

7CCSMSUF: Software Engineering and Underlying Technologies for Financial Systems

Kevin Lano

`kevin.lano@kcl.ac.uk`

Part 4. Software Design using UML

Kevin Lano

Weeks 7 and 8

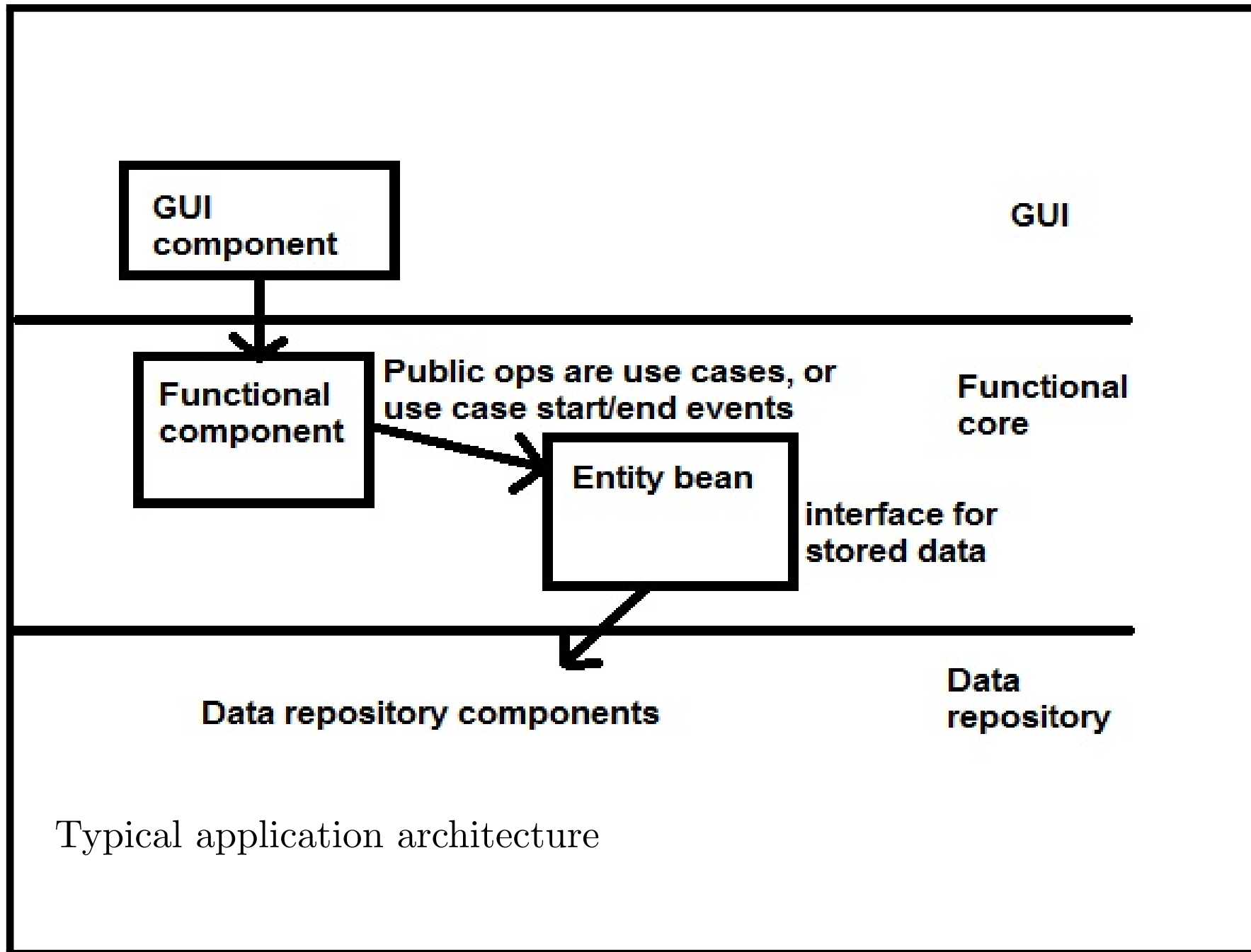
- Architecture diagrams
- Design patterns
- Design refactoring
- Case study: QuantLib

Architecture diagrams

Design concerns construction of components and organising their interactions to achieve specified system requirements.

Involves:

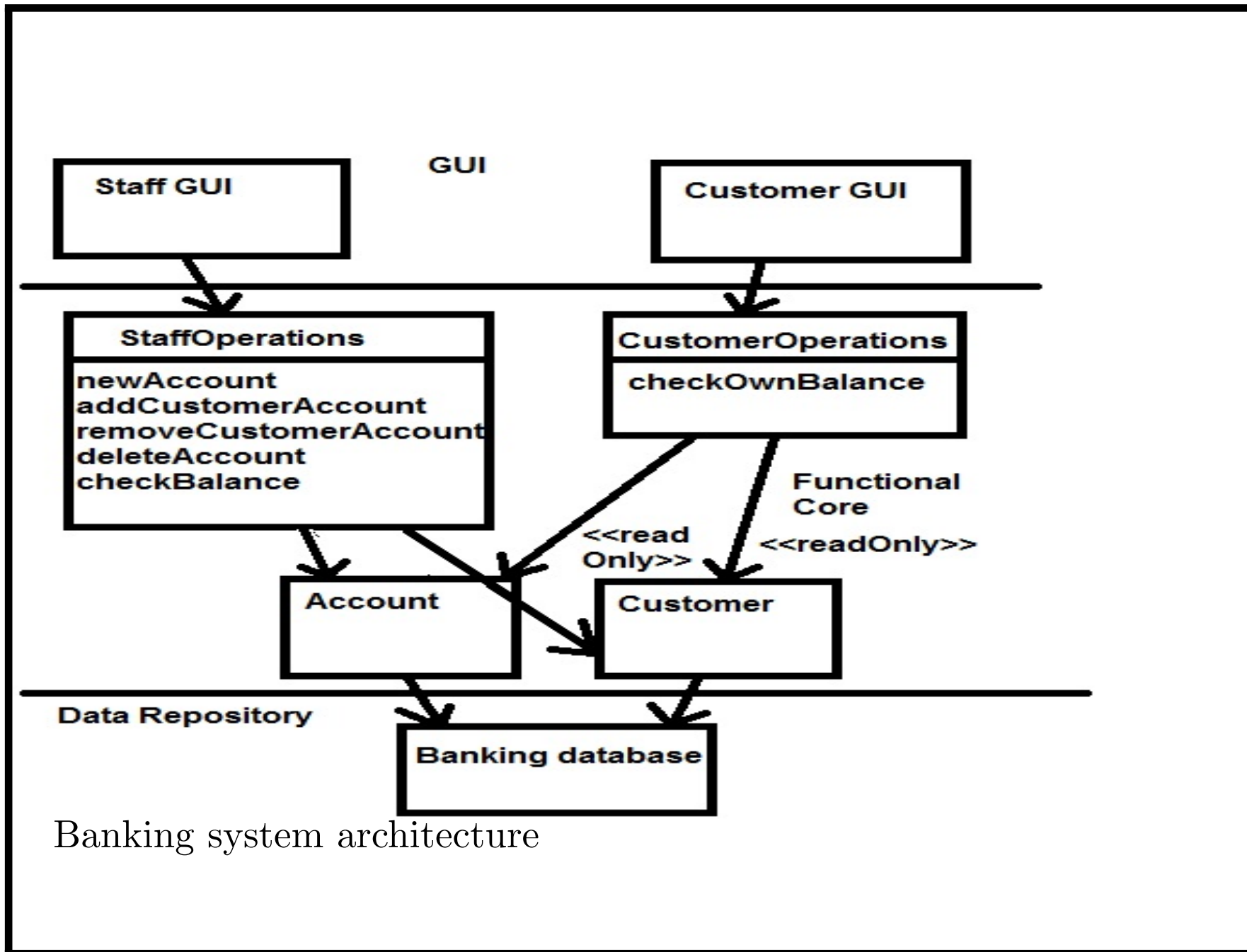
- Identifying subsystems + modules with coherent + well-defined purpose + functional cohesion.
- Identifying dependencies between modules, specifying their interfaces and responsibilities.
- Modules may be single classes, or a group of closely related classes.



Architecture completeness

- In completed design each required use case + system constraint must be carried out/maintained by some module, or by combination of modules.
- Some functionality can be achieved by an individual module.
- Other functionalities will need complex design decisions, + interaction of several modules.

E.g., *checkOwnBalance(cId, aId)* needs to read both *Account* and *Customer* data.



Architecture diagrams

- Architecture description diagrams + natural language text specifications can describe system components, + dependencies between them.
- Components (subsystems and modules) represented as rectangles.
- Subcomponents of component nested inside it.
- Arrow from component X to component Y means X *depends on* Y : it invokes operations of Y , or refers to data of Y .
- X is termed a *client* of Y , Y is a *supplier* of X .
- Operations of module (services it offers to clients) can be listed in module rectangle.

Architecture diagrams

- Such diagrams can also be expressed using UML package notation.
- Natural-language or UML descriptions to define module responsibilities, data it manages, operations it performs, + its interface (set of operations it provides to rest of system).
- *Functional co-ordinator components*: Modules whose operations correspond to use cases, accessed via UIs for actors linked to cases. E.g., *StaffOperations*.

Design patterns

- Design patterns are structures of software used to solve particular design problems.
- Mainly independent of programming languages, can be used for platform-independent design.
- Concept of ‘design pattern’ originated in building architecture (Christopher Alexander, 1977).
- Subsequently, software researchers discovered ‘design patterns’ of software.
- Popularised by “Gang of Four” (aka GoF) book – introduced 23 design patterns for object-oriented software.
- GoF divided patterns into 3 general categories: *creational*, *structural*, *behavioural*.

Kinds of design patterns

Creational: Organise creation of objects + object structures. E.g:
Singleton, Factory

Behavioural: Organise distribution of behaviour amongst objects.
E.g: *Iterator, Observer, Strategy*

Structural: Organise structure of classes and relationships. E.g:
Proxy, Facade.

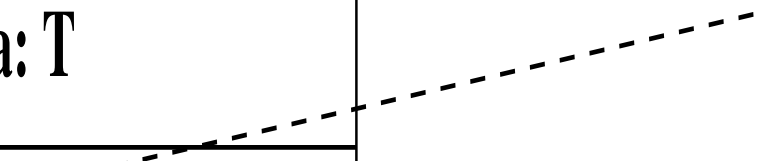
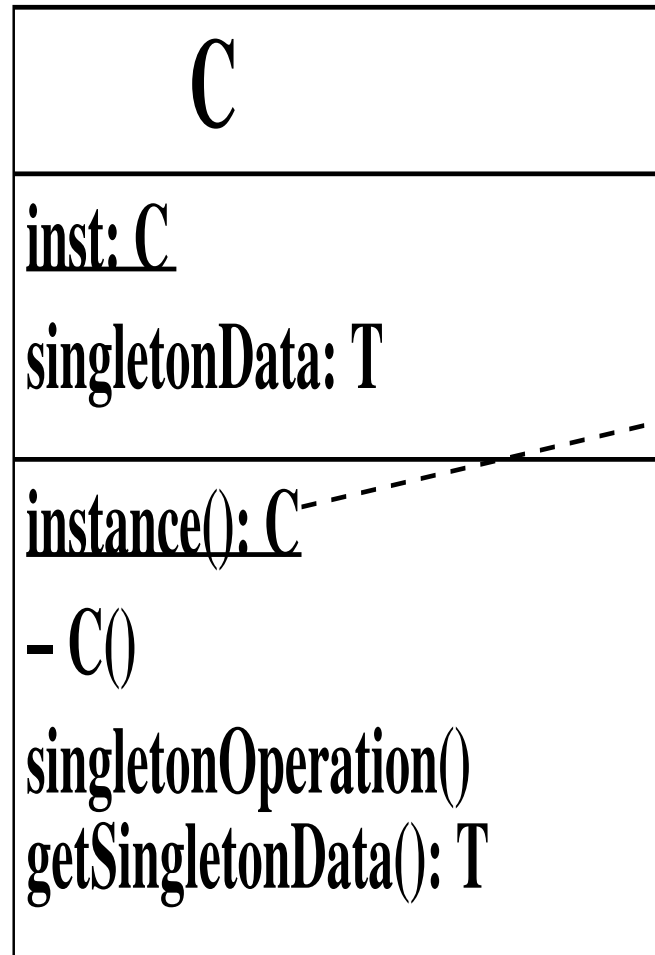
In any significant sized application, likely that some patterns will be useful: make design simpler, more flexible, more comprehensible.

Singleton pattern

Creational pattern used to define classes which should have only a single instance (e.g, a database connection pool, etc).

Singleton is used when:

- There must be unique instance of a class, accessible to clients from well-known access point;
- when instance should be extensible by subclassing, clients should be able to use an extended instance without modifying their code.



```
if (inst == null)
{ inst = new C(); }
return inst;
```

Structure of Singleton pattern

Singleton pattern

The involved classes are:

- *Singleton* – defines operation *instance* that lets clients access its unique instance. *instance* is a class-scope (static) operation.
- *Singleton* may also be responsible for creating its own unique instance.

The constructor of *Singleton* is private (hence the - visibility).

Singleton pattern

Benefits of this pattern are:

- provides controlled access to *Singleton* instance – can only be accessed via *instance* method;
- reduces name space of program – a class instead of a global variable;
- permits subclassing of *Singleton*;
- can be adapted easily to give fixed number $N > 1$ of instances.

In a language translation system, a *Dictionary* could be a *Singleton* class, so can be used globally, reducing number of parameters of operations which look up words:

```
public class Dictionary
{ private static Dictionary uniqueInstance = null;
  ...

  private Dictionary() { }

  public static Dictionary getDictionary()
  { if (uniqueInstance == null)
    { uniqueInstance = new Dictionary(); }
    return uniqueInstance;
  }

  public boolean lookup(Word w)
  { ... }
```

}

Because constructor is private, only *Dictionary* class itself can construct *Dictionary* instances: only done once, when *getDictionary* called for first time.

Client uses dictionary by calls

```
boolean b = Dictionary.getDictionary().lookup(w);
```

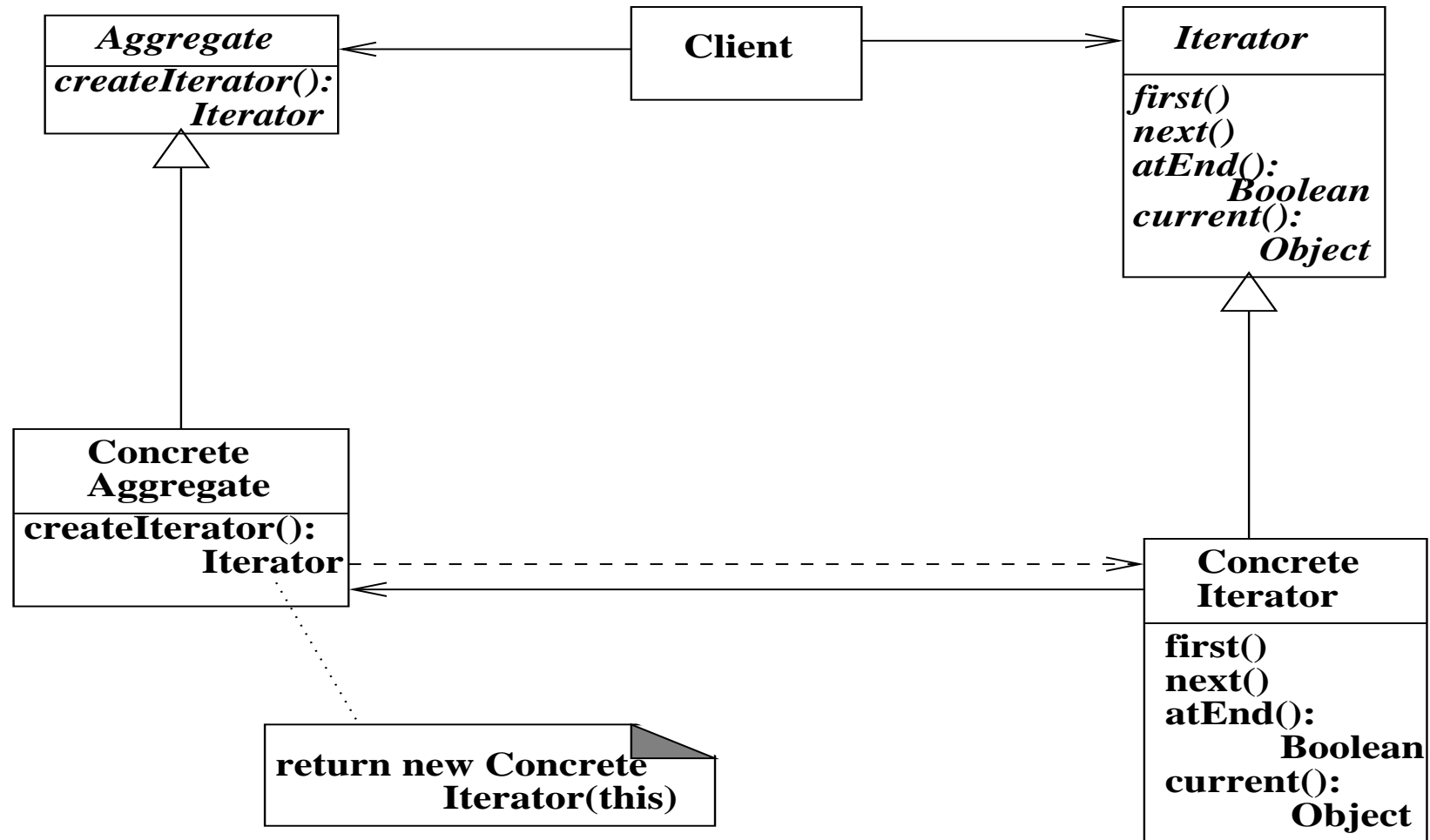
This call will be valid anywhere in the application.

The Iterator pattern

Purpose of this behavioural pattern is to support access to elements of aggregate data structure (such as tree or array) sequentially.

It is applicable whenever:

- We require multiple traversal algorithms over an aggregate;
- we require uniform traversal interface over different aggregates;
- or when aggregate classes and traversal algorithm must vary independently.



Structure of Iterator pattern

Iterator pattern

- Iterator object acts like cursor or pointer into a structure, indicating current location within structure and providing operations to move cursor forwards or backwards in structure.
- Normally an *Iterator* class has operations such as *atEnd* / *hasNext* to test if iteration has reached an end, *advance* / *next* to step forward to next element, and *current* to obtain current element.

Classes involved are:

- *Aggregate* – class defining a general composite data structure such as list or tree.
- *ConcreteAggregate* – specific subclass defining particular data structure such as linked list or binary search tree.
- *Iterator* – interface for general iteration operations such as accessing first element in collection, stepping through collection, etc.
- *ConcreteIterator* – iterator subclass specific to particular data structure. *createIterator()* method of the data structure returns a *ConcreteIterator* instance for structure.

Benefits/disadvantages of Iterator pattern

Consequences of pattern are an increase in flexibility, because aggregate and traversal mechanism are independent.

Possible to have multiple iterators acting on same aggregate object simultaneously, possibly with different traversal algorithms: e.g, using both a post and pre-order traversal of a tree.

If iterator pattern was not used, would require direct access to private parts of data structures being iterated over.

However, the pattern requires instead communication between objects – operation calls from one object to another.

So we gain flexibility at cost of efficiency.

Iterator example in Java

```
List<String> list = new LinkedList<String>;  
// add some elements to list  
  
ListIterator<String> iter = list.listIterator();  
  
while (iter.hasNext())  
{ String item = iter.next();  
  iter.set(item.toUpperCase());  
} // change items in list to upper case
```

Cannot change list itself (by *add*, *remove*, etc) while iteration is in process, only via iterator.

Iterator operations in Java, C#, C++

<i>Operator</i>	<i>Java</i>	<i>C#</i>	<i>C++</i>
<i>atEnd</i>	!hasNext	!MoveNext	it == col.end()
<i>first</i>		Reset	col.begin()
<i>getIterator</i>	iterator listIterator	GetEnumerator	col.begin()
<i>next</i>	next	MoveNext	++
<i>previous</i>	previous (if supported)		--
<i>current</i>		Current	*it

C++ has several different iterator categories with different capabilities.

Iterators in C++: the Standard Template Library (STL)

C++ defines library of algorithms which work on *iterators* for data structures, not directly on the data structures. The algorithms can therefore work on many different structures.

For data structure v , $v.begin()$ is iterator positioned at start of v . $v.end()$ is end-marker iterator.

```
std::find(v.begin(), v.end(), x);
```

performs search for x starting at $v.begin()$.

```
std::sort(v.begin(), v.end());
```

sorts v between $v.begin()$ and $v.end()$.

Types of iterators in C++

Basic iterator with *start()*, *next()*, *element()*, *atEnd()* is termed a *ForwardIterator*, other kinds of iterator are:

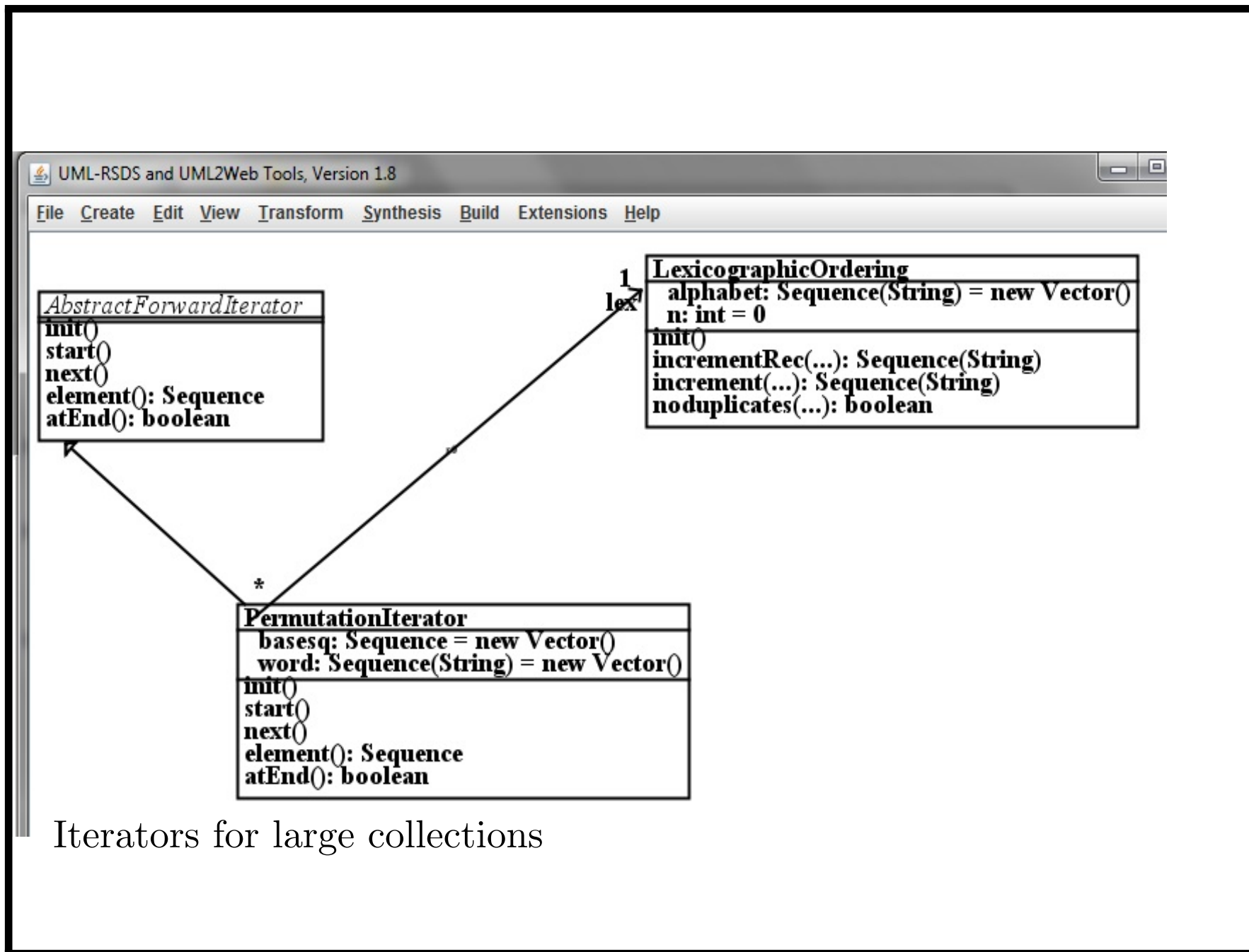
- *Bidirectional iterators*, supporting additional *previous()* and *atStart()* operations
- *Random access iterators*, supporting access to positions via an index, these iterators provide operations such as *get(i : int)*.

General iteration algorithm for a forward iterator *it* is:

```
it.start() ;  
while not(it.atEnd())  
do  
    ( ... process it.element() ...;  
      it.next()  
    )
```

Iterator example: iterating over permutations

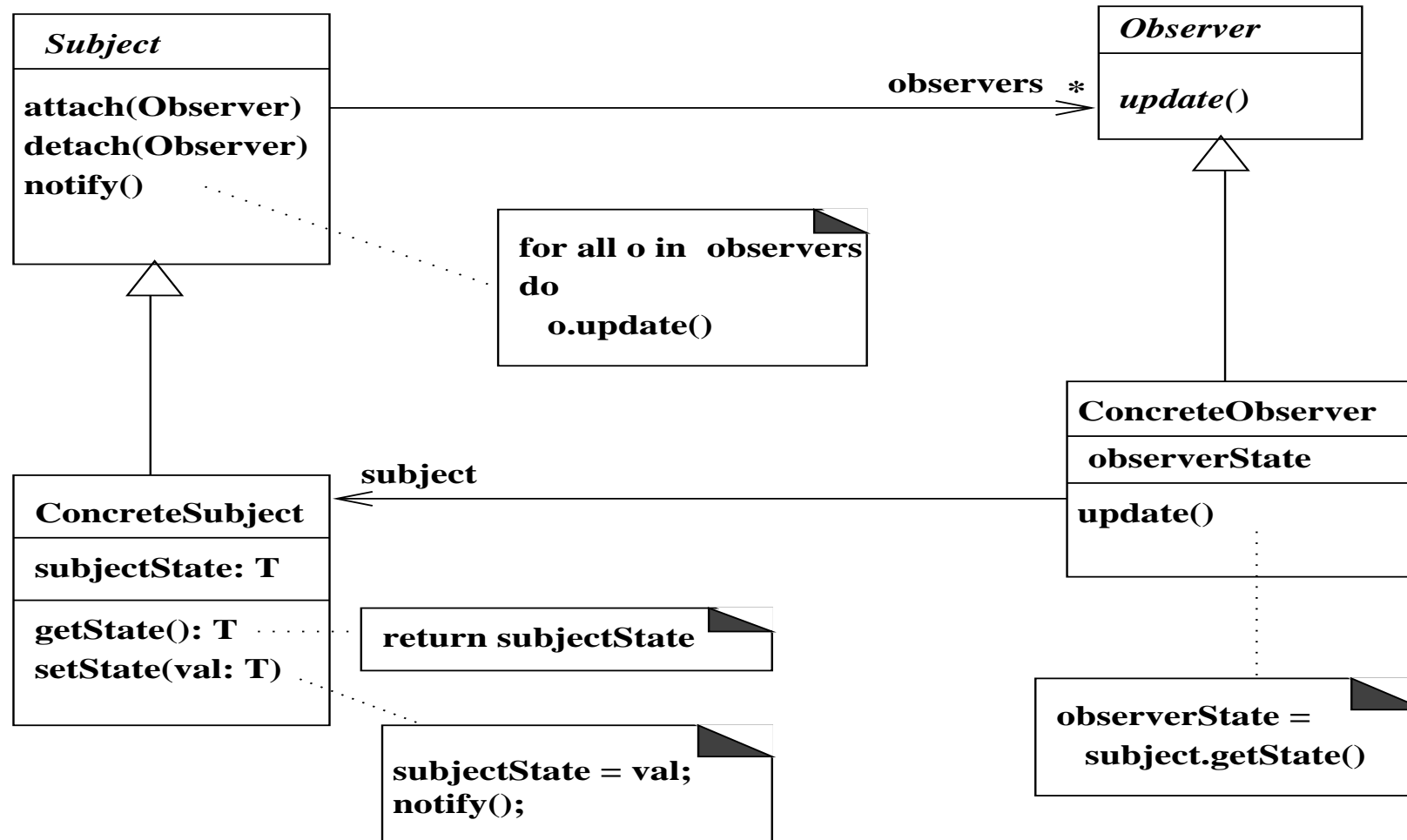
- Can use iterators to generate, one by one, elements of collections which are too large to hold in memory at once.
- E.g., all permutations of a large collection (number of permutations of n elements is $n!$).
- Idea is to step in lexical order through all words of an alphabet formed from collection, selecting words with no duplicates.
E.g., **abc**, aca, **acb**, acc, baa, **bac**, bba, ...



Iterators for large collections

The Observer pattern

- Behavioural pattern intended to manage multiple views or presentations of data, such as alternative graphical views (pie charts and bar charts of sales figures, for example).
- Defines one-to-many dependency between subjects + observers so that when subject (data) changes state, all its dependants (views) are notified and update themselves.
- Applicable whenever two entities represented in software, one dependent on other, so that change to one object requires changes to its dependants.



Design structure of Observer pattern

Observer classes

- *Subject* – abstract superclass of classes containing observed data. Has methods *attach* and *detach* to add/remove observers of subject, *notify* to inform observers that state change occurred on observable, so they may need to update their presentation of it.
- *ConcreteSubject* – holds observable data, any method of this class which modifies data may need to call *notify* on completion.
- *Observer* – abstract superclass of observers of subjects. Declares *update* method to adjust observer's presentation on any subject state change.
- *ConcreteObserver* – class defining specific view, such as bar chart.

Observer properties

- Important to maintain *referential integrity*, means that for subject object s , its set $s.observers$ of attached observers has property

$$o : s.observers \equiv o.subject = s$$

I.e., its observers are exactly the observer objects whose subject is s .

- Pattern widely used in commercial languages + libraries, e.g., Model-View-Controller (MVC) paradigm for web or mobile applications.

Benefits/disadvantages of Observer pattern

Positive consequences of using pattern:

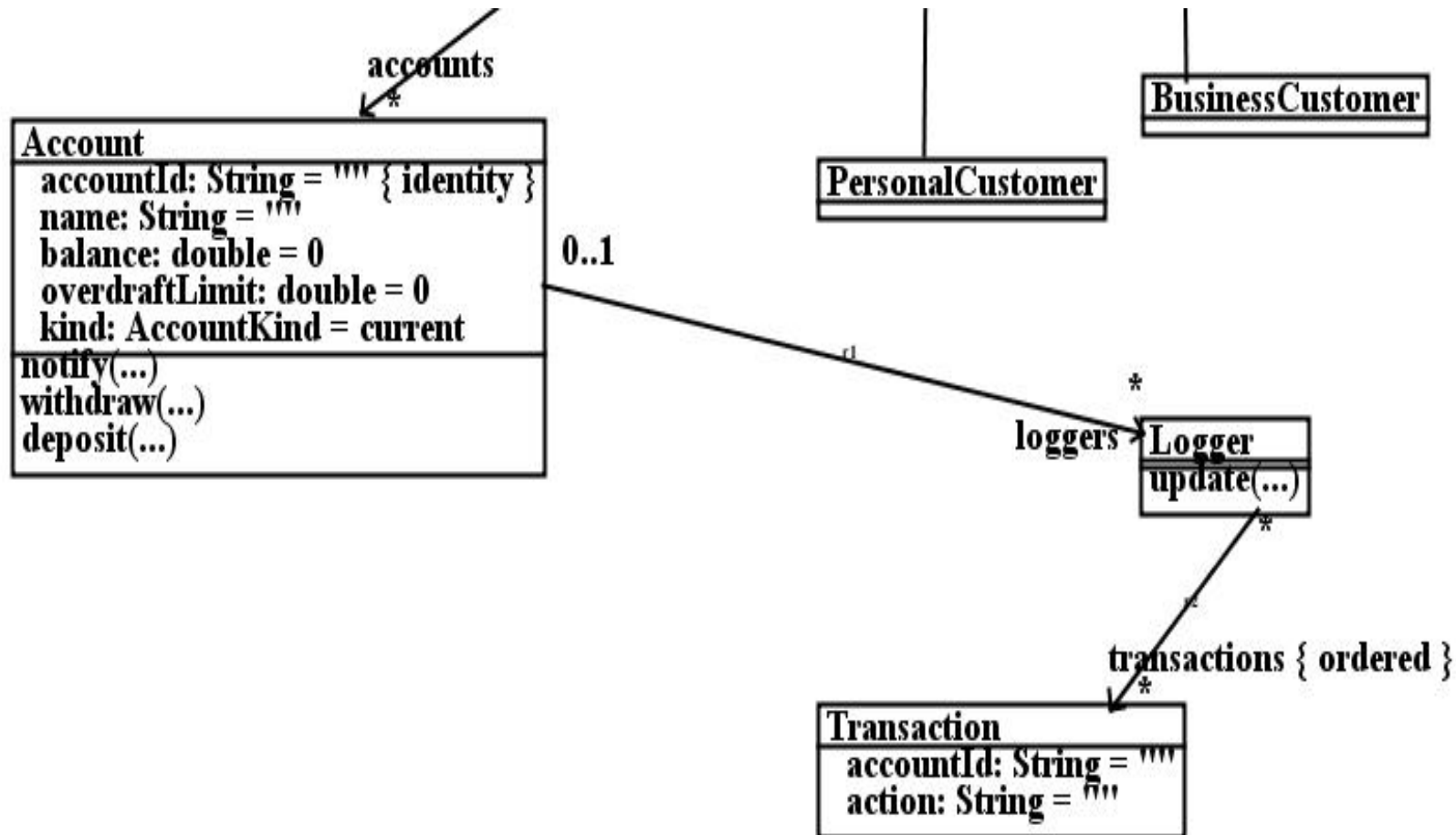
- Modularity: subject and observers may vary independently.
- Extensibility: can define and add any number of observers to given subject.
- Customisability: different observers provide different views of subject.

Disadvantages – (i) cost of communication between objects; (ii) maintaining referential integrity of *observers/subject*.

Communication cost can be reduced by sending data changes in *update* call, so no *subject.getState()* call needed.

Example: logging of accounts

- Bank needs to retain log of all transactions on set of accounts
- Define *Logger* class to store sequence of transactions, each transaction records *accountId* and action performed (*withdraw*, *deposit*, etc)
- *Logger* is an *Observer* with respect to *Account* in role of *Subject*
- *notify* invoked by *deposit* and *withdraw*, sends transaction data to all attached *Logger* objects, via *update* invocations
- Each logger responds to *update(aId, data)* by creating new *Transaction* and storing this in log sequence.



Observer pattern for account logging

Account logging: Subject code

Account operations call *notify*:

Account::

deposit(amt : double)

pre: amt >= 0

post:

balance = balance@pre + amt &

notify("deposit " + amt)

Account::

withdraw(amt : double)

pre: balance - amt >= -overdraftLimit

post:

balance = balance@pre - amt &

notify("withdraw " + amt)

Logging observer code

notify sends data to loggers:

Account::

notify(s : String)

post:

loggers->forall(lg | lg.update(accountId, s))

Logger::

update(id : String, s : String)

post:

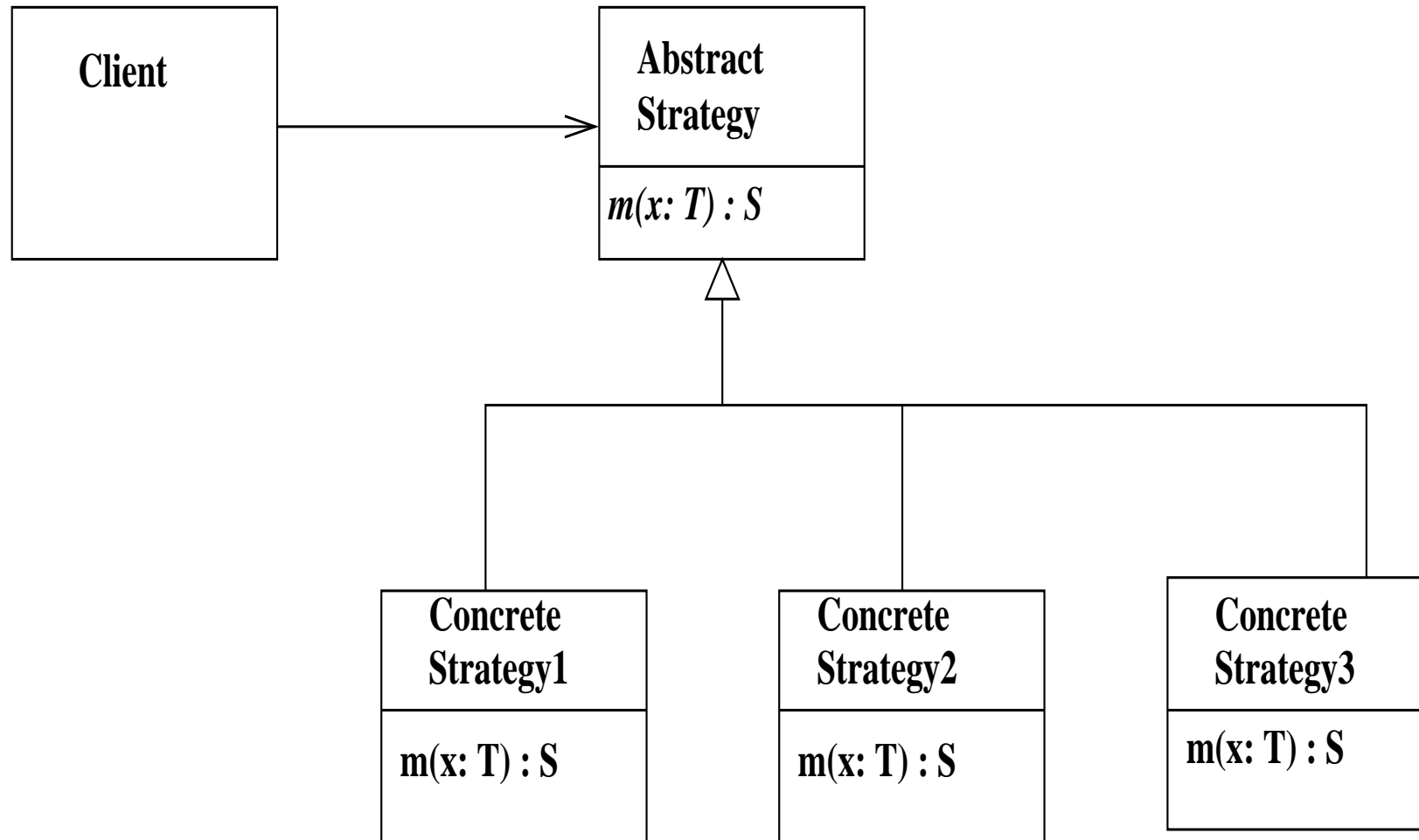
Transaction->exists(t | t.accountId = id &
t.action = s & t : transactions)

Strategy pattern

Behavioural pattern, which defines common interface for alternative versions of an algorithm, enabling a client to easily select these versions by varying object the operation is invoked on.

Elements of pattern are:

- *Client* – class which uses operation via abstract class or interface.
- *AbstractStrategy* – a class (or interface) that provides common interface for different implementations of operation.
- *ConcreteStrategy* – a class that implements particular version of the operation.



Strategy pattern

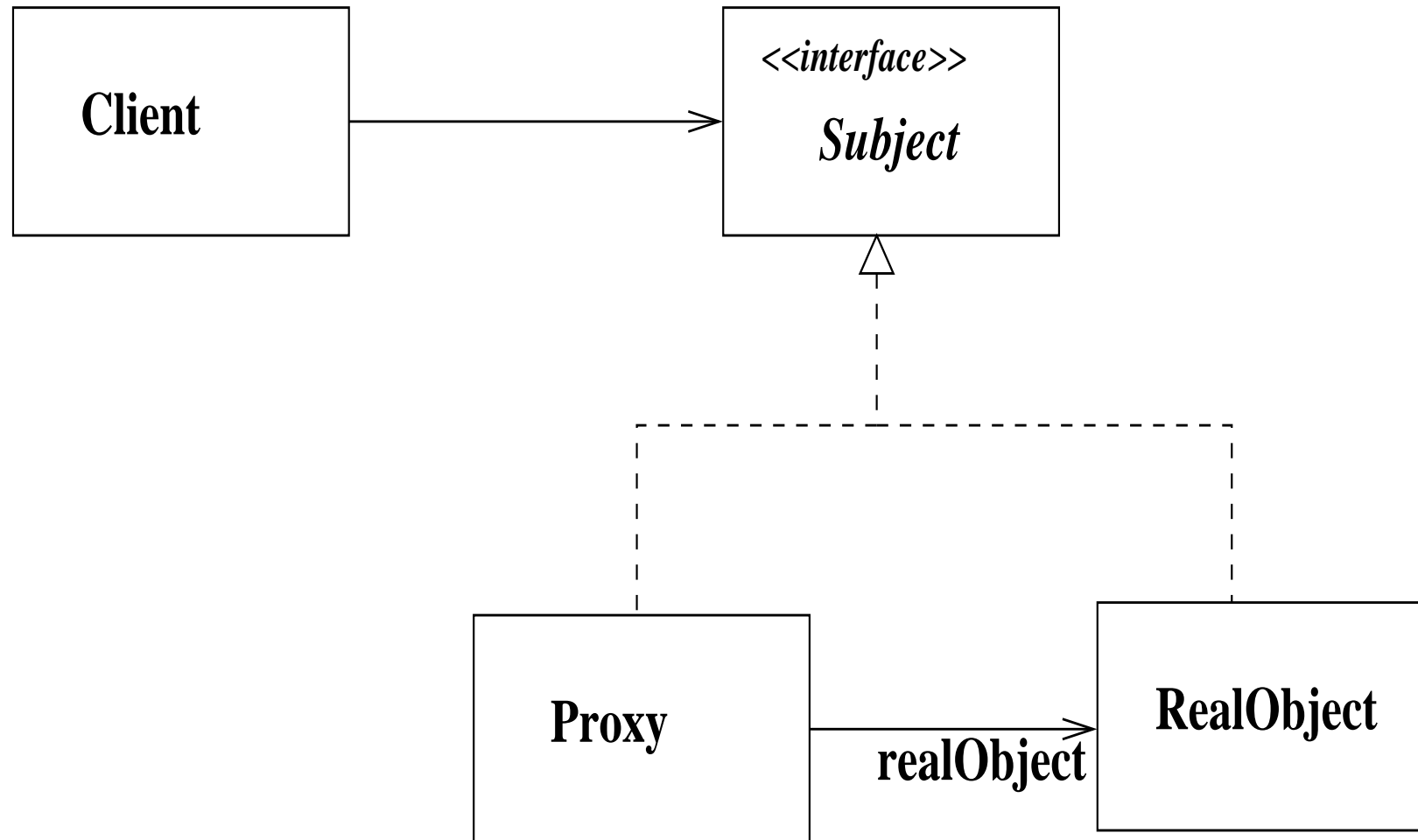
Benefits of Strategy pattern

- Pattern simplifies code of system by separating alternative versions of an algorithm into different classes
- allows dynamic selection of algorithm versions by a client.

Relevant e.g., to yield-curve estimation using different algorithms.

Proxy pattern

- Handles situation where your application needs to communicate with remote objects (e.g., on another computer).
- Idea of pattern is to define local proxy for the remote object – more convenient for your application to use.
- Knowledge of how to invoke/interact with remote object is separated out into the proxy.



Structure of Proxy pattern

Proxy benefits and costs

- Simplifies your code, separating remote invocation code from main processing.
- Increases modularity.
- If remote protocol changes, only Proxy code needs to change.

Costs: one extra method call per communication.

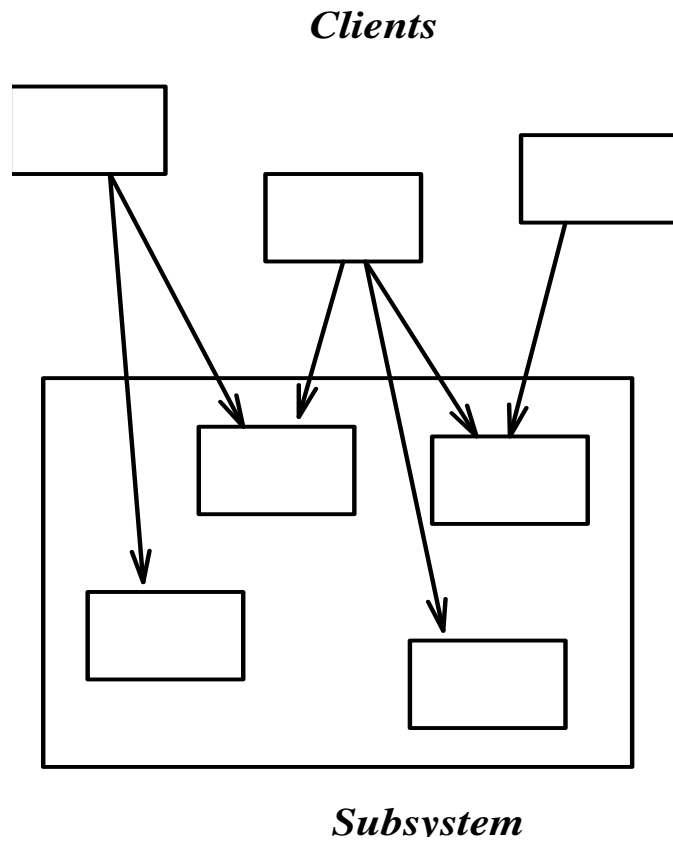
Proxy pattern

- Application requests service from Proxy object, e.g., to buy a quantity of a share.
- Proxy handles the details of how request is implemented, e.g., by constructing FIX (Financial information exchange) message and communicating with an exchange to make the purchase, using FIX protocol.

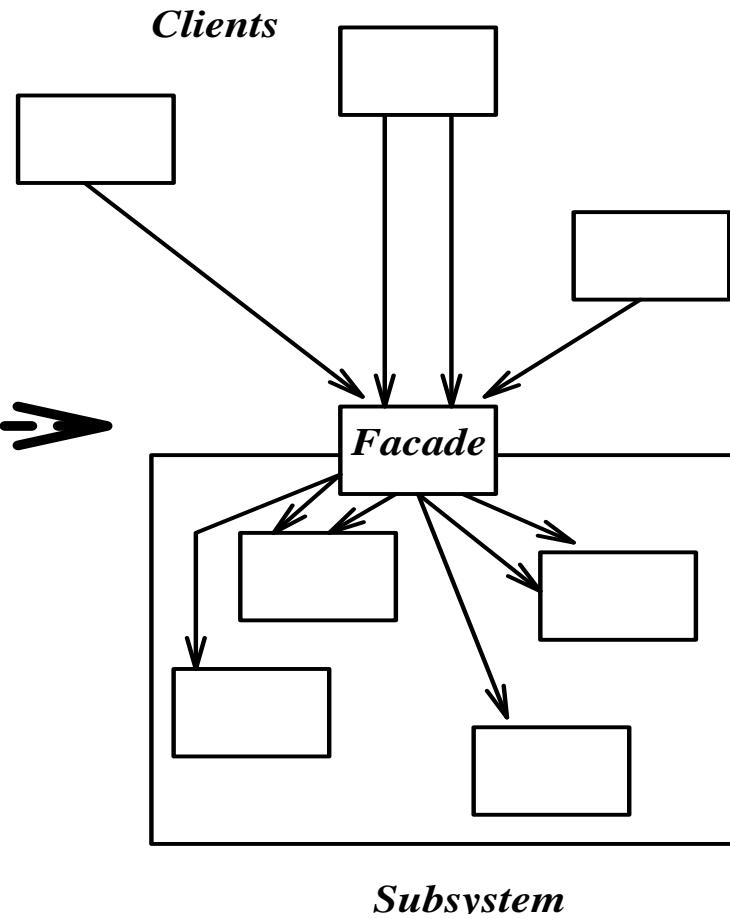
Facade pattern

- If a group of classes is used by different clients, bundle up functionalities into a separate Facade class which hides internal details of group
- Facade provides operations for external clients
- Typical use – facade for functional co-ordinator components in architectures, to provide operations invoked from user interface(s)

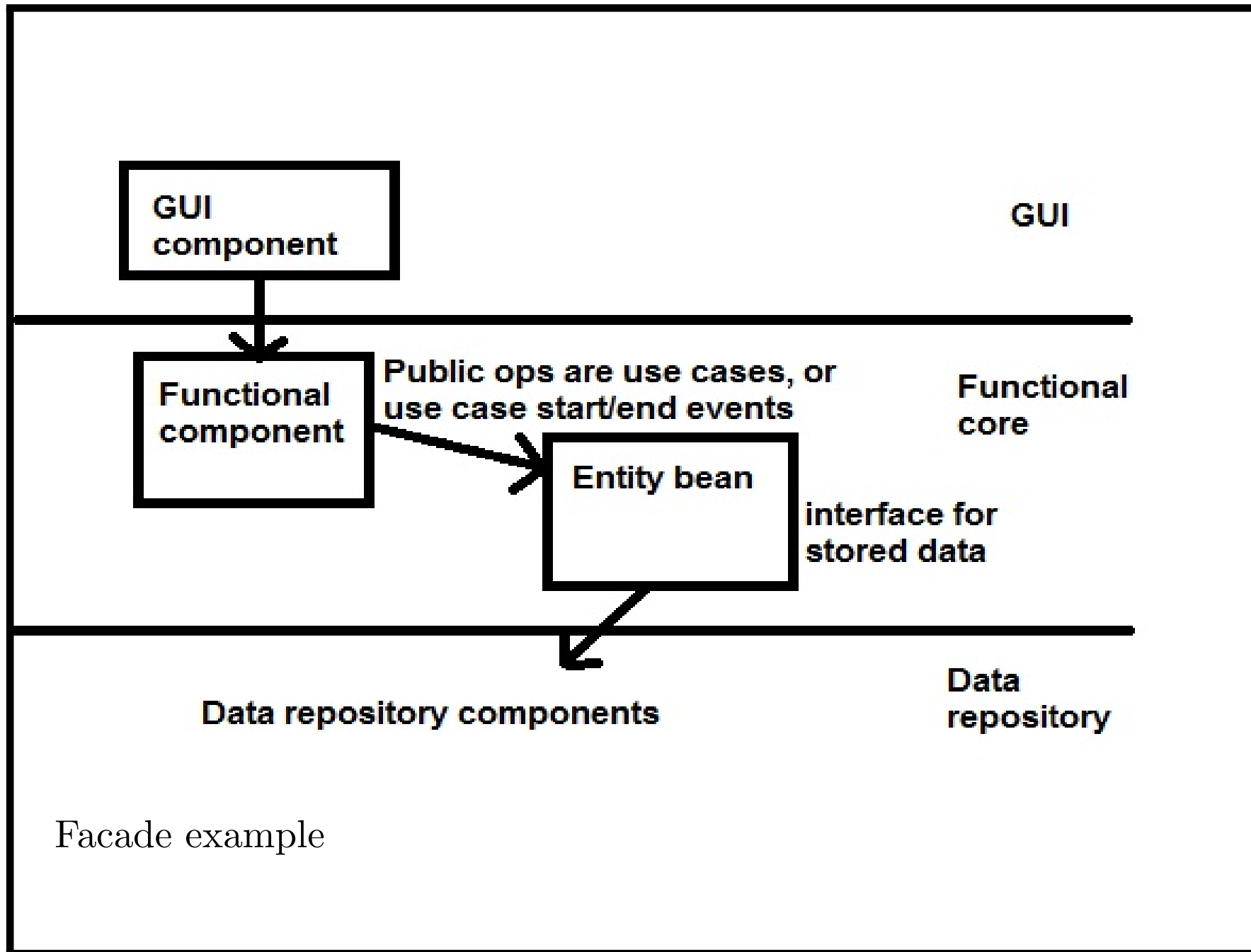
Old System:



New System:



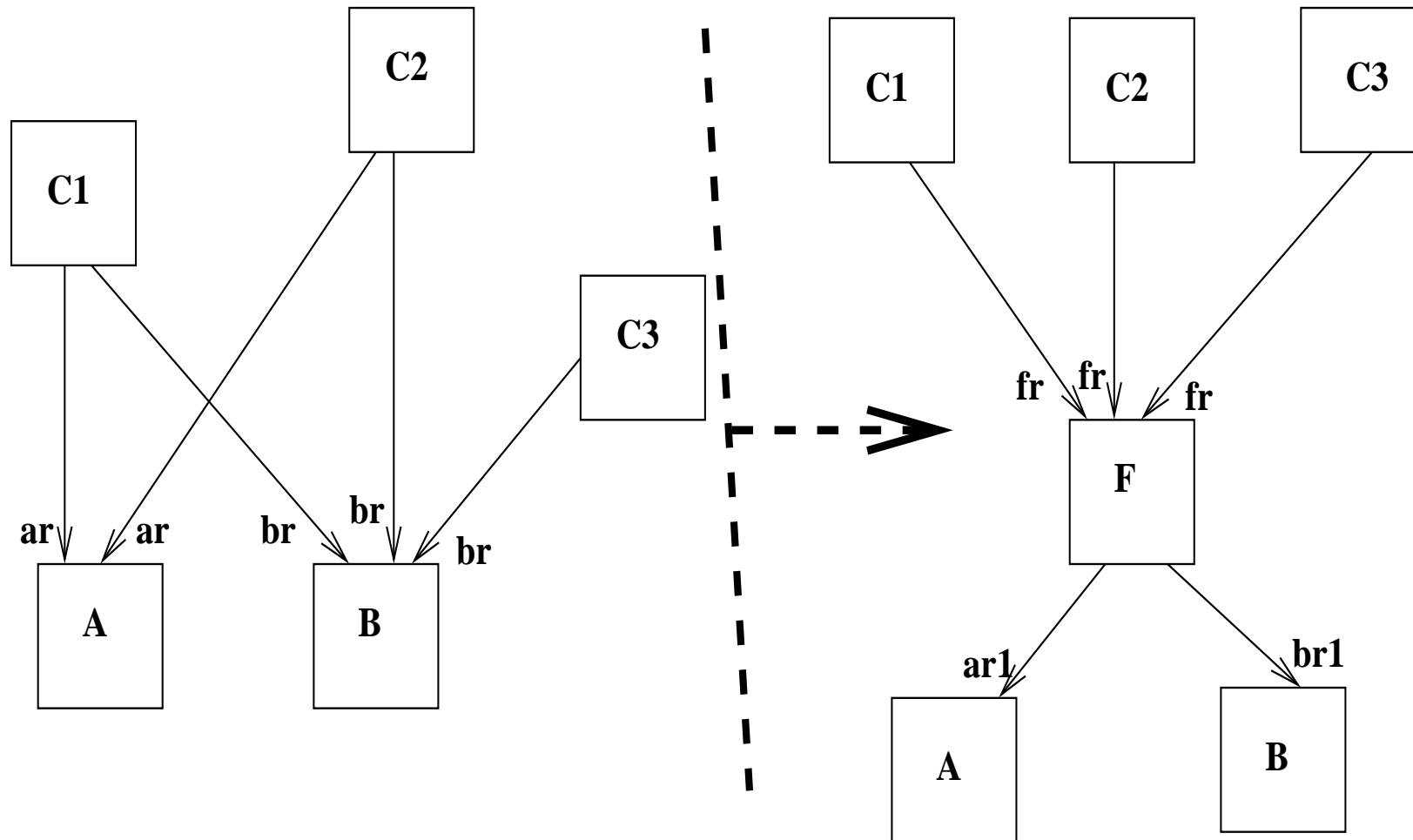
Facade pattern



Benefits/issues of Facade pattern

- Reduces complexity of clients – they only interact with facade class
- Hides internal complexity of subsystem behind the facade
- Changes to subsystem need not affect clients, only the facade – the facade insulates clients from internal subsystem changes.

Facade class is a 'functional class', a group of operations. Termed an 'interactor' in clean architecture.



Introducing Facade

Design refactoring

- Code can have quality flaws/‘code smells’
- Impair maintenance, lead to higher costs
- Agile development can lead to *technical debt* – code flaws introduced because of time pressures to produce working code fast
- Tools such as SonarQube and Pylint can be used to detect technical debt/flaws
- Code refactoring used to remove flaws.

Design refactoring

<i>Flaw</i>	<i>Refactoring</i>
Excessive operation parameters (EPL)	Group related parameters into a new object
Duplicated code (DC)	Factor out cloned code into new operation
Unused local variables (UVA)	Remove variable
‘Magic numbers’ in code (MGN)	Replace by named constants

Design refactoring

Parameter bundling: replace

```
op(x : X, y : Y, p1 : T1, p2 : T2, p3 : T3)  
... code of op ...
```

by new class

```
class P {  
    attribute p1 : T1;  
    attribute p2 : T2;  
    attribute p3 : T3  
}
```

and revised operation:

```
op(x : X, y : Y, p : P)  
... code of op: pi replaced by p.pi ...
```

Design refactoring

Clone removal: replace

```
... code of op1 ...  
s1; s2;
```

```
... code of op2 ...  
s1; s2;
```

by new operation + refactored code:

```
op(...)    activity: s1; s2;
```

```
... code of op1 ...  
self.op(...);
```

```
... code of op2 ...  
self.op(...);
```

Design refactoring

May also be performance/efficiency flaws:

<i>Flaw</i>	<i>Refactoring</i>
Recursion	Replace tail recursion by iteration
Repeated function computation	Cache function results
Repeated data lookup	Cache data when constant

Design refactoring

Replace tail recursion:

```
operation op(x : T) : S
activity: ...;
    return self.op(e);
```

by:

```
operation op(x : T) : S
activity:
    while (true)
    do (...;
        x := e);
```

and similar cases of tail recursion.

Design refactoring

Replace uncached calls of expensive function:

```
operation fact(x : int) : long
```

```
activity:
```

```
    result = Integer.subrange(1,x)->prd();
```

by cache *fact_cache* : *Map(int, long)* and calls of:

```
operation fact_cached(x : int) : long
```

```
activity:
```

```
    if x in fact_cache
```

```
    then return fact_cache[x]
```

```
    else
```

```
        (var y := fact(x);
```

```
            fact_cache[x] := y;
```

```
            return y);
```

QuantLib

- Open source C++ library for quantitative finance
- Launched in 2000 and widely used
- Open source has advantages over proprietary code (e.g., Excel): developers can see precisely what computations are being performed.

QuantLib modules

- Numeric types: synonyms such as *Real*, *Time*, *Rate* for *double*
- Currencies, FX rates: predefined national currencies and management of exchange rates
- Design patterns: templates for Factory, Singleton, Observer, etc
- Date and time: dates, calendars, day counts and date calculations.
- Math tools: random number generators, root finders, optimisation algorithms.
- Monte Carlo: framework for Monte Carlo simulations.
- Cash flows: classes + functions for cash flows and cash flow sequences.
- Term structures: framework for defining yield term structures.

- Financial instruments: wide range of different contract models.
- Pricing engines: pricing algorithms for many kinds of financial instrument.
- Volatility models
- Stochastic processes.

Design patterns in QuantLib

- *Factory*: a class which produces instances of other classes, e.g., a factory for different varieties of financial options.
- *Singleton*: used for repositories of data that are used in several places in code. E.g., a repository holding market data points for interest rates, or currency exchange rates.

Repository :: *instance()* returns the instance of singleton class *Repository*.

- *Observer* used in any situation where some objects may need to be notified when another object changes state.

E.g., bond valuation object that depends upon a yield curve: valuation must be recalculated if yield curve changes.

QuantLib *Quote* class is an Observable (Subject), and also an Observer (View).

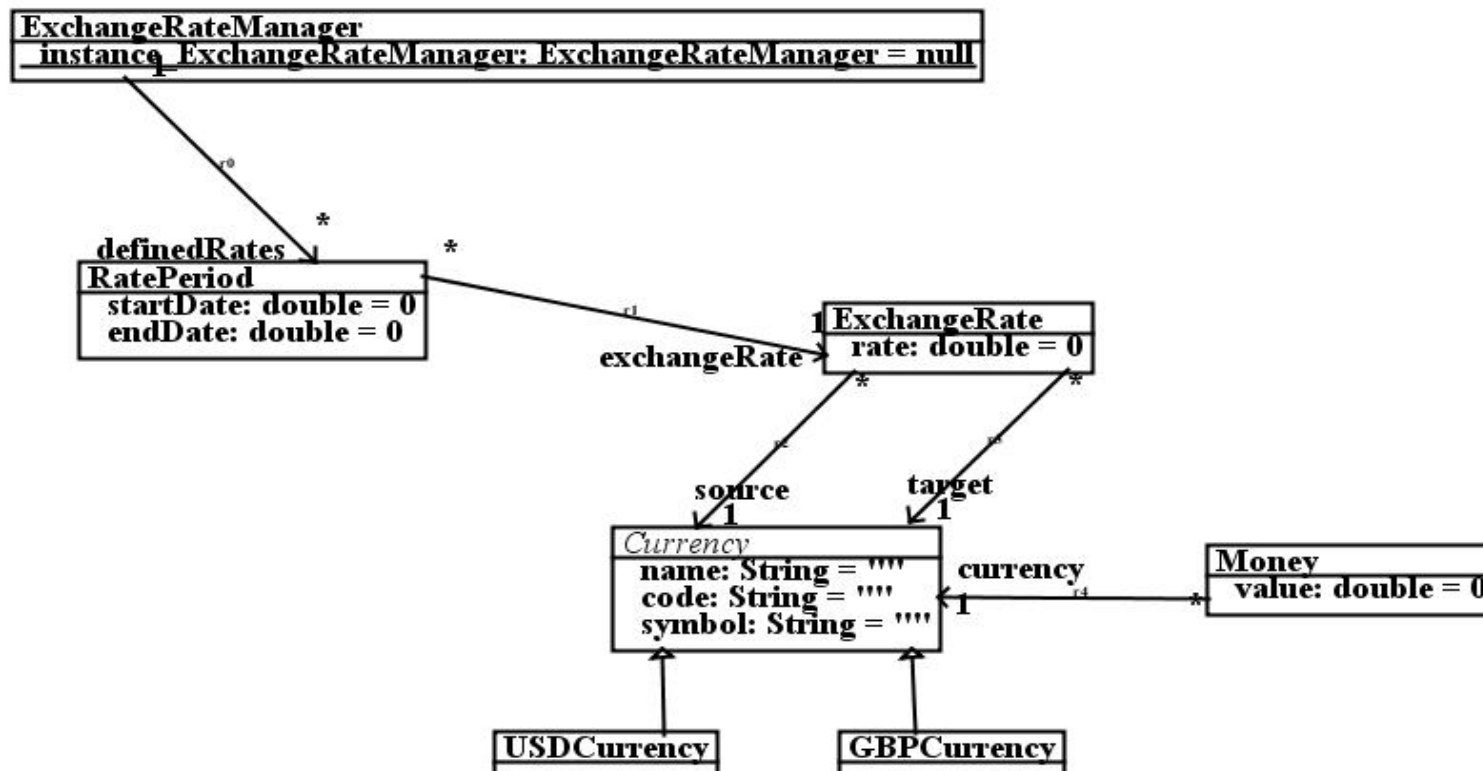
Currencies and exchange rates in QuantLib

The key classes are:

- *Currency* – has *name* : *String*, such as “Euro”, *code* : *String* for 3-letter code, e.g., “EUR”, and *symbol* : *String* for currency symbol, e.g., “£” for GB pound.
- Predefined currency subclasses for most national currencies, e.g., *USDCurrency* for US dollars, *GBPCurrency* for GB pounds.
- *Money* – a pair of a currency and *value* amount.

Currencies and exchange rates in QuantLib

- *ExchangeRate* – associates *rate* to pair of currencies, a *source* (conversion from) currency, and a *target* (conversion to) currency. Provides *exchange*(*m* : *Money*) : *Money* operation to map amount *m* in one currency to corresponding amount in the other. Exchange rates can be chained.
- *ExchangeRateManager*, this is a *Singleton* class – stores exchange rates together with a time period for validity of the rate. Period defined by start date and end date.



FX classes in QuantLib

Summary of Part 4

- Design architecture diagrams
- Design patterns
- Design refactoring
- Design patterns in Quantlib