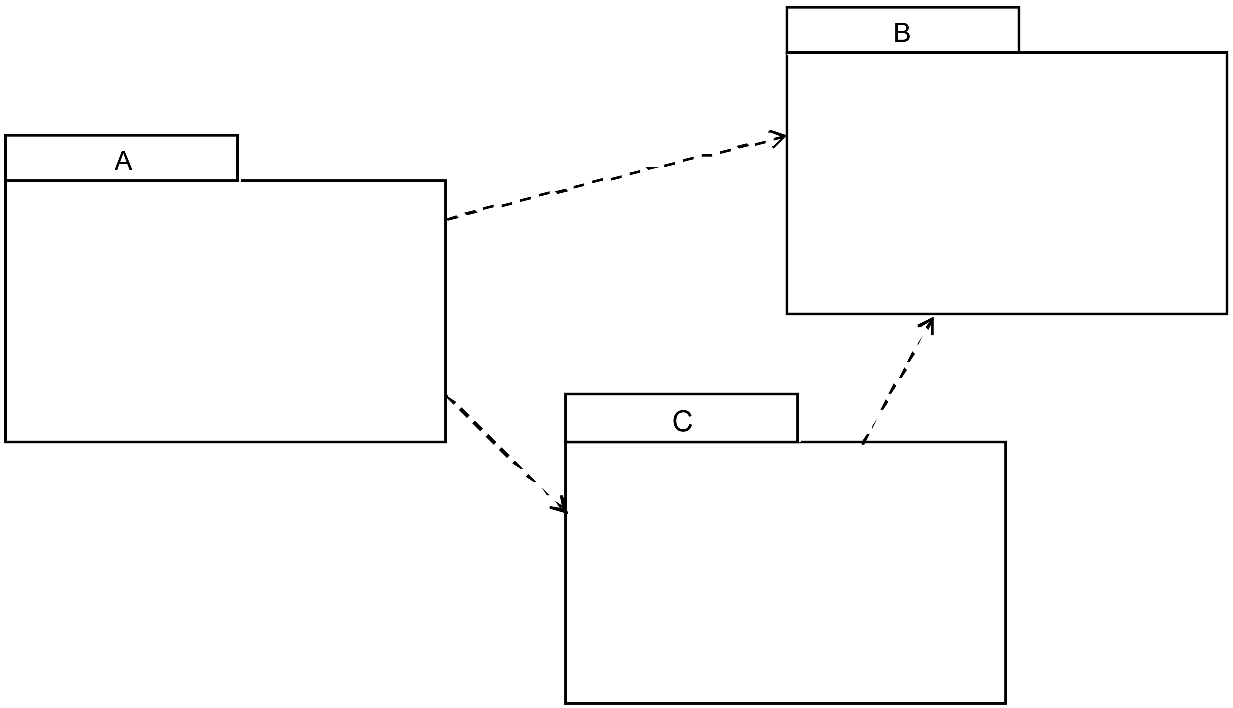
Reasoning:

There are two reasons why we might need to change code in a package.

1. Because we want to (because the logic or design changes)
2. Because changes in another package force us to

   It is for that reason that packages should depend on packages that are more *stable*. Package B has two other packages depending upon it. A change in package B might force us to make changes in A and C. We say that B is *stable* because the effort required to change code in B will be higher, and therefore we're less likely to do it. Package A depends on B and C, and therefore is more likely to have to change because of changes in those packages. We say that B is *instable*.

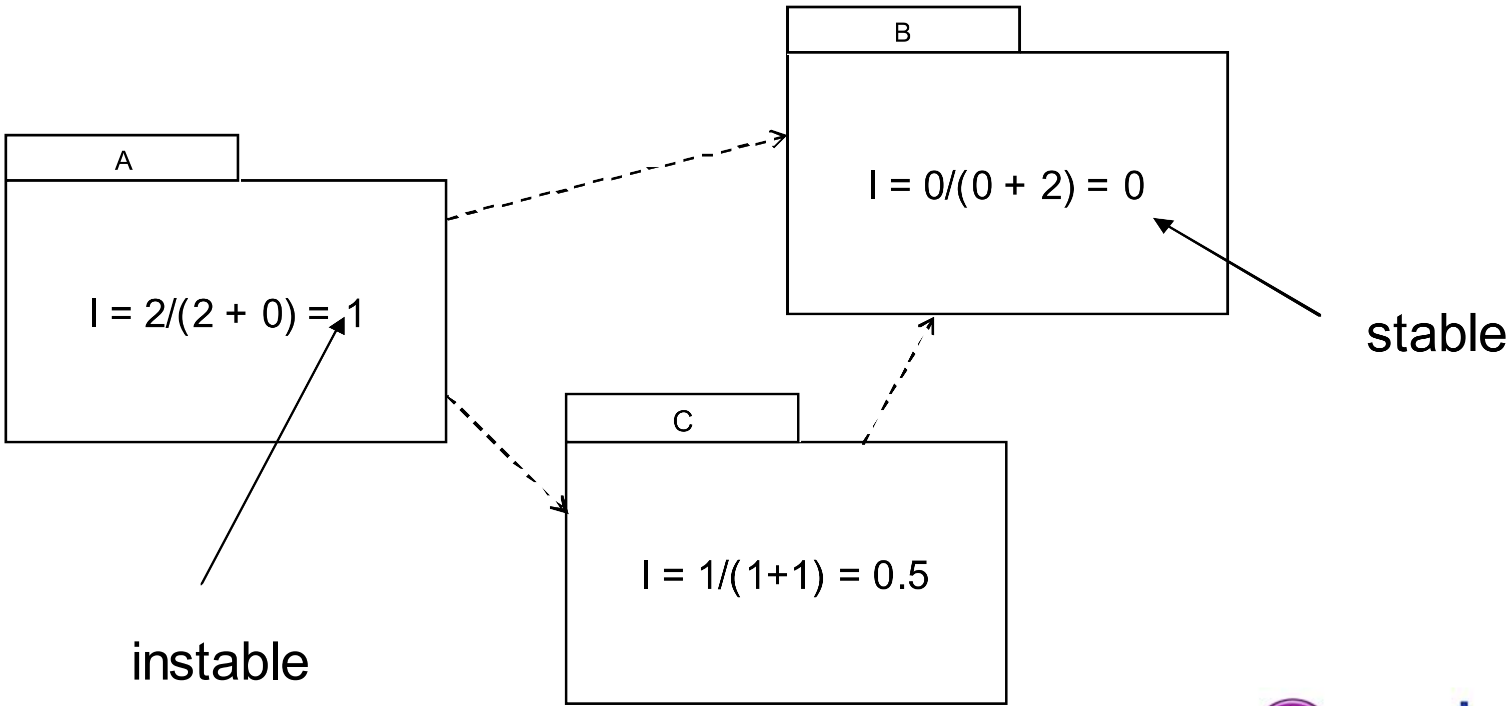
Packages must depend on more stable packages



Stable Dependencies -Metrics

⇒ For a package P, *efferent couplings* C_e is the number of packages that classes in P depend upon
⇒ For a package P, *afferent couplings* C_a is the number of packages that have classes that depend upon P
⇒ *Instability*, $I = C_e/(C_e + C_a)$

automation possible



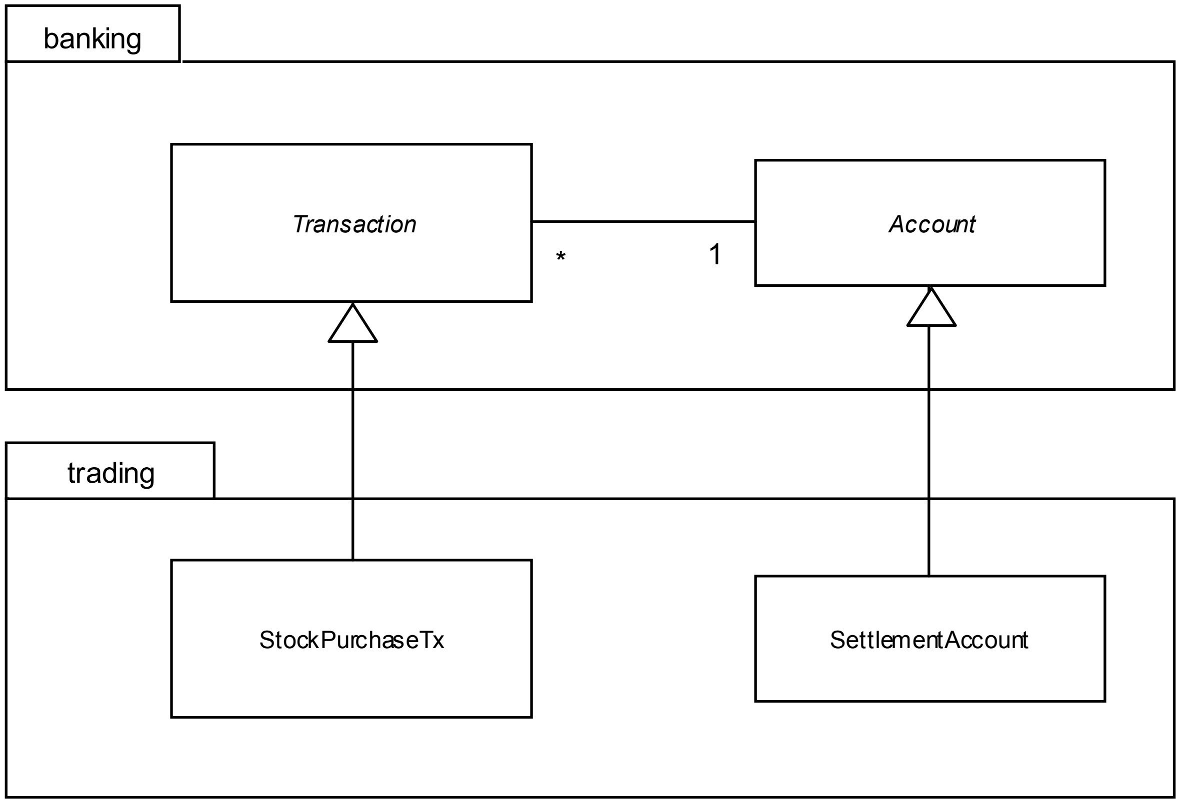
parlez|uml

Stable Abstractions

Reasoning:

Should all software be stable? If our goal is ease of change, then a totally stable package presents a problem. But the Open-Closed principle offers a loophole: a stable package can be easy to extend. By making stable packages abstract, they can easily be extended by less stable packages – which are easier to change. This is Dependency Inversion at the package level.

Packages must depend on more abstract packages.



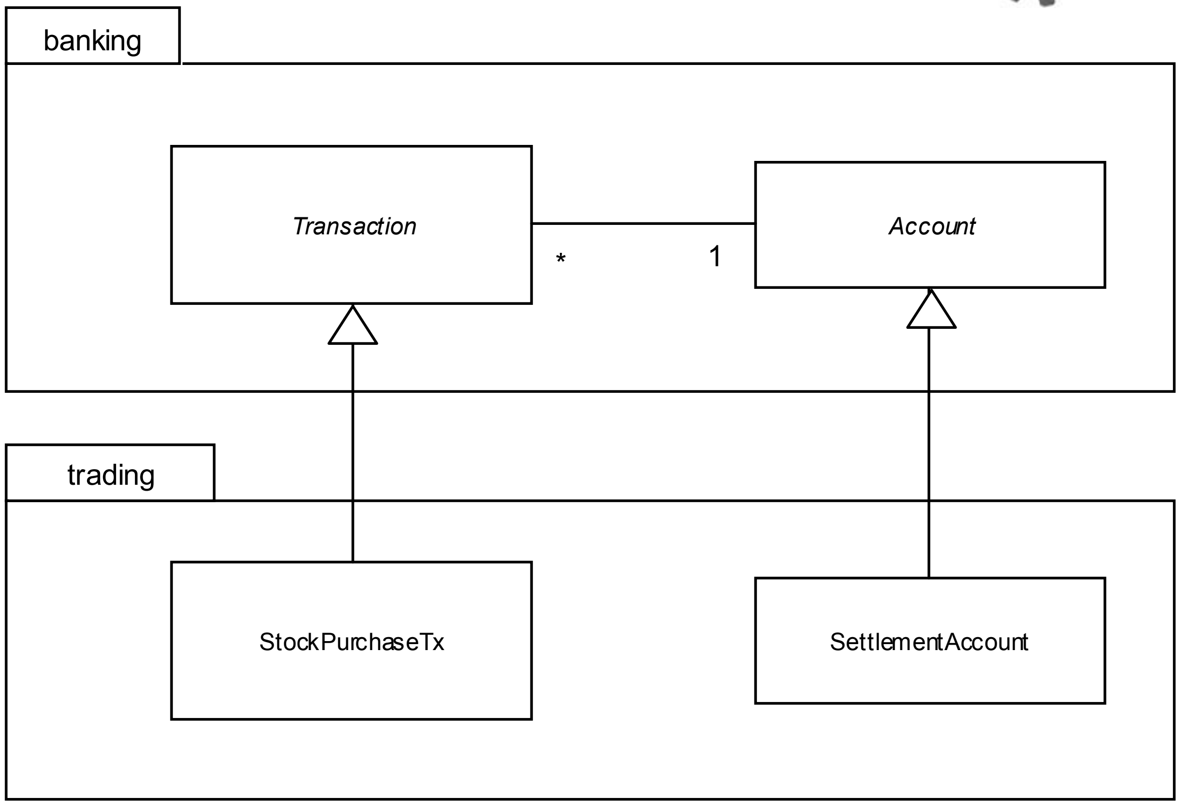
Stable Abstractions - Metrics

⇒Abstractness A = abstract types / all types in package

⇒ Package X depends on set of packages S

- ⇒ Count of packages P in S where abstractness of P – abstractness of X > 0 / total number of packages in S

automation possible



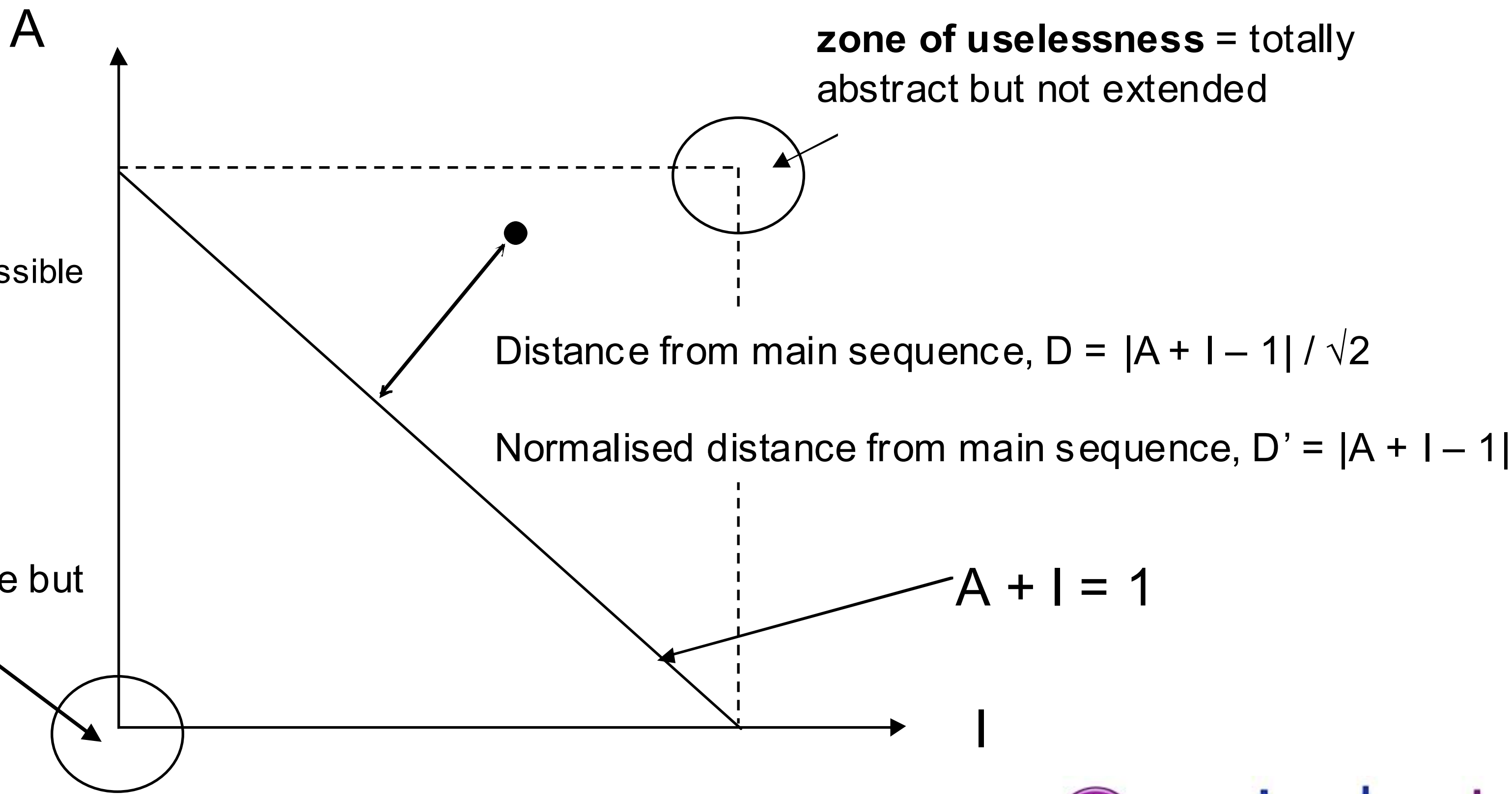
Abstractness vs. Instability - Metrics

<u>Reasoning:</u>

Stable packages should be open to extension, whereas instable packages should be easy to modify. Therefore we seek a *balance* between abstractness and instability

Packages that are more stable should be more abstract

automation possible



References

- Design Principles & Design Patterns – Robert C. Martin, ObjectMentor 2000
  - http://www.objectmentor.com/resources/articles/Principles_and_Patterns.PDF