PART 4: Software Design using UML

- Architecture diagrams
- Design smells and Refactoring
- Design patterns
- Libraries and reuse.

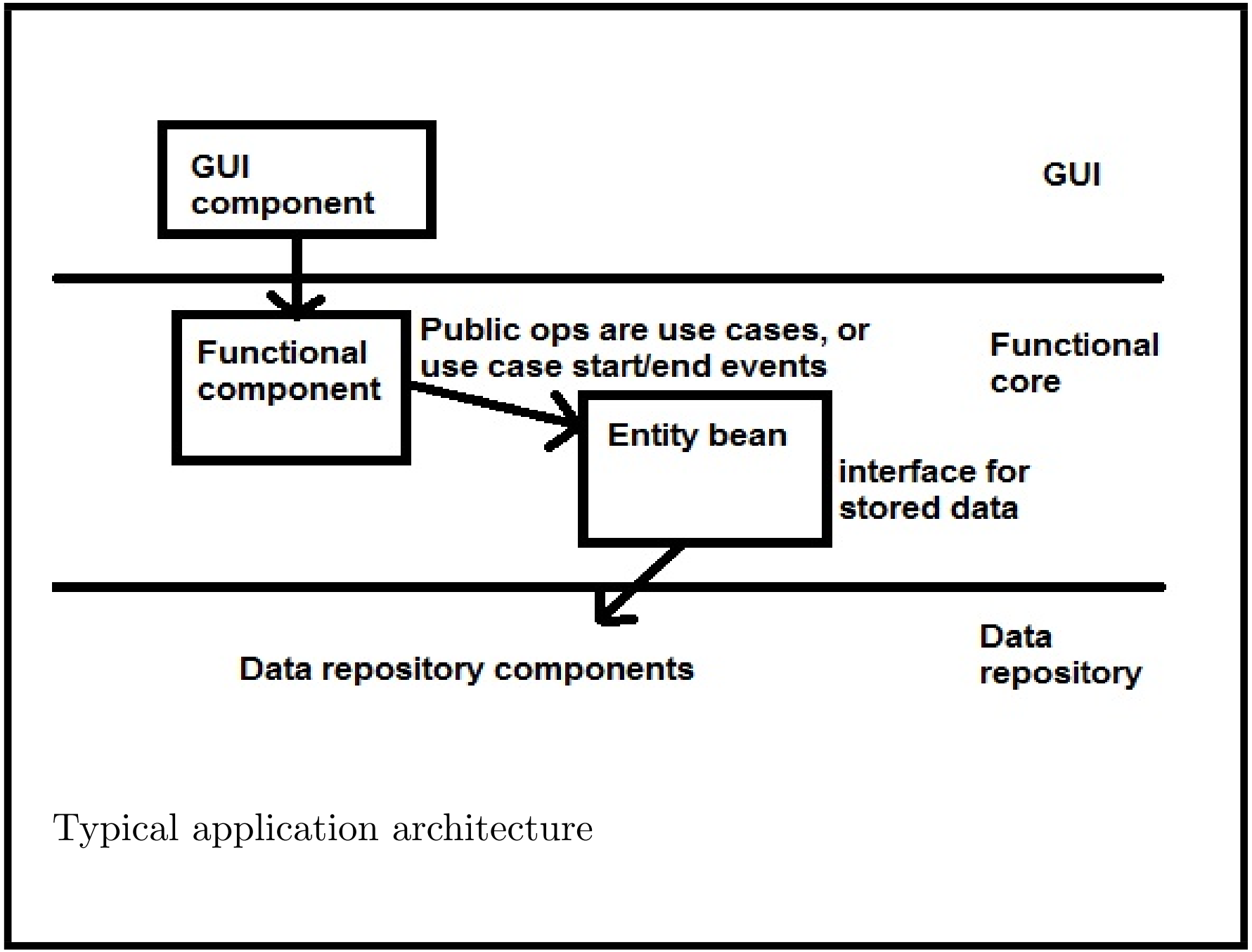
Chapters 9, 10 of main textbook relevant for this part.

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 (eg, by constraints).
- Modules may be single classes, or group of closely related classes.

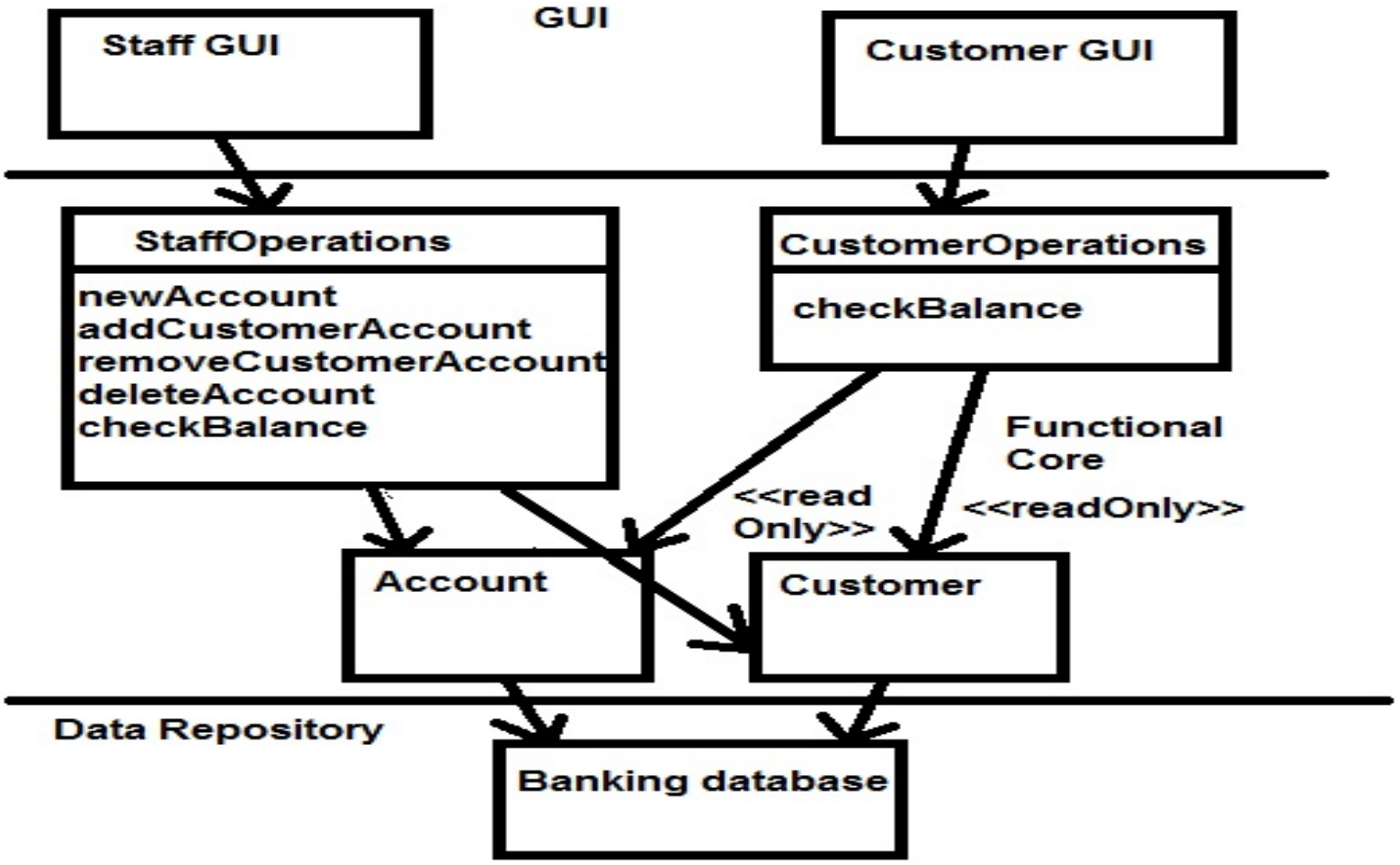


Typical application architecture

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 constraints/functionality satisfied by design of an individual module.
- Other constraints/functionalities will need complex design decisions, + interaction of several modules.

Eg., $checkOwnBalance(cId, aId)$ needs to read both *Account*, *Customer* data.



Banking system architecture

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*: Module operations correspond to use cases, accessed via UIs for actors linked to cases. Eg., *StaffOperations*.

Module design

- Module specification is only definition of module which other modules should rely on. Other modules should not directly use internal implementations of a module.
- This property, of 'loose coupling' between modules, enables modules to be developed and refined relatively independently, allows system to be separated into 'layers'/tiers, insulated from changes to other layers.

Design smells

As with code, UML models can have quality flaws such as:

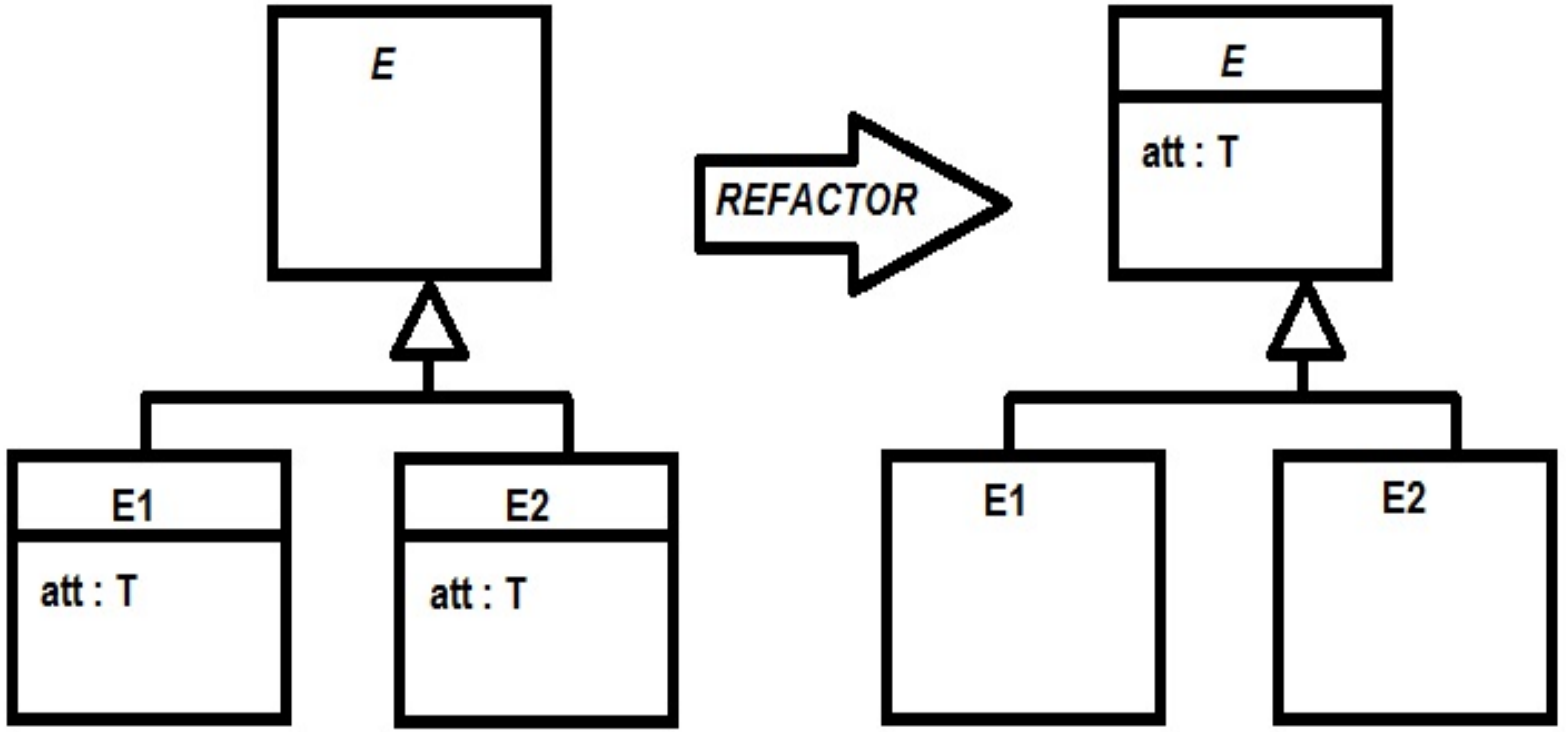
- *God Class*: one class carries out most of system functionality, others are auxiliary
- *Excessive Class Length*; *Excessive Method Length*; *Excessive Inheritance use*
- *Excessive Parameter List* (eg., more than 10 parameters in an operation)
- *Duplicate Code*
- *Cyclomatic Complexity* (number of logical loop/if conditions in an activity)
- *Too Many Methods* (eg, more than 20 in a class)
- *Too Many Fields* (eg., more than 20 attributes/roles).

Design smells and refactoring

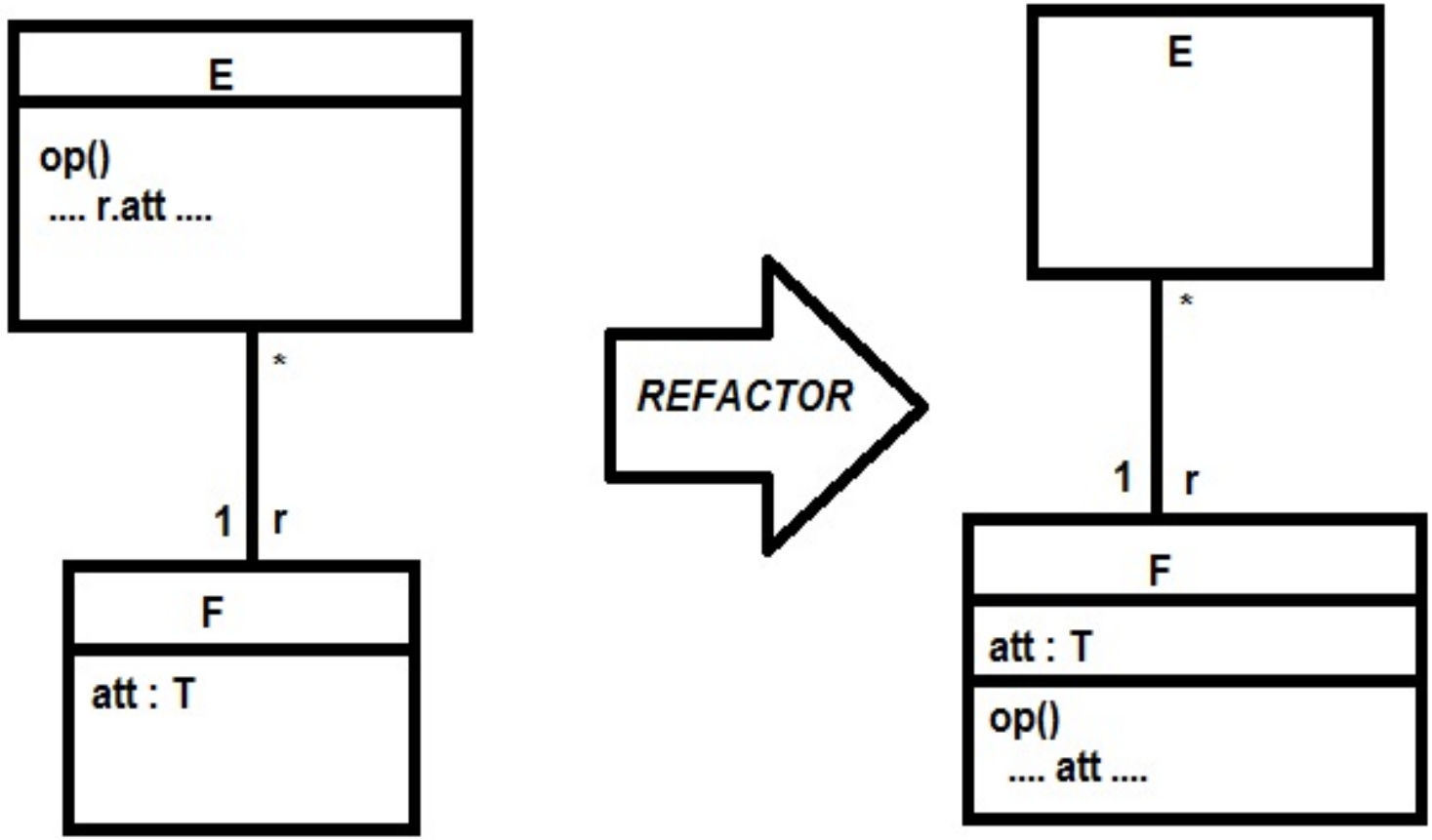
- Code/design smells makes maintenance of the system more expensive, increases likelihood of functional errors
- Refactoring used to remove these flaws + improve structure
- Eg., removing duplicated attributes in sibling classes; factoring out part of an operation which belongs in a client class.

Refactoring of models

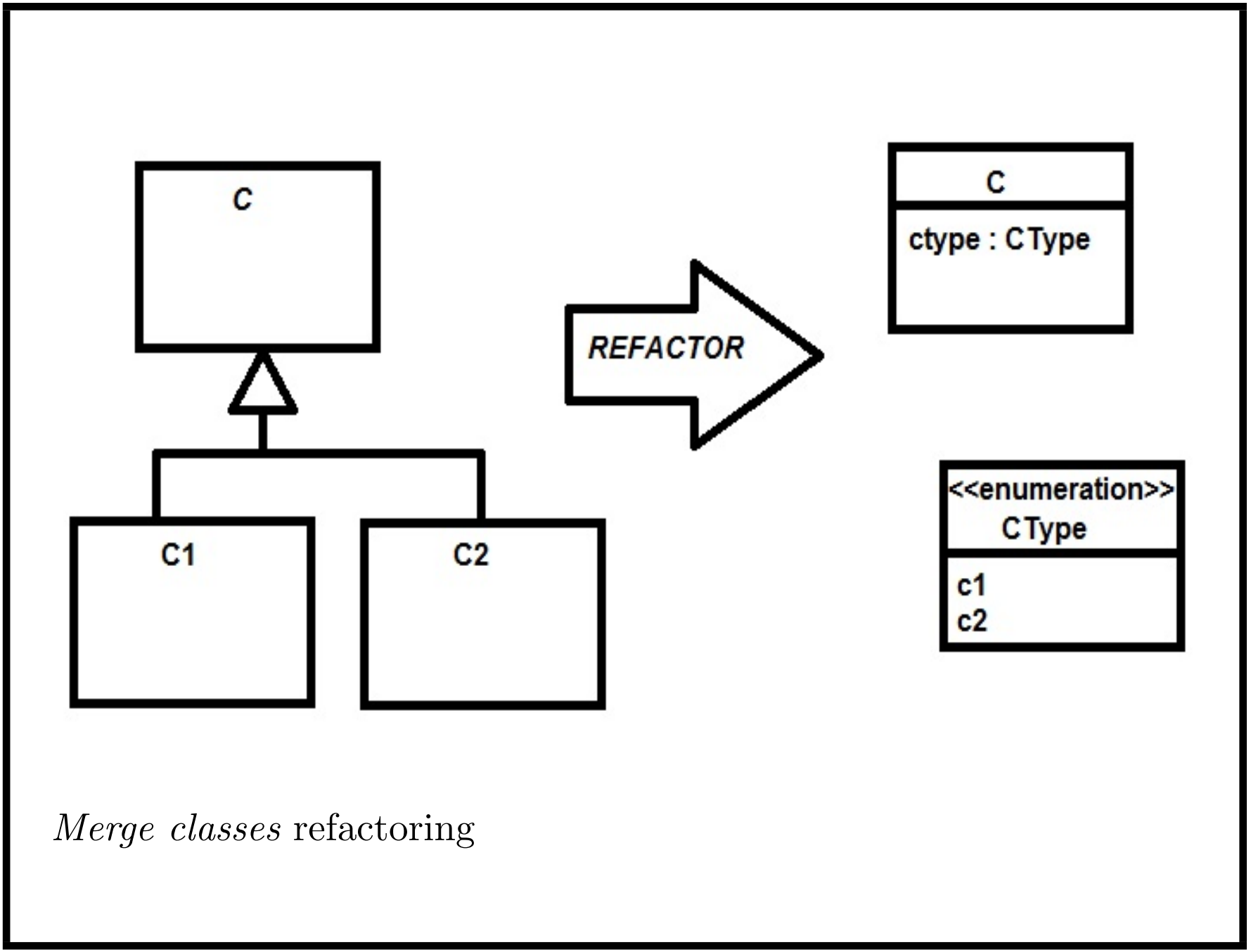
- *Refactoring*: modifying model to improve its structure, whilst keeping same semantics.
- Class diagrams can be refactored to improve structure, remove redundancies + improve correspondence to requirements:
  - "Pull up attribute" refactoring: if all (at least 2) direct subclasses of class E declare attribute $att : T$, replace by single definition in E
  - "Move operation" refactoring: if operation op of E refers to attributes/roles of class F via association E—F, try moving op to F
  - "Merge classes": if all subclasses of abstract class C are empty, replace by flag attribute of C of enumeration. Make C abstract.



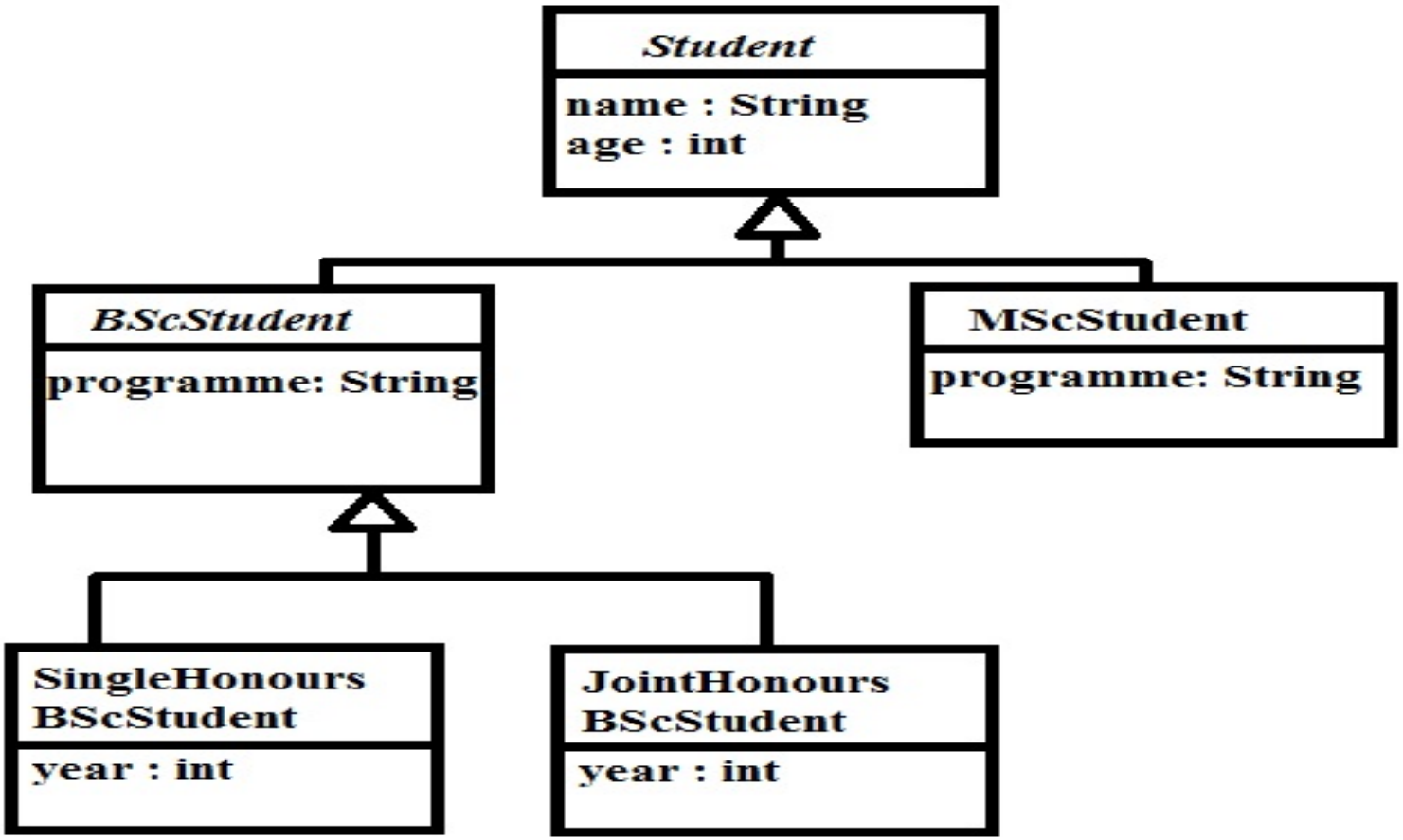
Pull up attribute refactoring



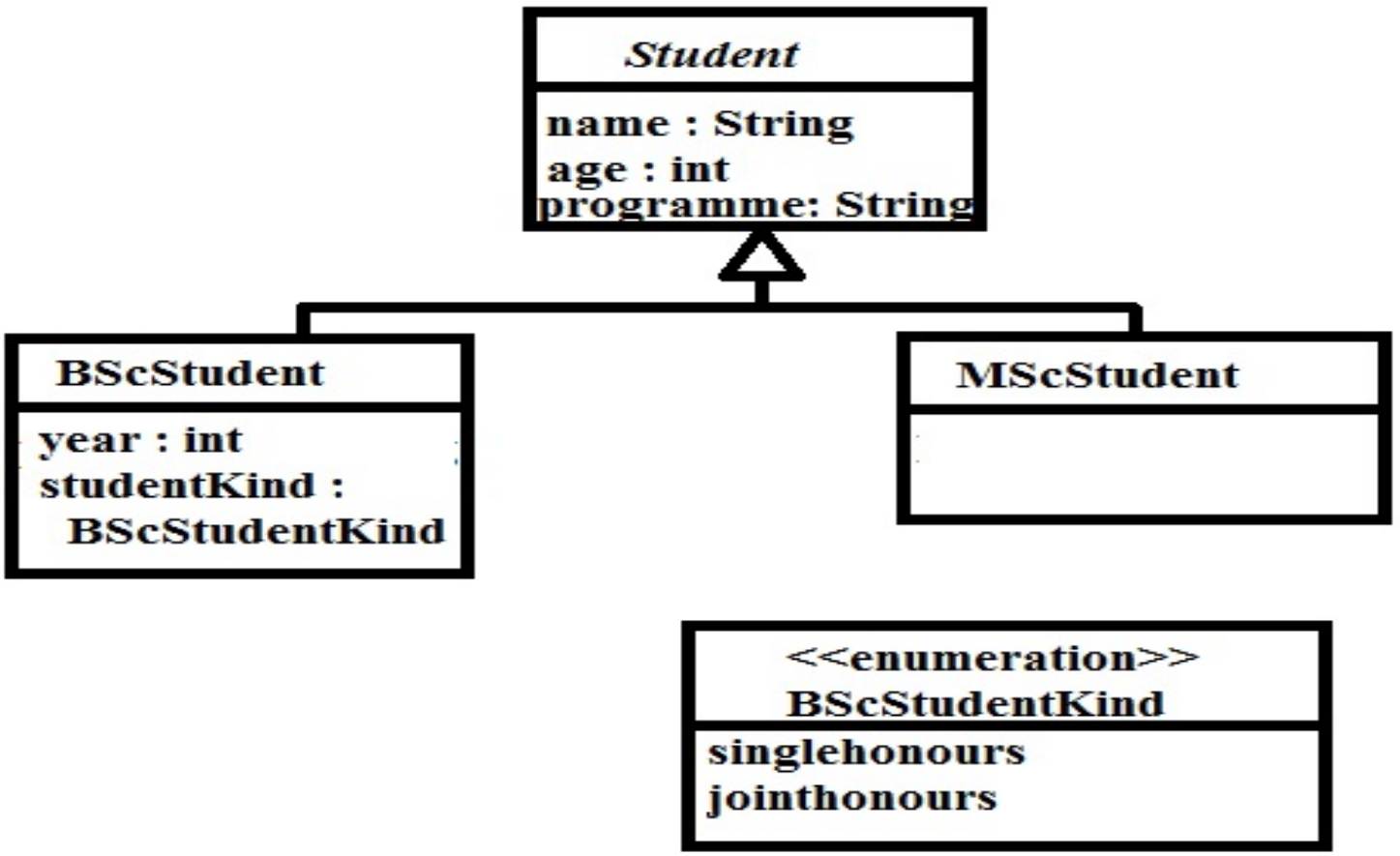
Move operation refactoring



Merge classes refactoring



Example: Student classes



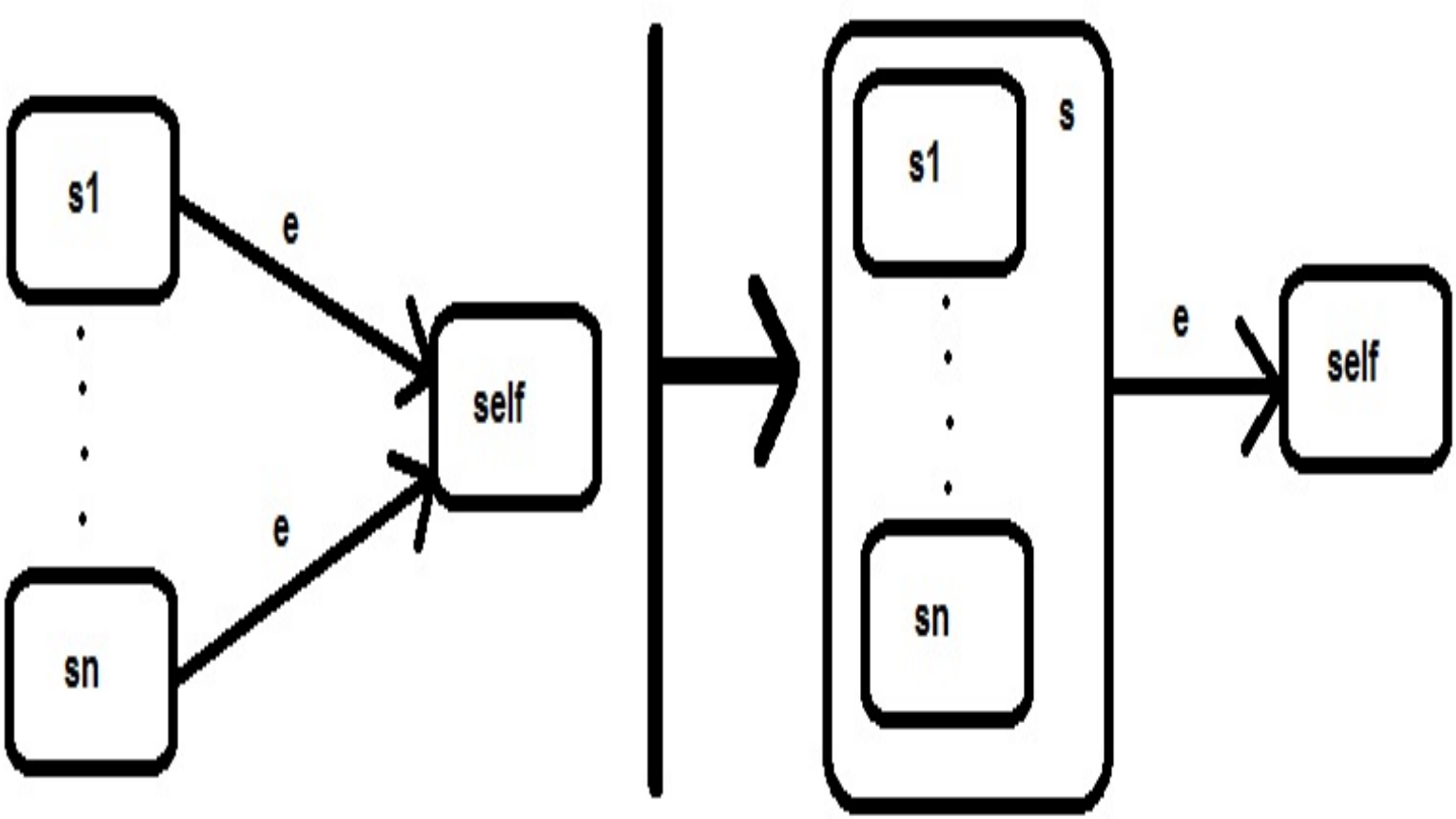
Refactored Student classes

State machine refactoring

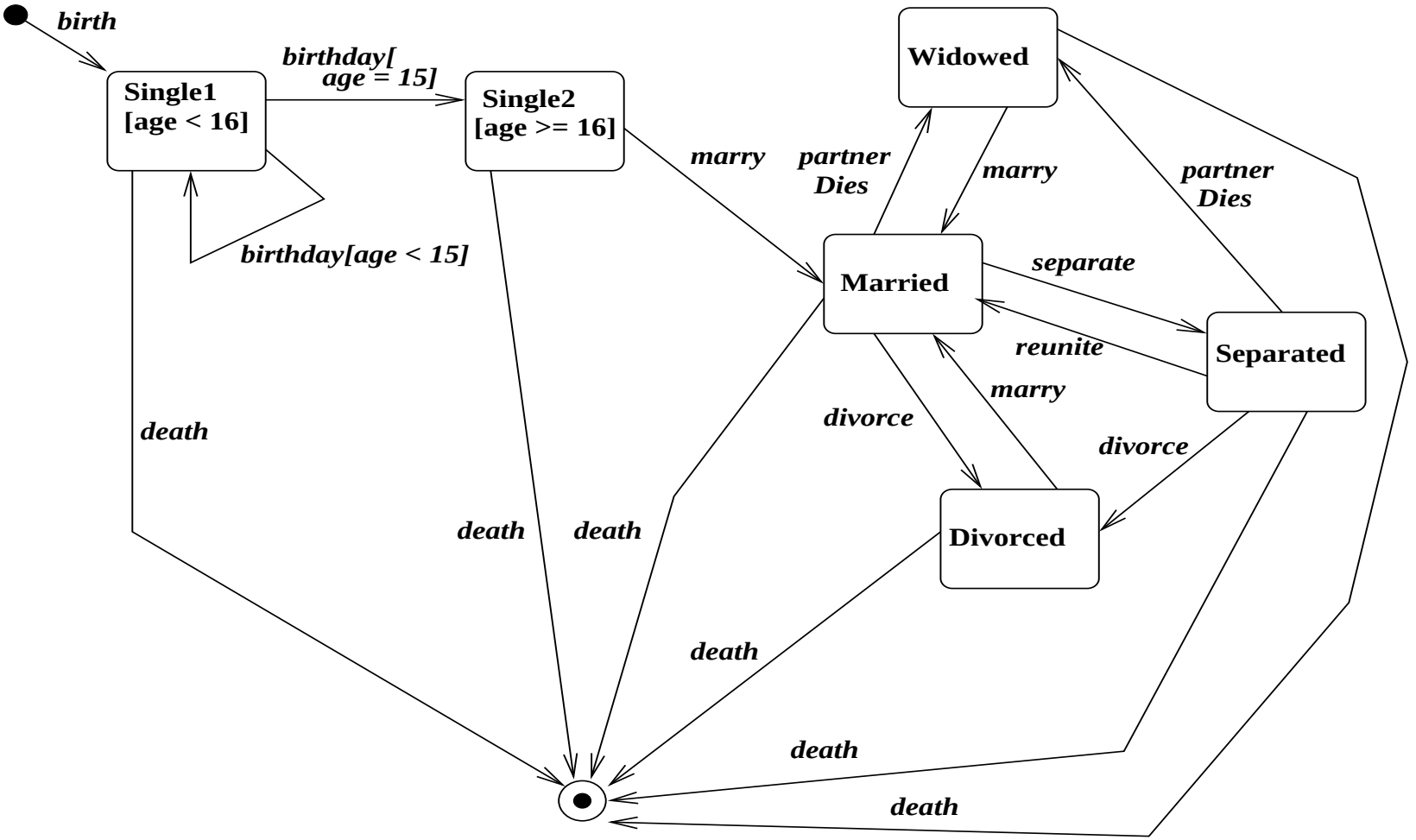
We've seen refactoring example of merging states that have same behaviour:

- *Introduce composite state*: if group of states $s1$, ..., sn (n > 1) all have identical outgoing transitions to states not in group, then create new superstate s of $s1$, ..., sn, + replace common outgoing transitions by transitions from s.

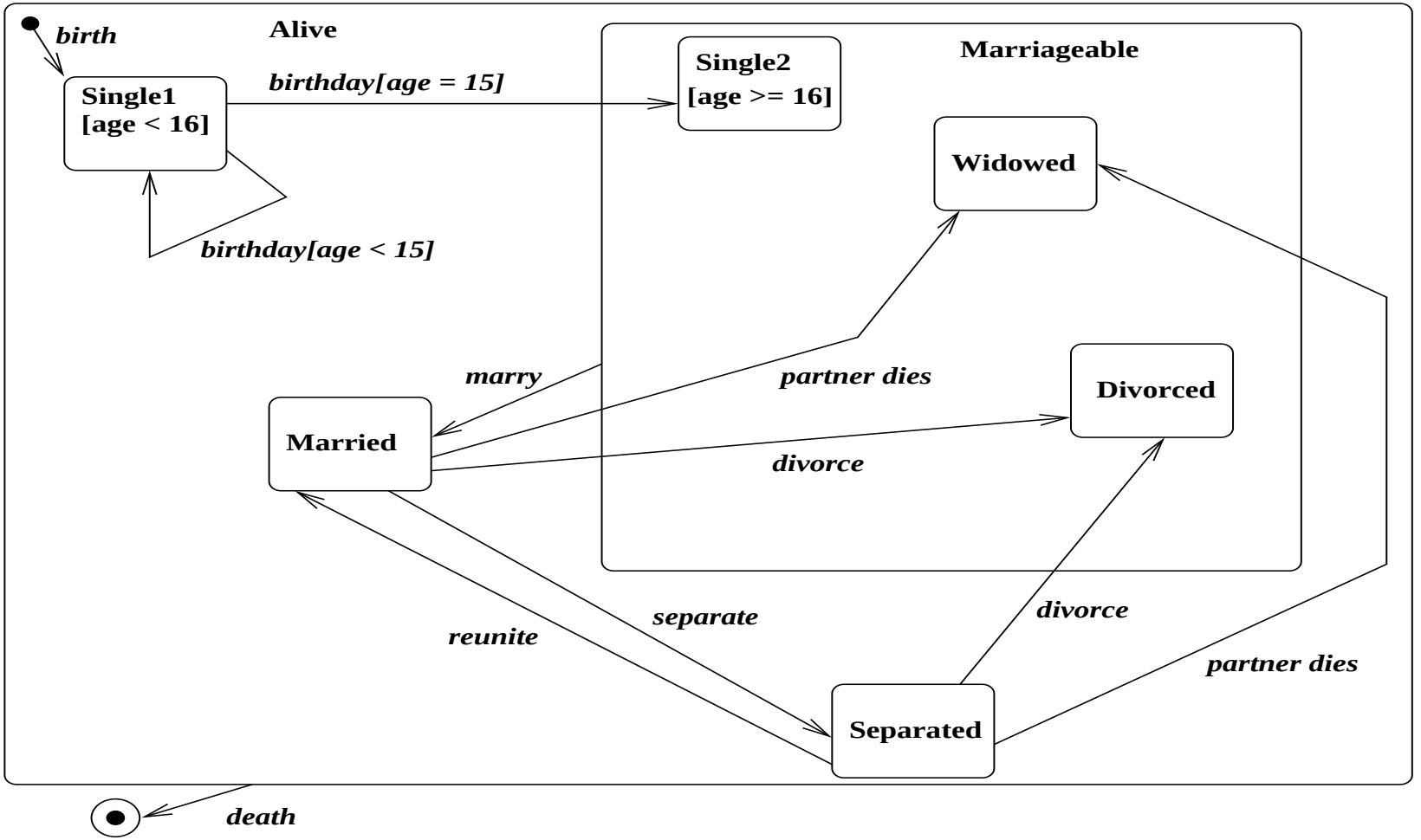
Reduces complexity of diagram, helps to identify commonalities between states.



Introducing a composite state



Marital status state machine



Refactored marital status state machine

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: *The Timeless Way of Building*, 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. Eg: *Singleton*, *Builder*

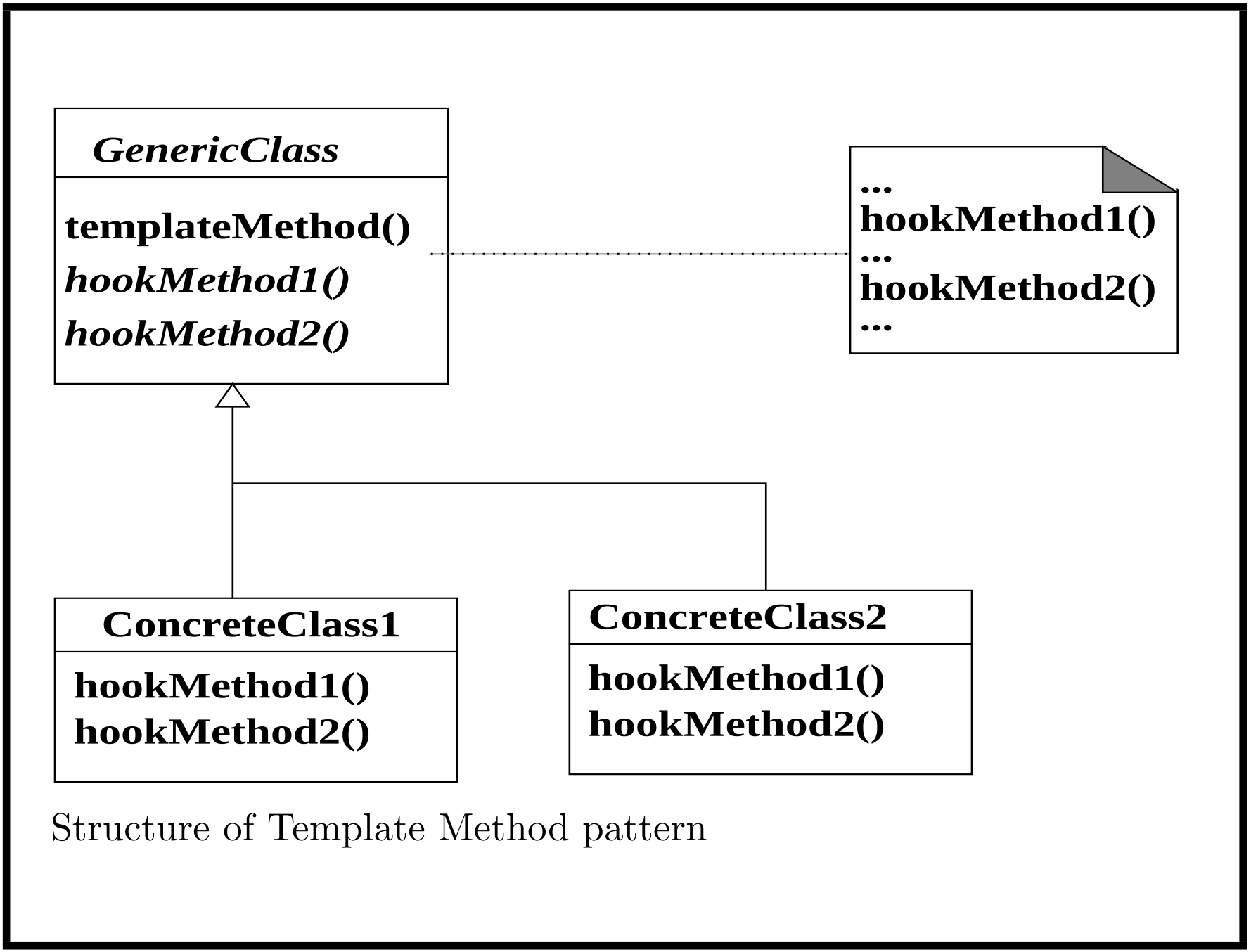
Behavioural: Organise distribution of behaviour amongst objects. Eg: *Template Method*, *Iterator*, *Observer*, *Strategy*

Structural: Organise structure of classes and relationships. Eg: *Adapter*, *Facade* (not considered here).

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

Template Method pattern

- Behavioural pattern used to organise/refactor design, if a method m has essentially same algorithm in two or more classes, with small variations.
- Pattern defines superclass G for classes of m, puts generic definition of m in G, subclasses only define parts of m in which they differ.
- *Benefits*: removes duplicated code; reduces code size; improves extensibility.
- *Disadvantages*: fragments the code; more method calls; requires run-time polymorphism.



Structure of Template Method pattern

Template Method pattern

Classes involved in pattern are:

- *GenericClass* – defines template method consisting of skeleton algorithm calling one or more hook methods. Also defines hook methods that subclasses override.
- *ConcreteClass* – implements hook methods.

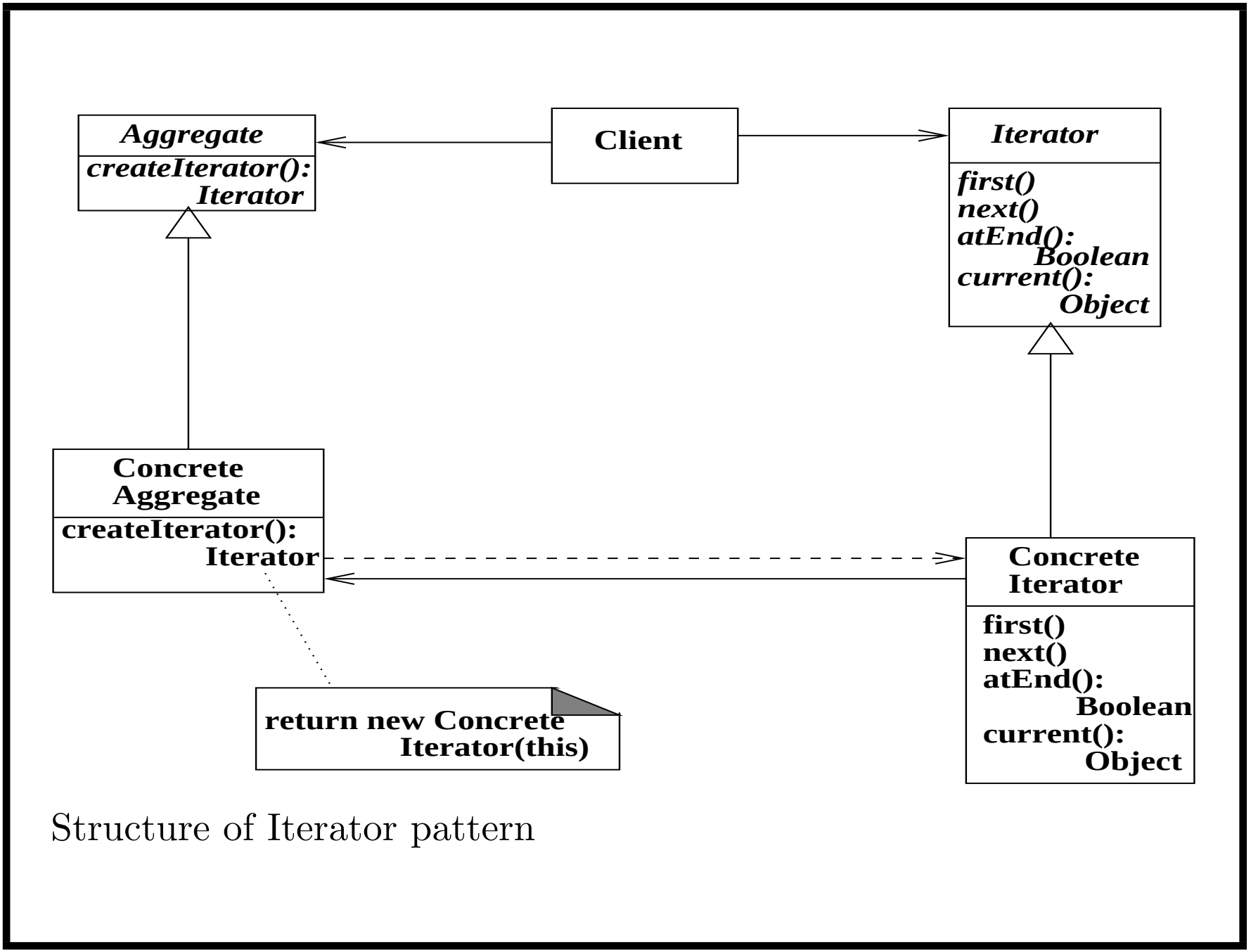
Hook methods may be defined as abstract in *GenericClass*, or given default implementations there.

The Iterator pattern

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

Applicable whenever we:

- require multiple traversal algorithms over an aggregate;
- 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, providing operations to move cursor forwards or backwards in structure.
- Normally *Iterator* class has operations such as *atEnd* / *hasNext* to test if iteration has reached end, *advance* / *next* to step forward to next element, and *current* to obtain current element.
- Iterator concept widely used – in Java, C#, C++. STL of C++ uses iterators to define many collection operations.

Classes involved are:

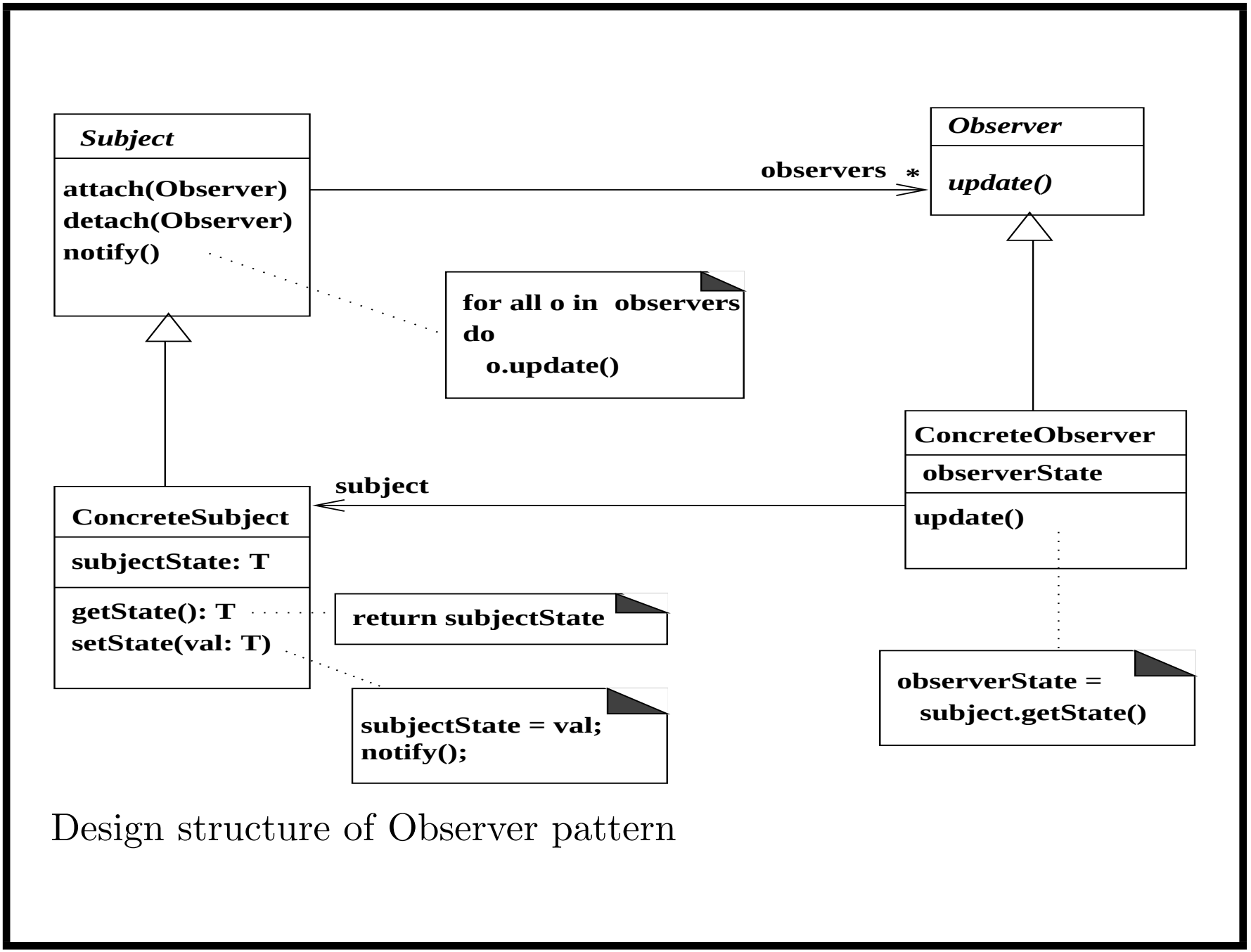
- *Aggregate* – class/interface defining general composite data structure such as list or tree.
- *ConcreteAggregate* – specific subclass defining particular data structure, eg., linked list or binary search tree.
- *Iterator* – interface for general iteration operations to start iteration, step through collection, etc.
- *ConcreteIterator* – iterator subclass specific to particular data structure. *createIterator*() method of data structure returns *ConcreteIterator* instance for structure.

Benefits/disadvantages of Iterator pattern

- Pattern increases flexibility, because aggregate and traversal mechanism are independent.
- Possible to have multiple iterators acting on same aggregate object simultaneously, possibly with different traversal algorithms: eg, using both post and pre-order traversal of a tree.
- If iterator pattern not used, would require direct access to private parts of data structures.
- However, pattern requires instead communication between objects – operation calls from one object to another.
- So we gain flexibility at cost of efficiency.

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

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

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

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

Benefits/disadvantages of Observer pattern

Positive consequences of using pattern:

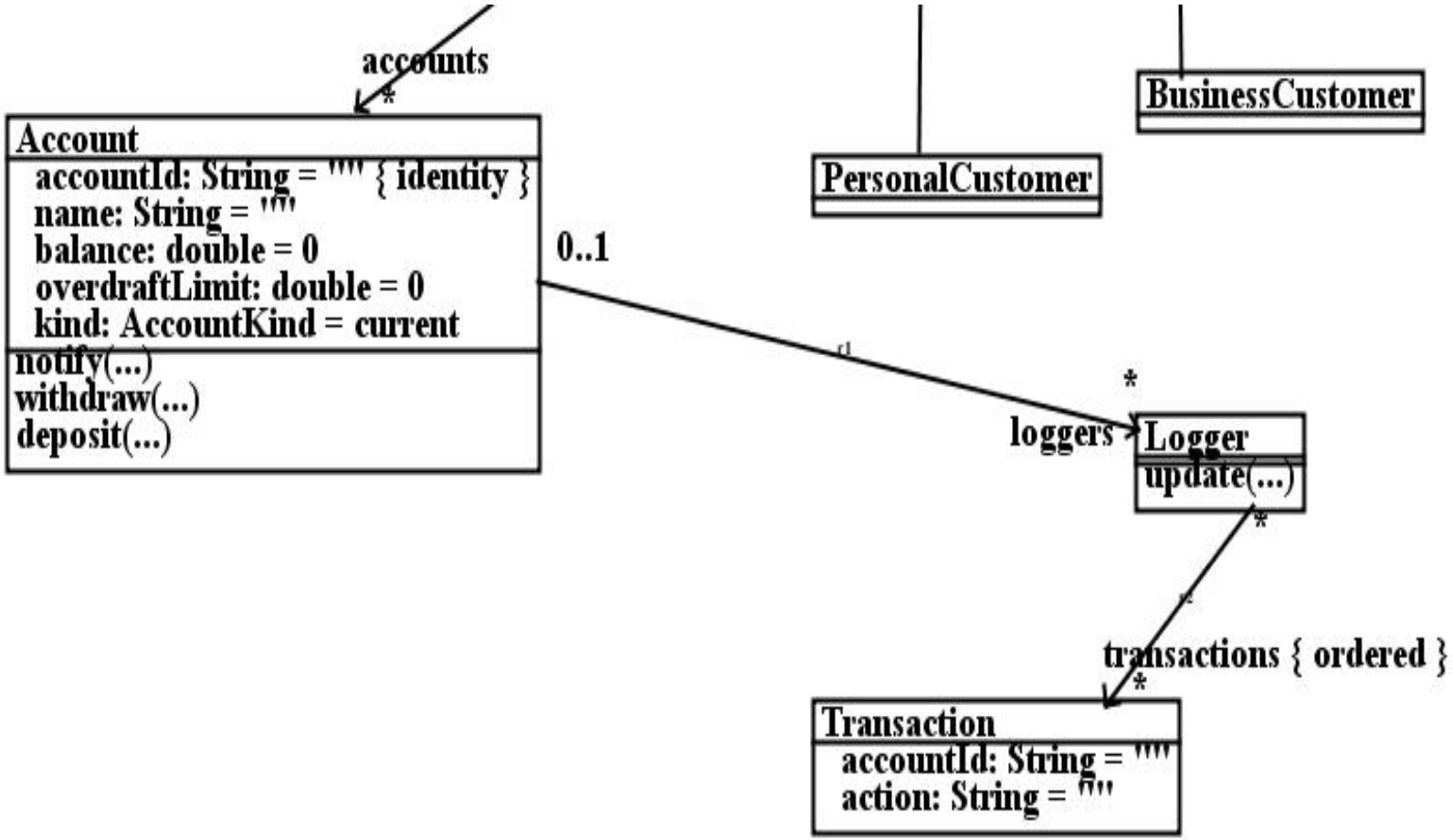
- Modularity: subject and observers may vary independently.
- Extensibility: can define and add any number of observers to given subject.
- Customisability: different observers provide different views of subject.

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* for *Account* (playing 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:

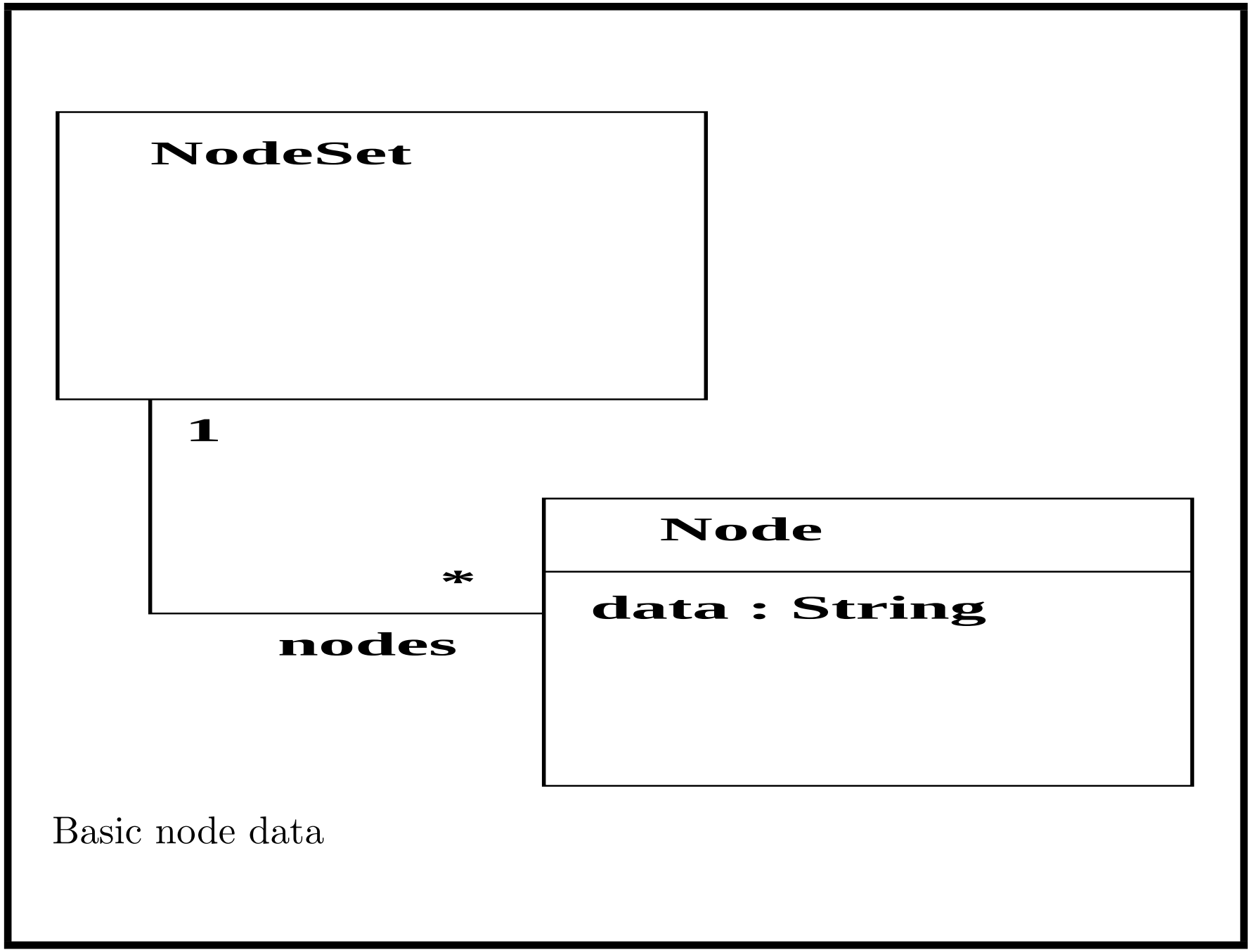
```
Account::
notify(s : String)
post:
  loggers->forAll( lg | lg.update(accountId, s) )

Logger::
update(id : String, s : String)
post:
  Transaction->exists( t | t.accountId = id &
                           t.action = s & t : transactions )
```

Example exam question

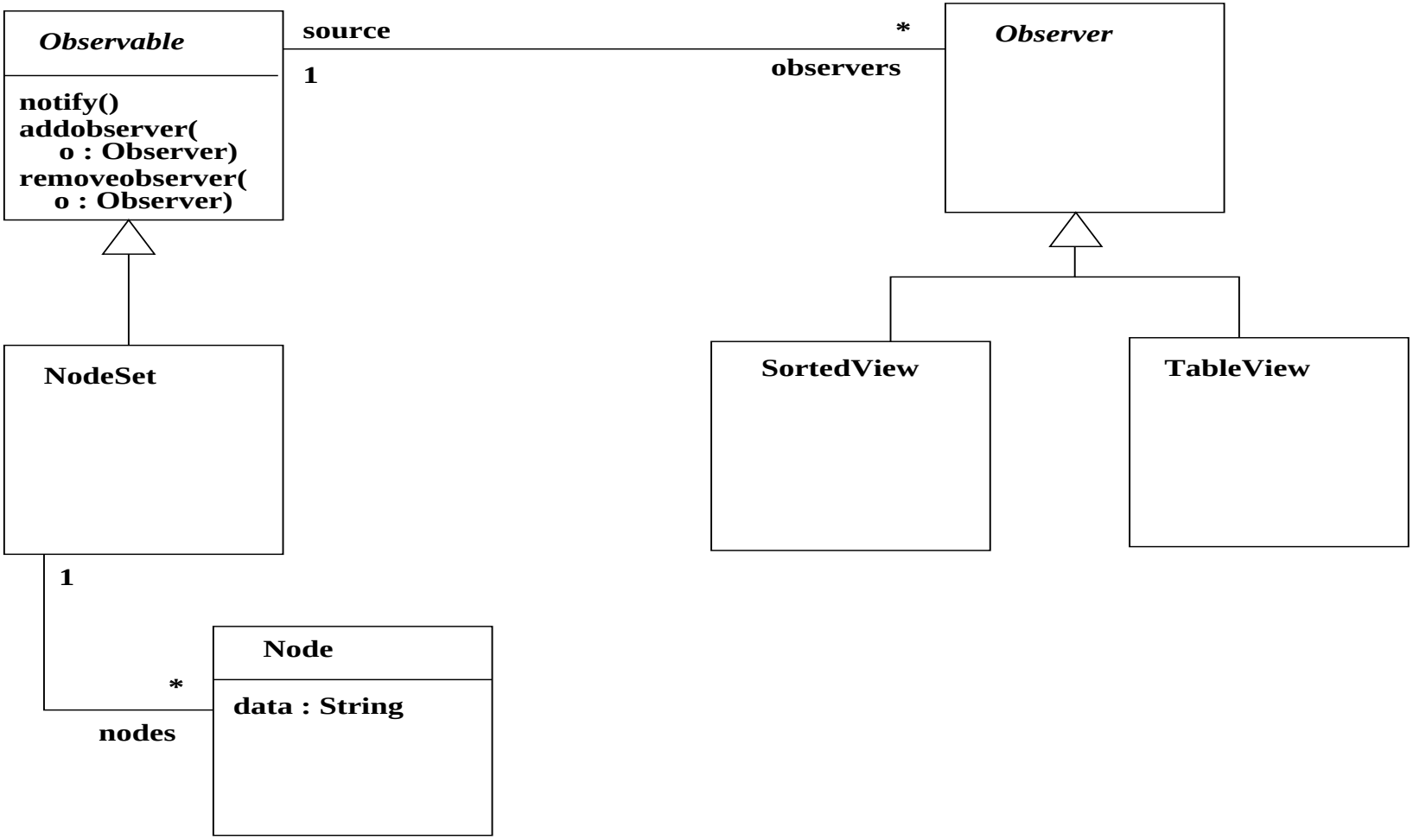
Apply the Observer pattern to construct a class diagram of a system, where data of *Node*s, organised into *NodeSet*s, is presented in two alternative ways.

A *SortedView* shows a list of the different *data* strings in sorted order, and *TableView* shows a table of the strings together with their frequency of occurrence (this is the number of times a particular *data* string occurs in *nodes*, divided by the number of *nodes*).

Show all attribute and operation declarations, but details of algorithms, or pre/postconditions are not necessary.



Basic node data



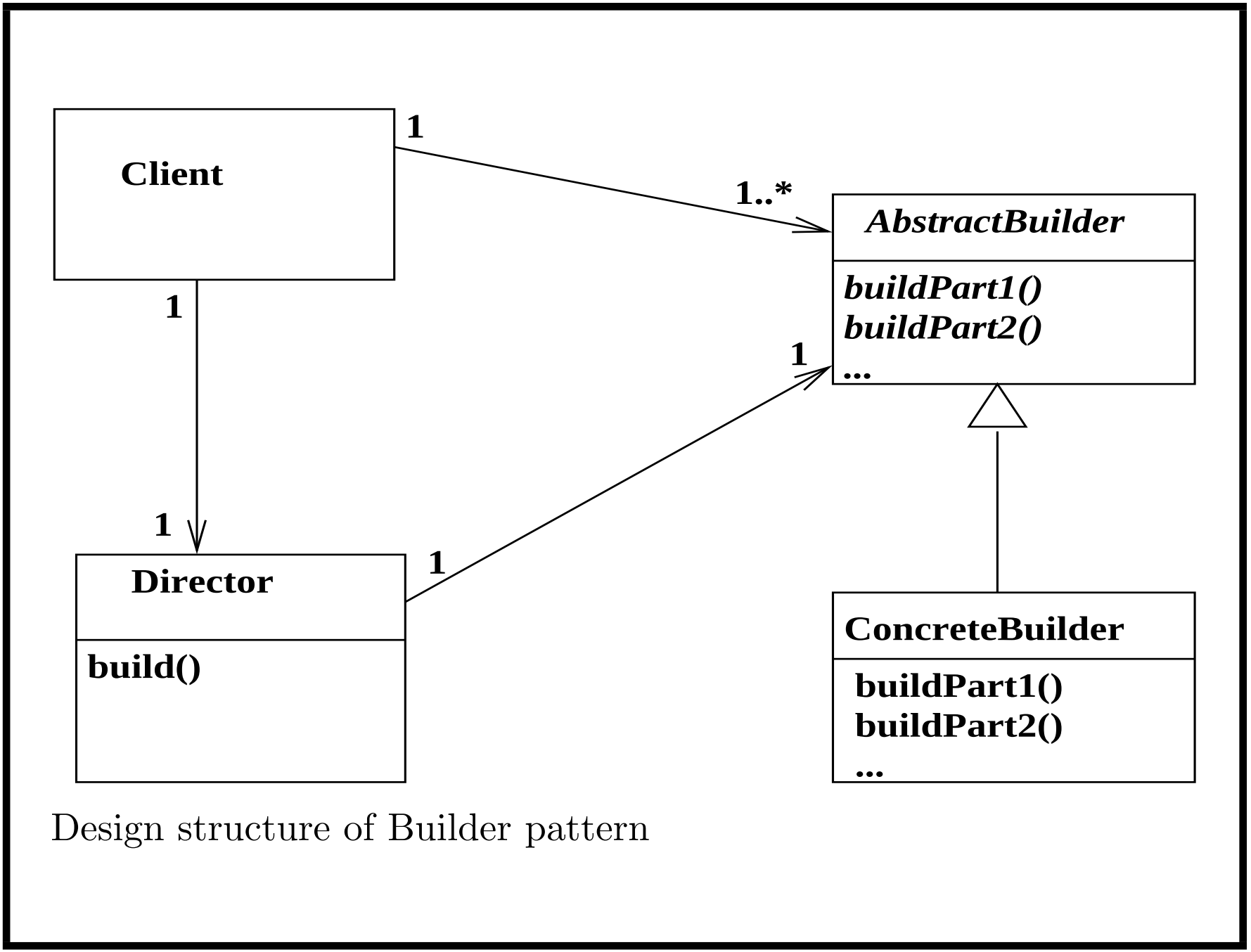
Solution: Observer pattern for nodes

Observer in programming languages

- Apart from MVC, many programming languages use Observer for GUI programming – Java Swing event model directly related to Observer, *Listener* objects play role of *Observer* for event source (*Subject*) objects.
- Other event production/consumption situations, such as Thread notification in Java, also use Observer event notification model.

The Builder pattern

- Creational pattern: allows a client object to construct complex objects by declarative specification of their type and content.
- Details of construction managed in separate classes, + different builder subclass for each variant of object to be constructed.



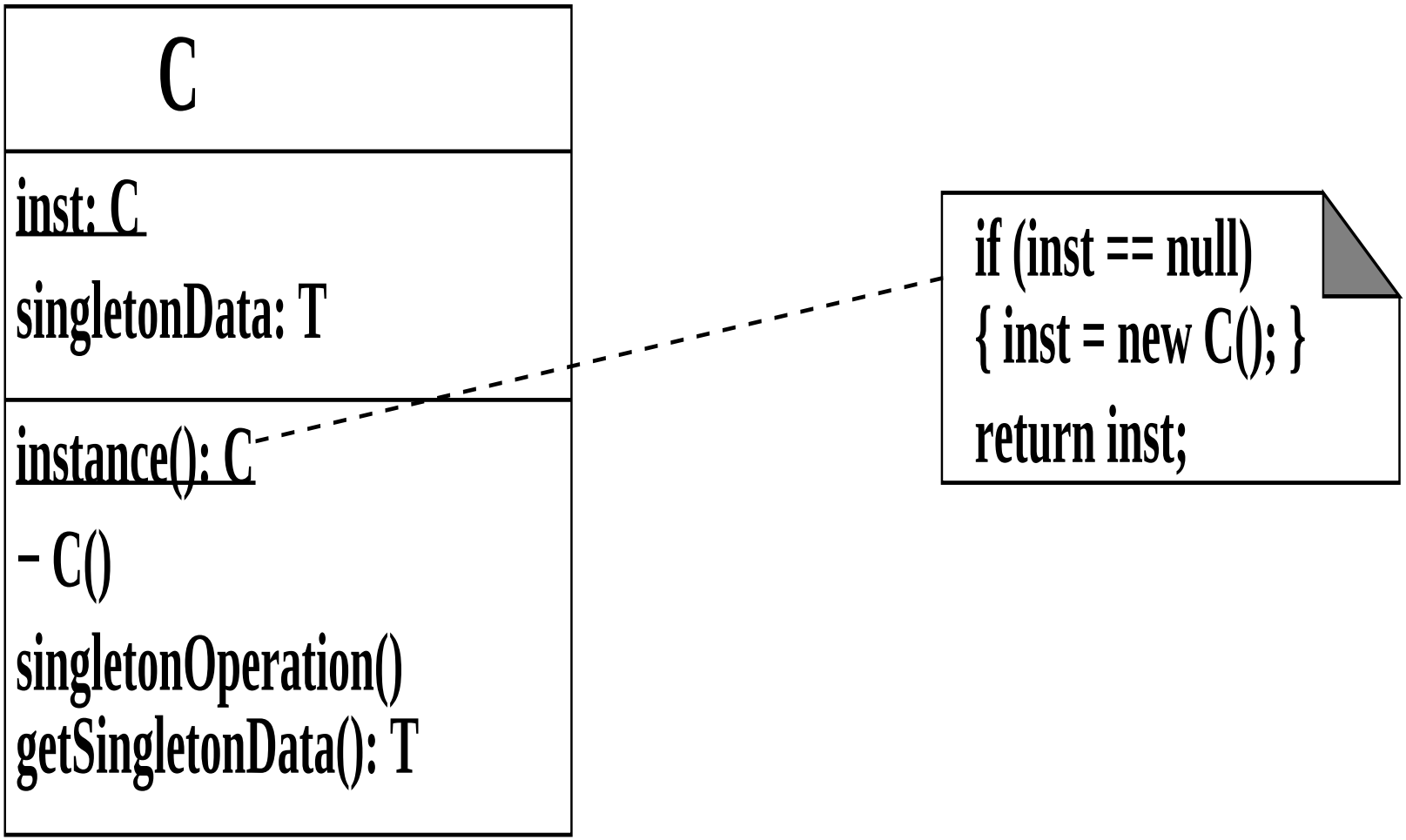
Design structure of Builder pattern

Singleton pattern

Creational pattern used to define classes which should have only a single instance (eg, 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.



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.

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 translation system, a *Dictionary* could be *Singleton* class, so it may 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);
```

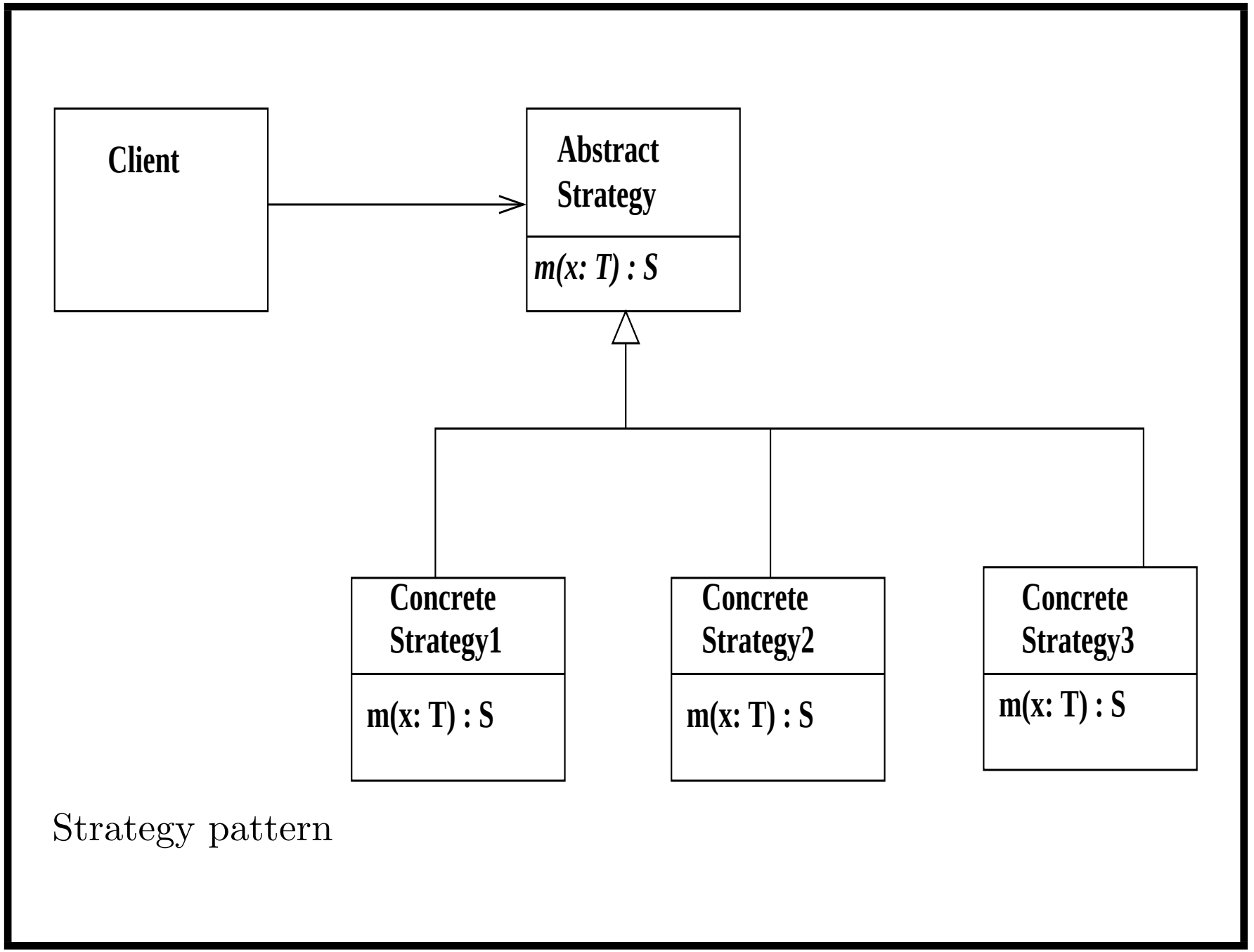
Alternative to Singleton would be a *static* class: all operations + data *static* (class scope). But non-OO, and inheritance cannot be used with static methods.

Strategy pattern

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

Elements of pattern are:

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



Strategy pattern

Strategy pattern

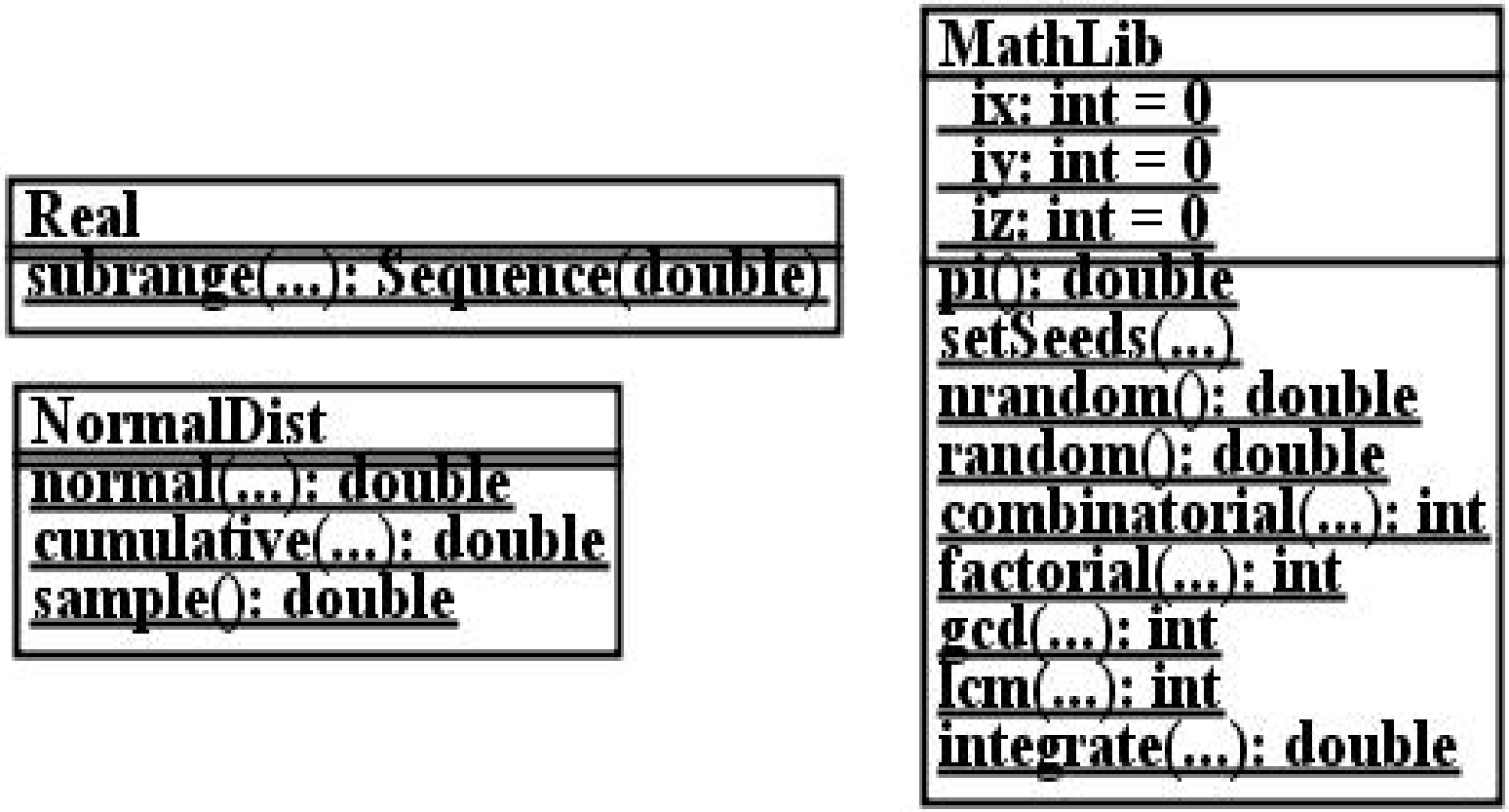
- Benefits of pattern: simplifies code by separating alternative versions of algorithm into different classes + allows dynamic selection of versions by a client.
- Commonalities between algorithms can be expressed using Template Method structure of hook methods.

Design patterns compared to refactoring

- Refactorings + patterns can both transform existing system structure into new improved structure
- Refactorings relatively small changes, general-purpose. Can be automated
- Patterns more elaborate and special-purpose, to solve specific problem. Require manual work to adapt to specific cases
- Patterns can be used to organise new designs.

Libraries and reuse

- During development, team may identify useful functionalities which could be reused in future projects.
- Define in appropriate library classes/models. Eg., if functions $factorial(n : int) : int$, $combinatorial(n : int, m : int) : int$ "n choose m", $gcd(n : int, m : int) : int$ and $lcm(n : int, m : int) : int$ identified for reuse, define them in language-independent *mathlibmm.txt* model to include in future applications.
- Class *MathLib* defined with functions as static operations.
- Since functions to be widely used, care should be taken to ensure efficiency and correctness.



Mathematical functions library

Libraries

For *factorial* and *combinatorial*, $Integer.Prd(a, b, i, e)$ operator used to compute $\Pi_{i=a}^{b}\ e$, to optimise operations:

```
static query combinatorial(n: int, m: int): int
pre: n >= m & m >= 0 & n <= 25
post:
 (n - m < m =>
  result = Integer.Prd(m+1,n,i,i)/Integer.Prd(1,n-m,j,j)) &
 (n - m >= m =>
  result = Integer.Prd(n-m+1,n,i,i)/Integer.Prd(1,m,j,j))

static query factorial(x: int): int
pre: x <= 12
post:
  ( x < 2 => result = 1 ) &
  ( x >= 2 => result = Integer.Prd(2,x,i,i) )
```

For *gcd*, well-known recursive computation:

```
static query gcd(x: int, y: int): int
pre: x >= 0 & y >= 0
post:
  (x = 0  =>  result = y) &
  (y = 0  =>  result = x) &
  (x = y  =>  result = x) &
  (x < y  =>  result = gcd(x, y mod x))  &
  (y < x  =>  result = gcd(x mod y, y))
```

Considered too inefficient for use in library, replaced by explicit algorithm:

```
static query gcd(x: int, y: int): int
pre: x >= 0 & y >= 0
post: true
activity:
  l : int ; k : int ; l := x  ; k := y  ;
```

```
while l /= 0 & k /= 0 & l /= k
do
    if l < k then k := k mod l
    else l := l mod k ;
if l = 0 then result := k
else result := l ;
return result
```

From *gcd*, the *lcm* can be directly calculated:

```
static query lcm(x: int, y: int): int
pre: x >= 1 & y >= 1
post: result = ( x * y ) / gcd(x,y)
```

Summary of Part 4

- We described how architecture diagrams can be used to describe system structure
- Described class diagram refactorings
- Described several important design patterns
- Discussed design reuse via libraries.

Coursework: Implementation

- Write implementations for your classes, use cases + operations in Java or another language.
- You may find it helpful to use the UML-RSDS code generator, or other UML tool with code generation capabilities.
- Write test cases and check your system using these.

Deadline: November 30th, 4pm.

Submission by team leader/one representative on Keats.

Coursework: Final report

- No more than 15 pages, in PDF format.
- Named by number of group, eg: Team15OSD.pdf
- Describe team members roles + contributions to project.
- Include class diagram, use case diagram, operation + use case pseudocode.
- Include a code listing of implemented classes and use case code.
- Include test cases and results.