YORK
UNIVERSITÉ
UNIVERSITY

Software Development
EECS3311

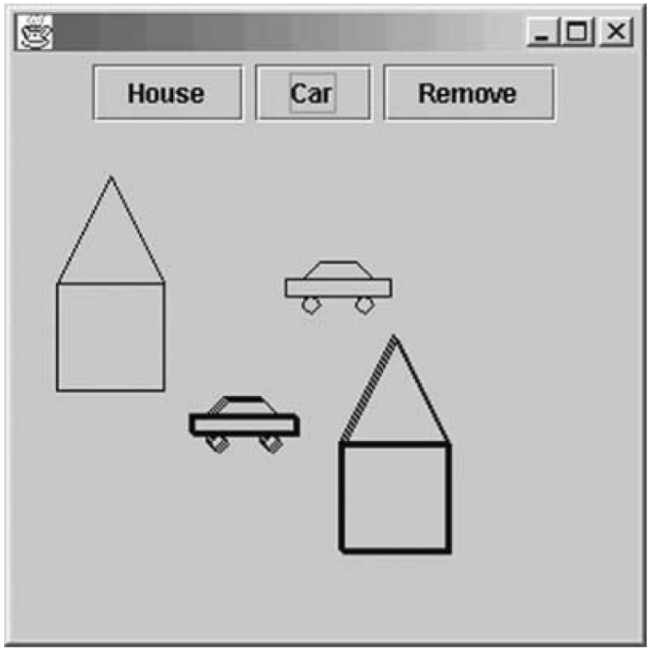
Review of Template Design Pattern

Composite Design Pattern

some more UML

Challenge!

Consider a scene editor



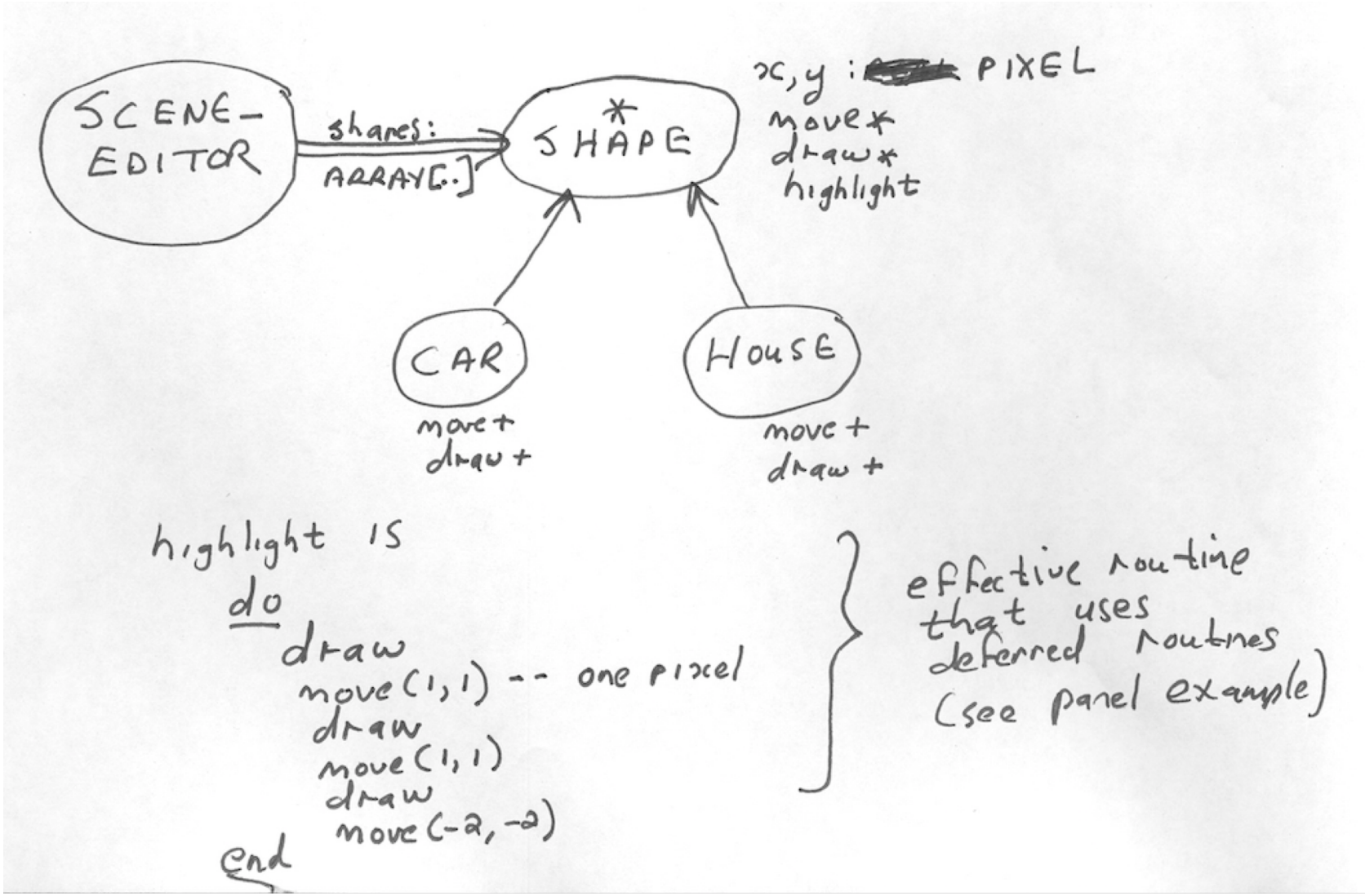
A scene consists of shapes such as cars, houses, and so on. The user is able to add, move, and delete shapes. In order to move objects, the user selects a shape by clicking it. Each object is highlighted when selected.

It is desirable that different kinds of shapes are highlighted in a similar fashion.

Develop a BON diagram demonstrating the classes you might include in this system along with their relationships. How would you ensure that shapes are highlighted in a similar fashion?

Hint: A shape-independent way to highlight is draw, move slightly, draw again, move again, draw again (shown in the picture)

Solution: Template Design Pattern



Template Method Design Pattern

Intent

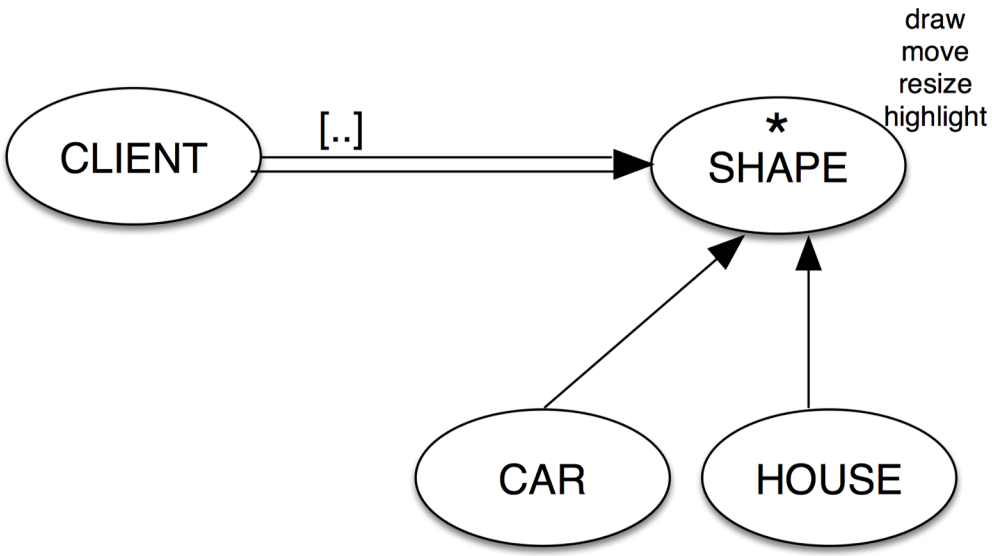
Define the skeleton of an algorithm in an operation, deferring some steps to client subclasses.

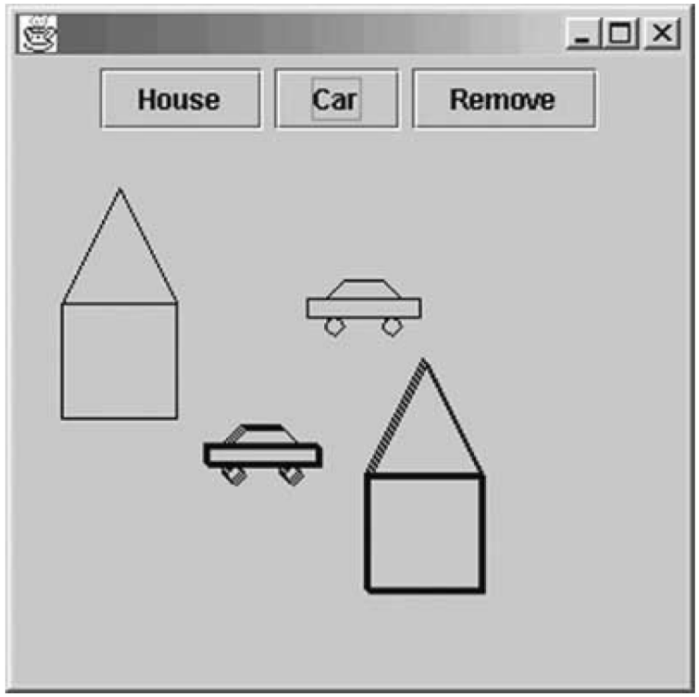
Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Base class declares algorithm 'placeholders', and derived classes implement the placeholders.

Use the Template Method pattern when you want to let clients extend only particular steps of an algorithm, but not the whole algorithm or its structure.

Challenge!



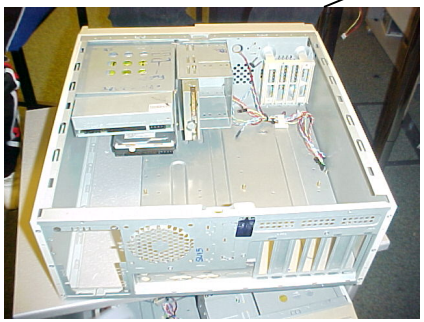


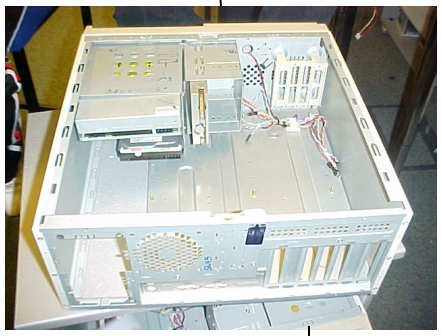
You are now asked to add a grouping feature to this system. This means that the user should be able to group two or more shapes together, and subsequently, move, resize, or highlight the group as one unit. How would you modify the design below in order to add this feature?

Tree structures with part-whole hierarchies

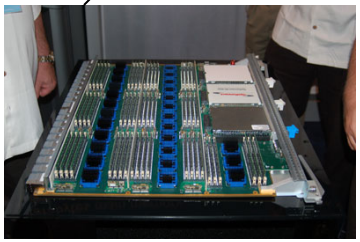
CABINET

CHASSIS





POWER_ SUPPLY



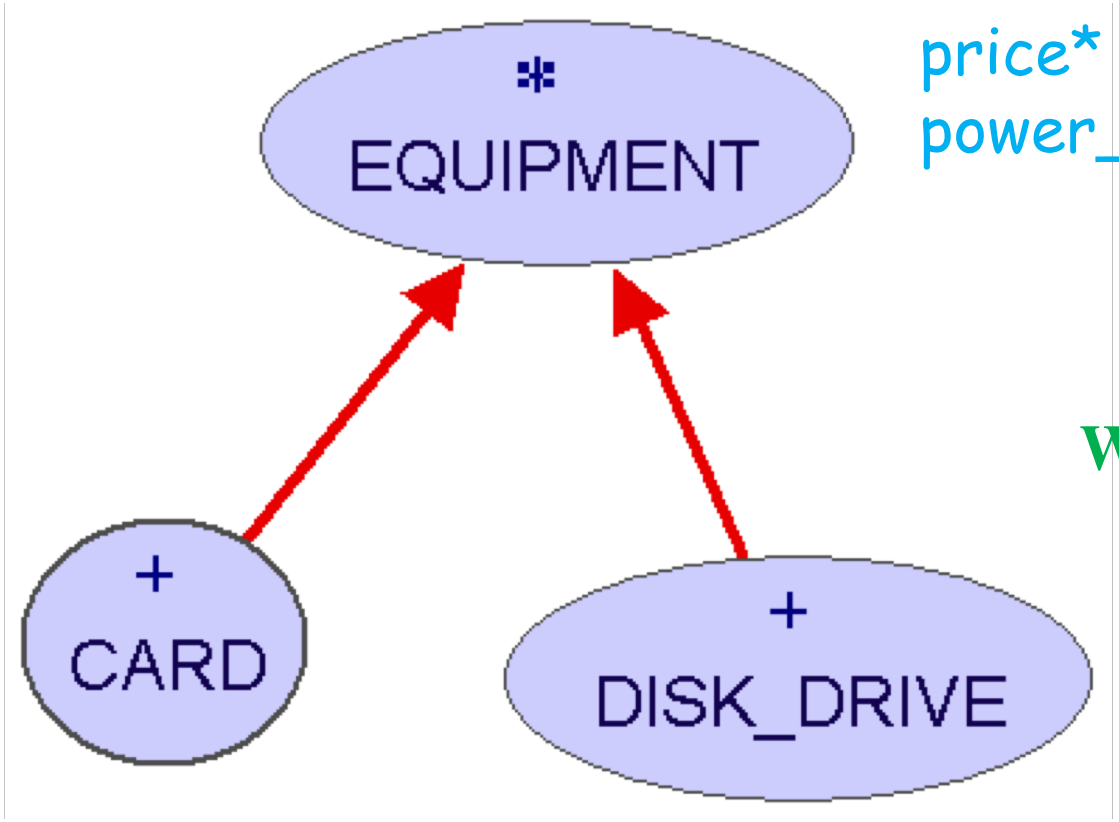
CARD

HARD_DRIVE

DVD-CDROM

Problem

- Many manufactured systems such as computer systems are composed of individual components and sub-systems that contain components.
- For example, a computer system can have cabinets that contain various types of chassis that (in turn) contain components (hard-drive chassis, power-supply chassis) and busses that contain cards.
- The entire system is composed of individual pieces of equipment (CPU, GPU, hard drives, DVD), as well as composites such as cabinets, busses and a chassis.
- Each piece of equipment has properties such as power-consumption and cost. Design a system that will allow us to easily build systems and calculate their total cost and power consumption.



price*
power_consumption*

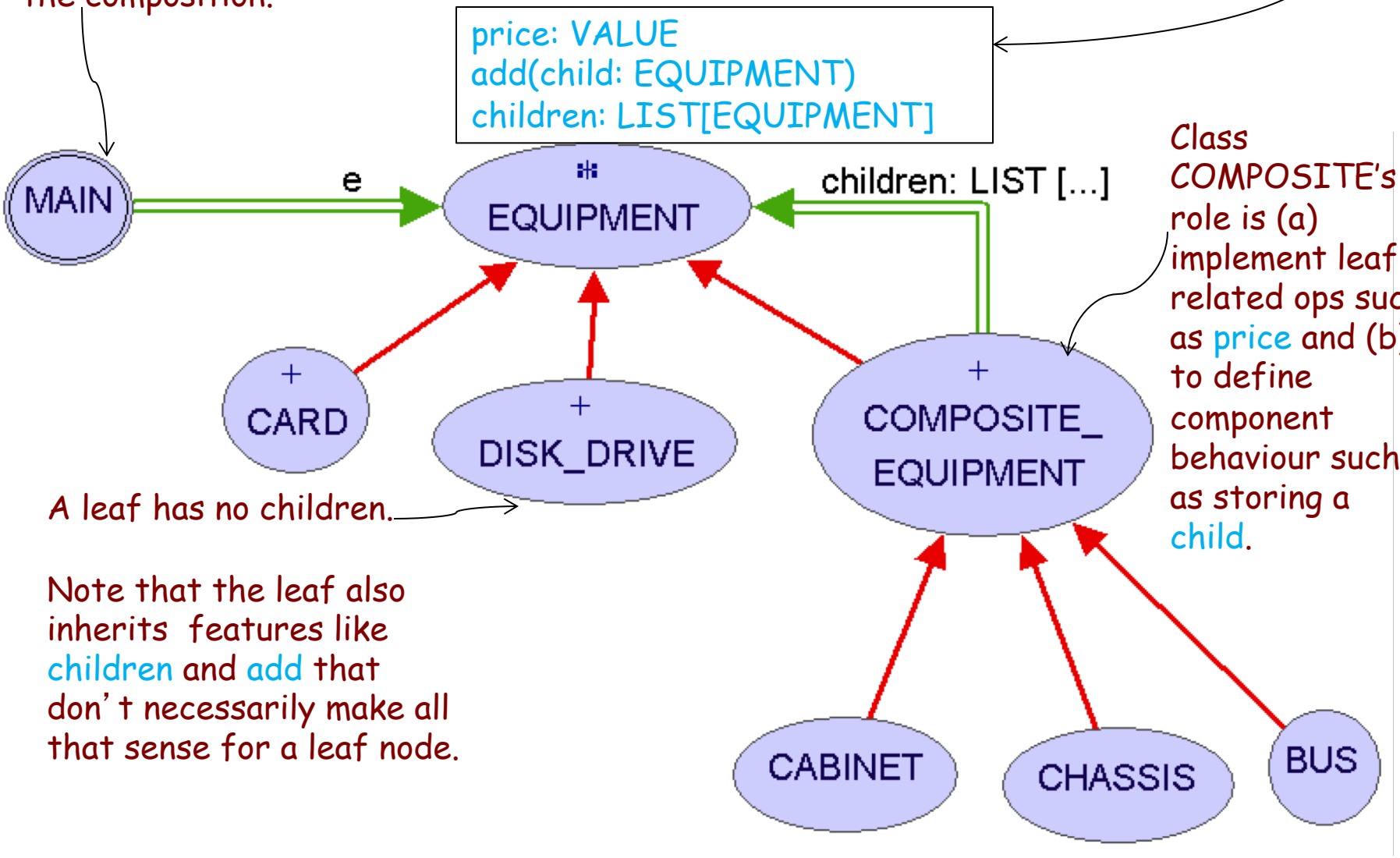
What about composites?

Composite Pattern

- Compose objects into tree structures that represent whole-part hierarchies. Composites let clients treat individual objects (leafs) and compositions (nodes) of objects uniformly.
- We can then apply the same operation (e.g. calculate price or power consumption) over both composites and individual objects (i.e. we can ignore the difference between leafs and nodes).

The client uses abstract class EQUIPMENT to manipulate objects in the composition.

Class EQUIPMENT defines an interface for all objects in the composition: both the composite and leaf nodes.
May implement default behavior for add(child) etc.

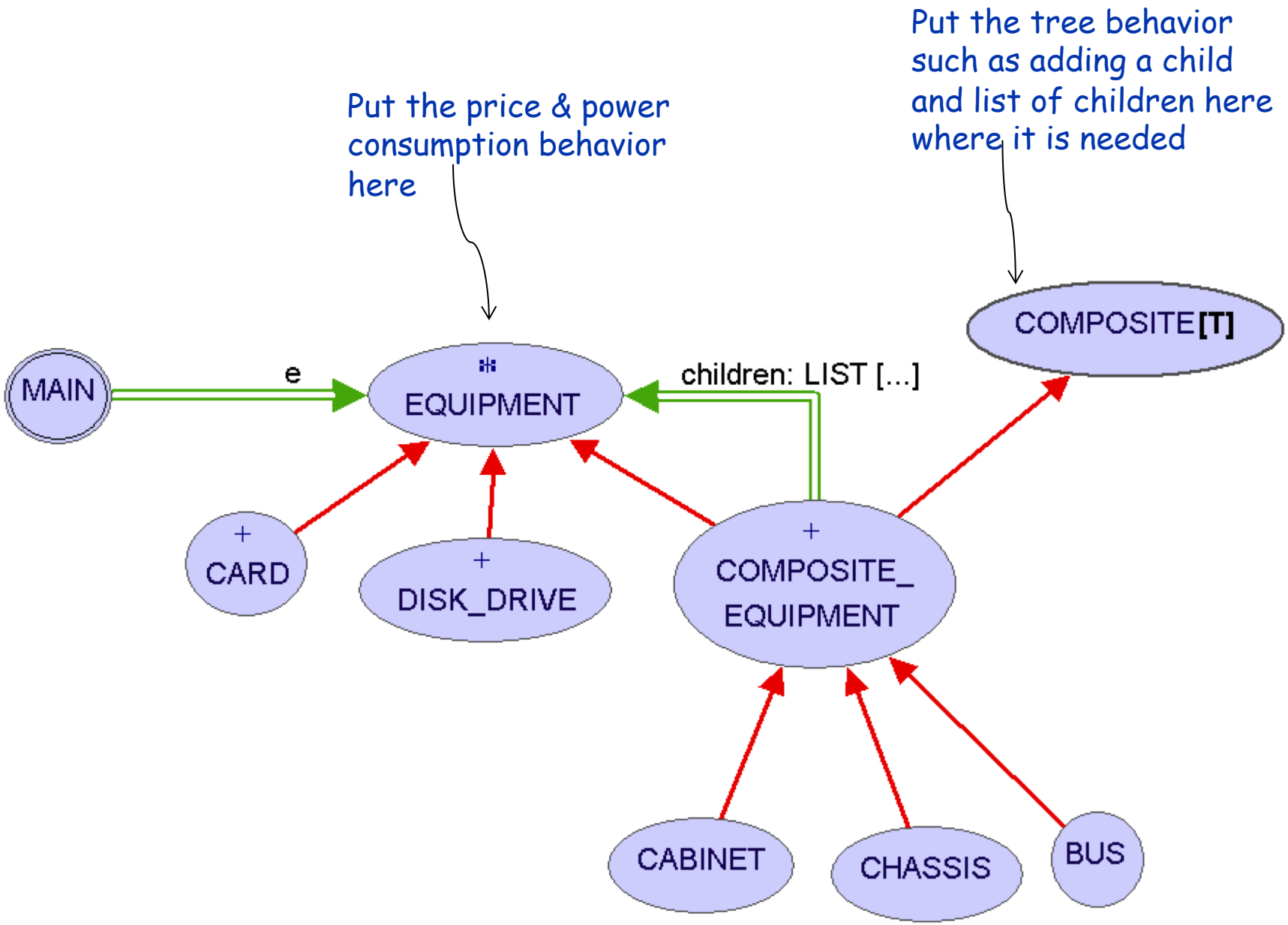


Class COMPOSITE's role is (a) implement leaf related ops such as price and (b) to define component behaviour such as storing a child.

A leaf has no children.

Note that the leaf also inherits features like children and add that don't necessarily make all that sense for a leaf node.

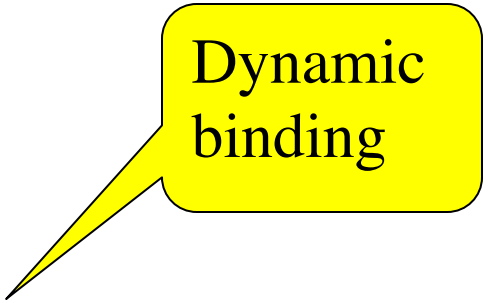
Cleaner solution – Multiple Inheritance

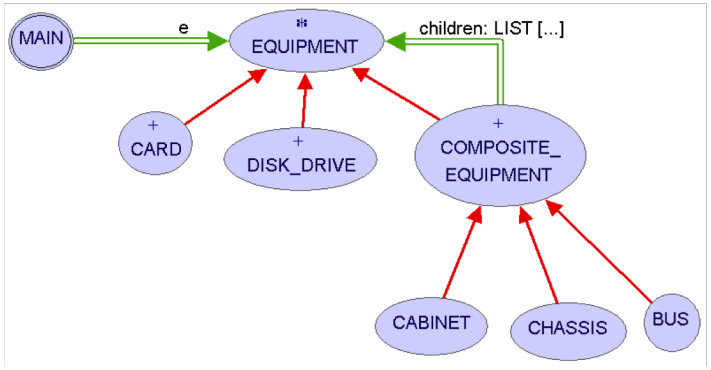


```
class COMPOSITE [T] feature
        children : LIST [T]
        add (new_child: T)
                require   new_child /= Void
                do
                   children.extend(new_child)
                ensure   has (new_child)

        remove (child: T)
                require  child /= Void
                ensure  not has (child)

        has (child: T): BOOLEAN
                   -- does 'child' belong to the composite?
                require  child /= Void
invariant
        children_not_void: children /= Void
end
```

```
deferred class EQUIPMENT feature
        make (its_name: STRING)
        discount_price: REAL
        name: STRING
        net_price: REAL
        power: REAL
invariant
        name_not_void: name /= Void
        positive_power: power >= 0.0
        positive_price: discount_price >= 0.0
        real_discount: net_price >= discount_price


end
```

```
class COMPOSITE_EQUIPMENT inherit
   COMPOSITE [EQUIPMENT]
   EQUIPMENT
feature
   net_price : REAL
         -- sum the net prices of the subsystems
      local i : INTEGER
      do
         from i := 1 until i > children.count
         loop
            Result := Result + children[i].net_price
            i := i + 1
         end
      end  ...
end
```






```
class MAIN create
        make
feature
        e : EQUIPMENT
  make
        local
          cabinet : CABINET -- Will hold a CHASSIS
          chassis : CHASSIS -- Will contain a BUS and a DISK_DRIVE
          bus : BUS -- Will hold a CARD
    do
        create cabinet.make("PC Cabinet");
        create chassis.make("PC Chassis")
        cabinet.add(chassis)
        create bus.make("MCA Bus");
        create {CARD}e.make("16Mbs Token Ring");
        bus.add(e);
        chassis.add(bus)
        create{DISK_DRIVE}e.make("500 GB harddrive");
        chassis.add(e)
         print("The net price is ")
         print(cabinet.net_price);
        io.put_new_line
    end -- make
end -- MAIN
```

Cabinet

Chassis

DISK

BUS

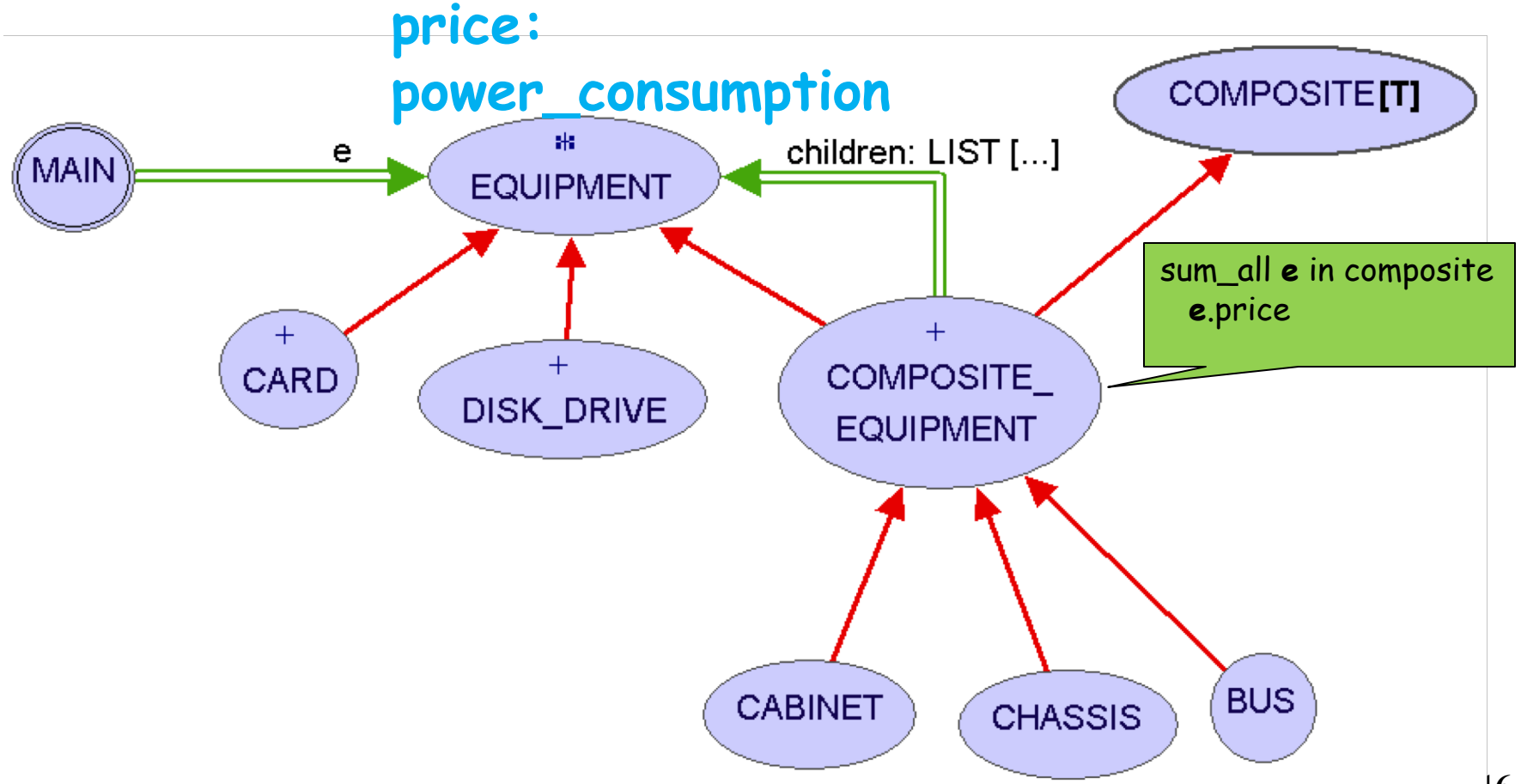
CARD

```
net_price : REAL
    -- Sum the net prices of the subequipments
  local i : INTEGER
  do
    from i := 1 until i > children.count
    loop
      Result := Result + children[i].net_price
      i := i + 1
    end -- loop
  end ...
```

See ITERABLE_ARITHMETIC in Mathmodels

Contracts

- How would you write the contracts?



```
expanded class
    ITERABLE_ARITHMETIC[G -> NUMERIC create default_create end]

feature
    zero: G
        do
            create Result
            Result := Result.zero
        end

    one: G
        do
            create Result
            Result := Result.one
        end

    sum(collection: ITERABLE[G]): G
        do
            Result := zero
            across collection as it loop
                Result := Result + it.item
            end
        end

    product(collection: ITERABLE[G]): G
        do
            Result := one
            across collection as it loop
                Result := Result * it.item
            end
        end
```

```
expanded class
    ITERABLE_ARITHMETIC[G -> NUMERIC create default_create end]

feature
    zero: G
        do
            create Result
            Result := Result.zero
        end

    one: G
        do
            create Result
            Result := Result.one
        end

    sum(collection: ITERABLE[G]): G
        do
            Result := zero
            across collection as it loop
                Result := Result + it.item
            end
        end

    product(collection: ITERABLE[G]): G
        do
            Result := one
            across collection as it loop
```

```
a: ARRAY[REAL]
ia1: ITERABLE_ARITHMETIC[REAL]
sum: REAL

make
            -- Run application.
    do
            a := <<4.5, 6.9, 8.2>>

            sum := ia1.sum (a)
            check 19.59 <= sum and sum <= 19.61 end
    end
```

```
a: ARRAY[REAL]
ia1: ITERABLE_ARITHMETIC[REAL]
sum: REAL
b: ARRAY[VALUE]
ia2: ITERABLE_ARITHMETIC[VALUE]

make
    local
        v1,v2,v3, vsum, v196: VALUE
    do
        a := <<4.5, 6.9, 8.2>>

        sum := ia1.sum (a)
        check 19.59 <= sum and sum <= 19.61 end

        v1 := "4.5"; v2 := "6.9"; v3 := "8.2"
        v196 := "19.6"
```

Name	Value	Type	Address
Current object	<0x10438C0B0>	ROOT	0x10438C0B0
a	<0x10438C0B8>	ARRAY [REAL_32]	0x10438C0B8
b	<0x10438C0C0>	ARRAY [VALUE]	0x10438C0C0
ia1	<0x10438C0C8>	ITERABLE_ARITHMETIC [RE...	0x10438C0C8
ia2	<0x10438C0D0>	ITERABLE_ARITHMETIC [VA...	0x10438C0D0
sum	19.6	REAL_32	
Once routines			
Locals			
v1	4.50	VALUE	0x10438C120
v196	19.60	VALUE	0x10438C140
v2	6.90	VALUE	0x10438C128
v3	8.20	VALUE	0x10438C130
vsum	19.60	VALUE	0x10438C138

Arbitrary numerical expression f:NUM→NUM

∑x∈collection:f(x)

```
expanded class
    ITERABLE_ARITHMETIC[G -> NUMERIC create default_create end]

    sumf(collection: ITERABLE[G]; f: FUNCTION[G, G]): G
        do
            create Result.default_create
            Result := Result.zero
            across  collection as it loop
                Result := Result + f(it.item)
            end
        end

    productf(collection: ITERABLE[G]; f: FUNCTION[G, G]): G
        do
            create Result.default_create
            Result := Result.one
            across  collection as it loop
                Result := Result * f(it.item)
            end
        end
```

$(\sum x \in 1..3: x*x+1) = 2+5+10 = 17$

Non-software analogy

- Arithmetic expressions are composites
- Each arithmetic expression has
  - An operand
  - An operator such as + - * /
  - Another operand
- Each operand may be another arithmetic expression
- Valid arithmetic expressions
  - 2 + 3
  - (2 + 3) + (4 * 6)

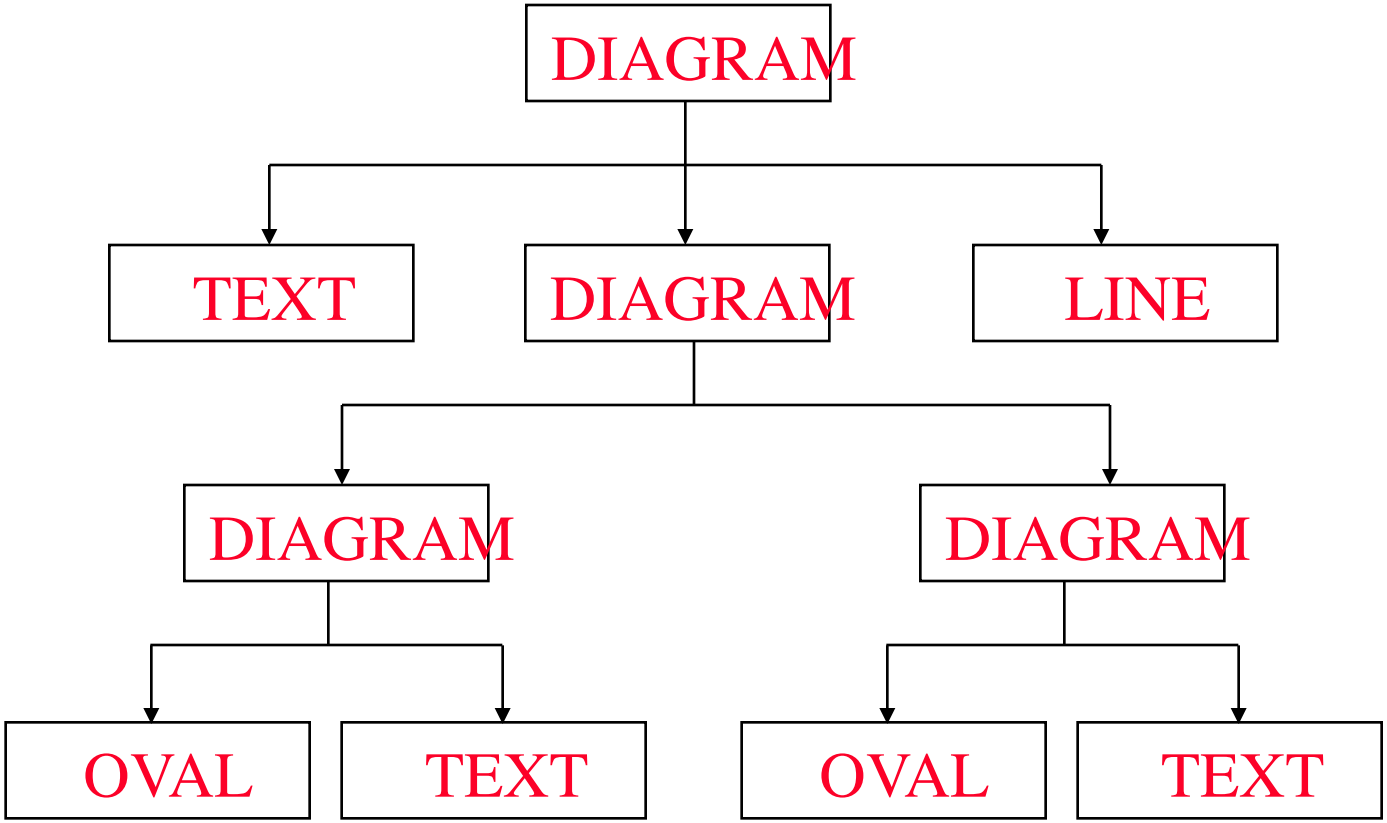
Composite Pattern

- **Intent**
  - Compose objects into tree structures representing part-whole hierarchies
  - Clients deal uniformly with individual objects and hierarchies of objects

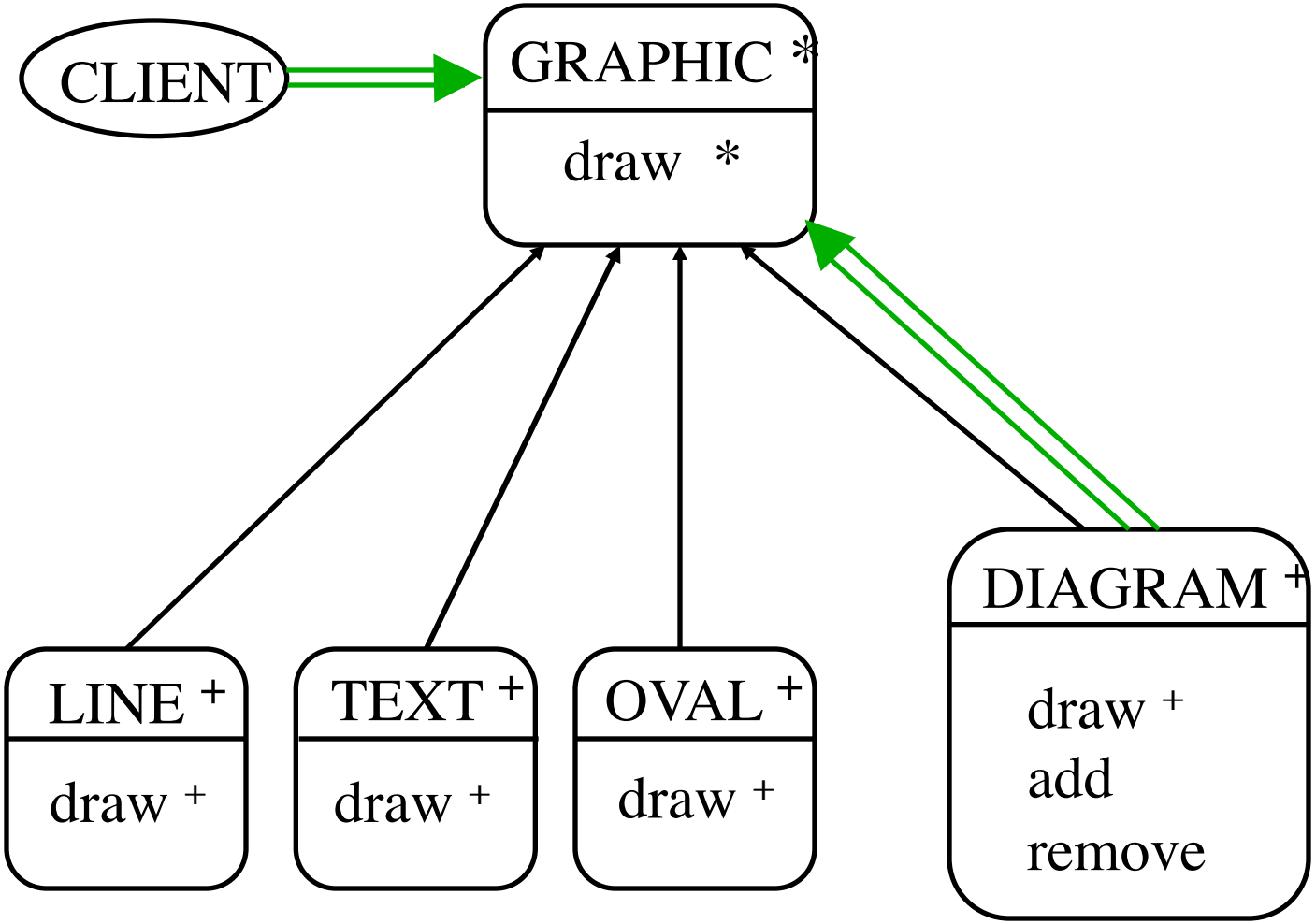
Composite – Motivation

- Applications that have recursive groupings of primitives and groups
  - Drawing programs
    - Lines, text, figures and groups
  - Directory structure
    - Folders and files
- Operations on groups are different than primitives but clients treat them in the same way

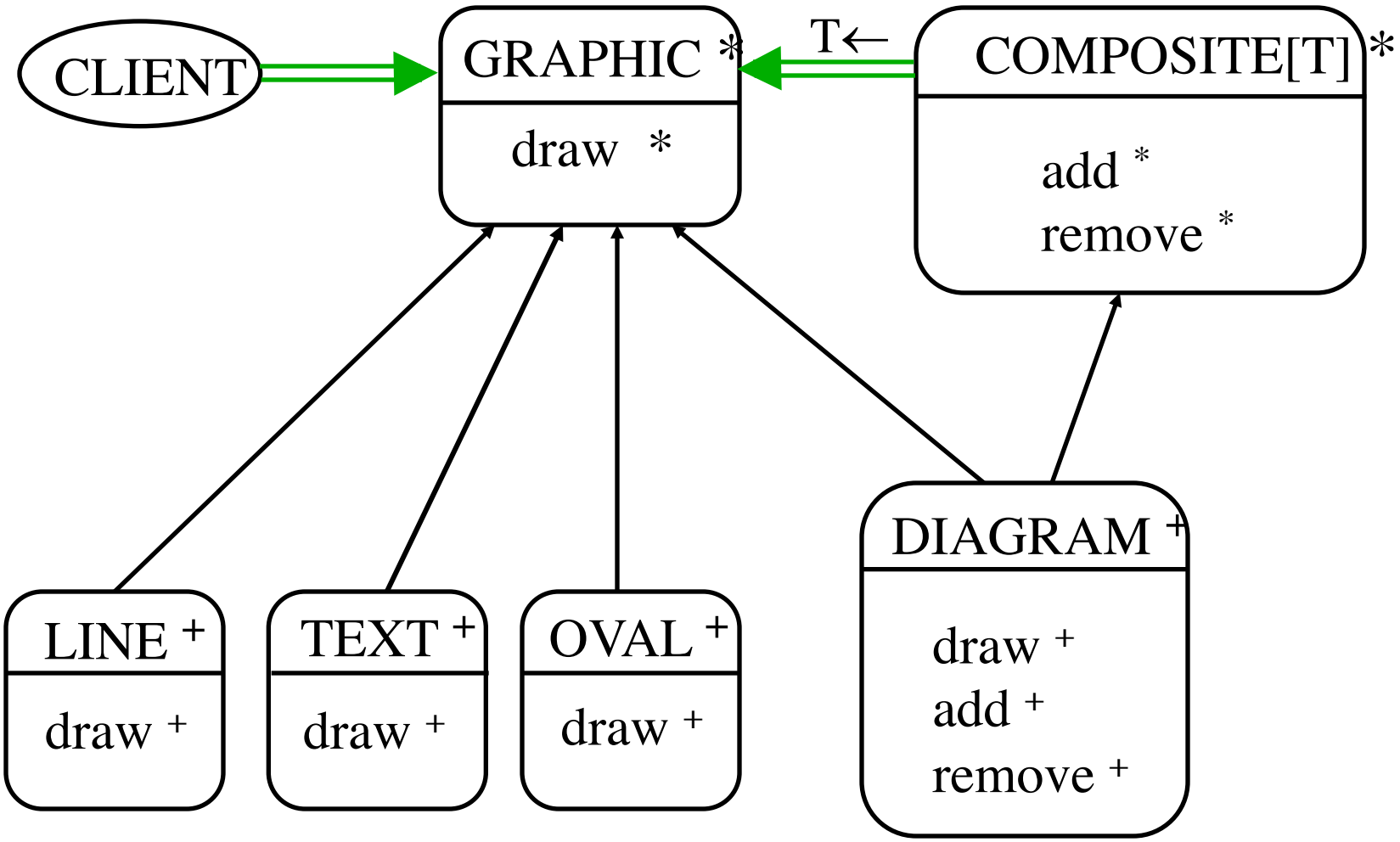
Whole-Part Hierarchy Example



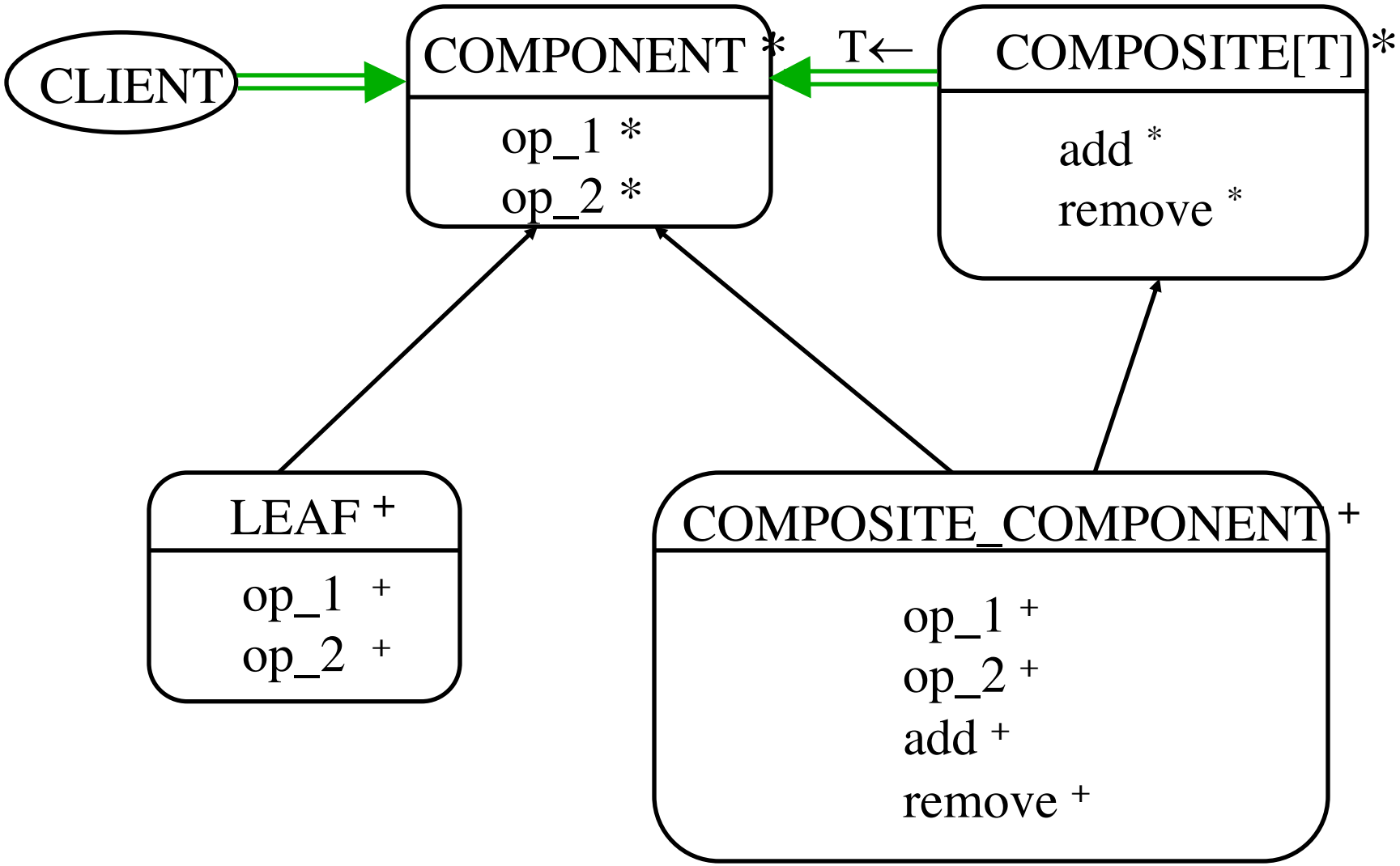
Composite – Example Architecture



Composite – Alternative Architecture



Composite – Abstract Architecture



Composite – Applicability

- Represent part-whole hierarchies of objects
- Clients can ignore difference between individual objects and compositions
- Clients deal with all objects in a composition in the same way

Composite – Participants

- **Component**
  - Defines properties of an entity
- **Leaf Component**
  - Defines properties of a primitive entity
- **Composite**
  - Declares properties of a collection of entities
- **Composite Component**
  - Combines properties of a collection of entities and properties of a primitive entity
- **Client**
  - Uses components (leaf or composite)

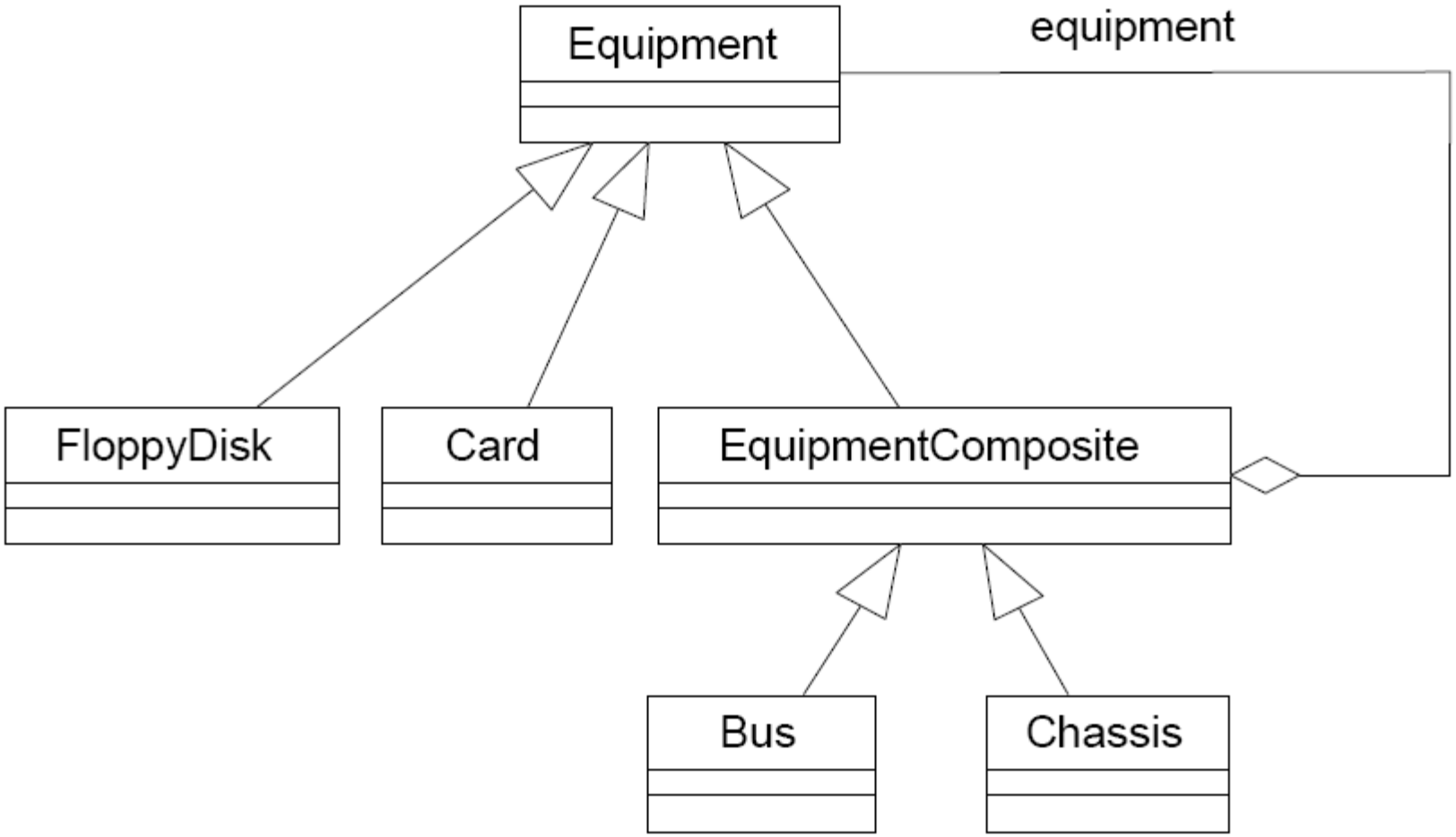
Composite – Consequences

- Whenever client expects a primitive it can accept a composite
- Client is simplified by removing tag-case statements to identify parts of the composition
- Easy to add new components by subclassing, client does not change
- If compositions are to have restricted sets of components, have to rely on run-time checking

Composite real-world examples

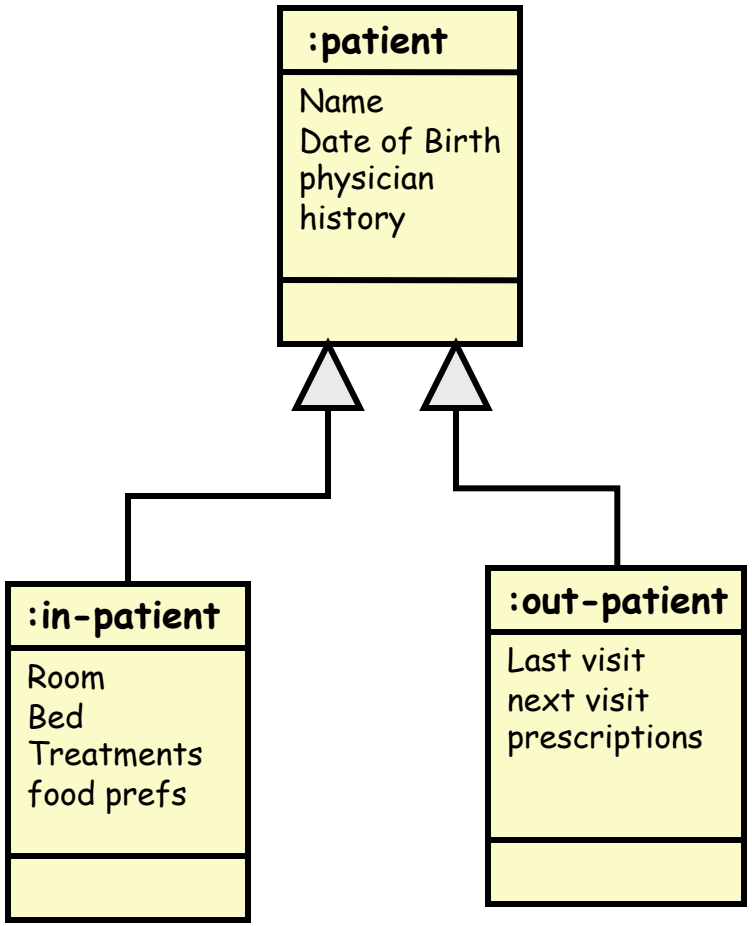
- java.awt.Component is a class representing objects that have a graphical representation that can be displayed on the screen and that can interact with the user [COMPONENT]
- java.awt.Container represents components that can contain other components [COMPOSITE]
  - Contains methods add(Component) and remove(Component)
  - Example subclasses: Panel, Window
- java.awt.Button is a component that does not contain other components [LEAF]

UML class diagram

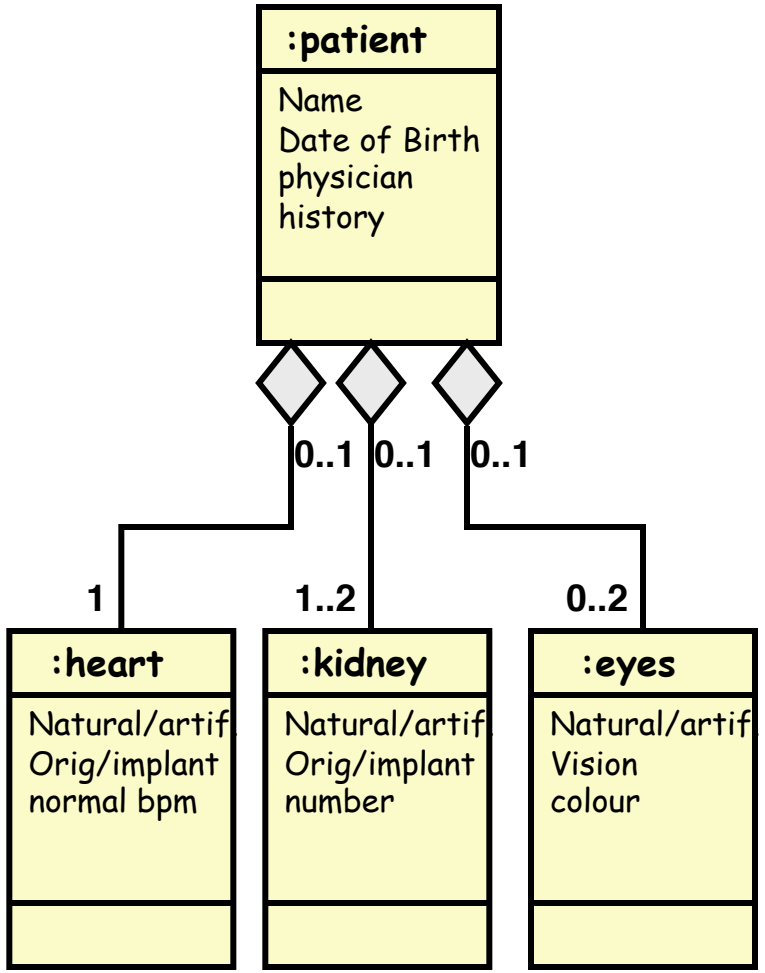


UML aggregations/compositions

Generalization
(an abstraction hierarchy)



Aggregation
(a partitioning hierarchy)



BON Diagram

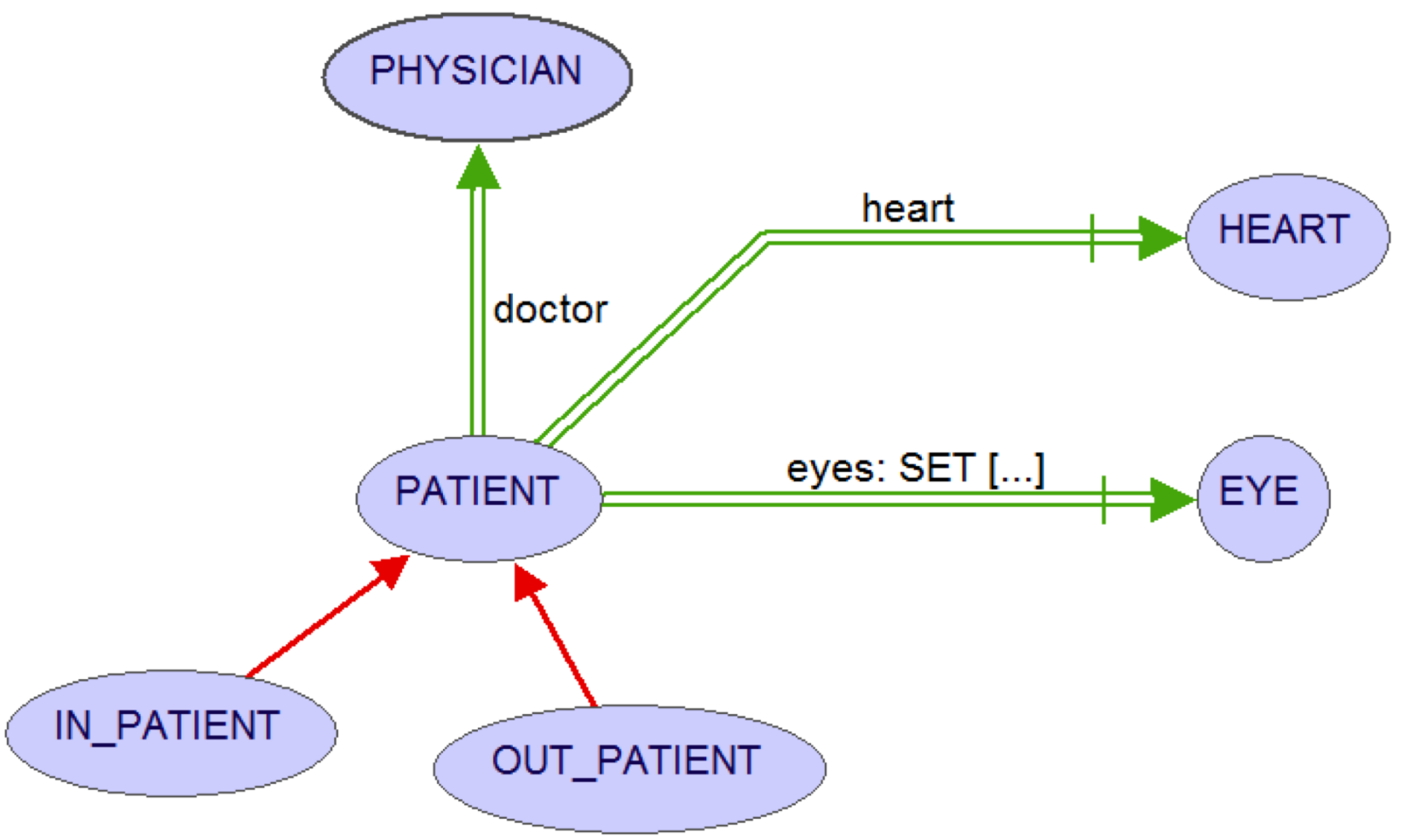
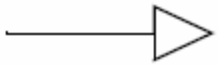
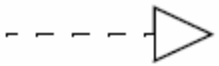
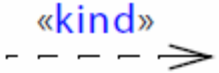
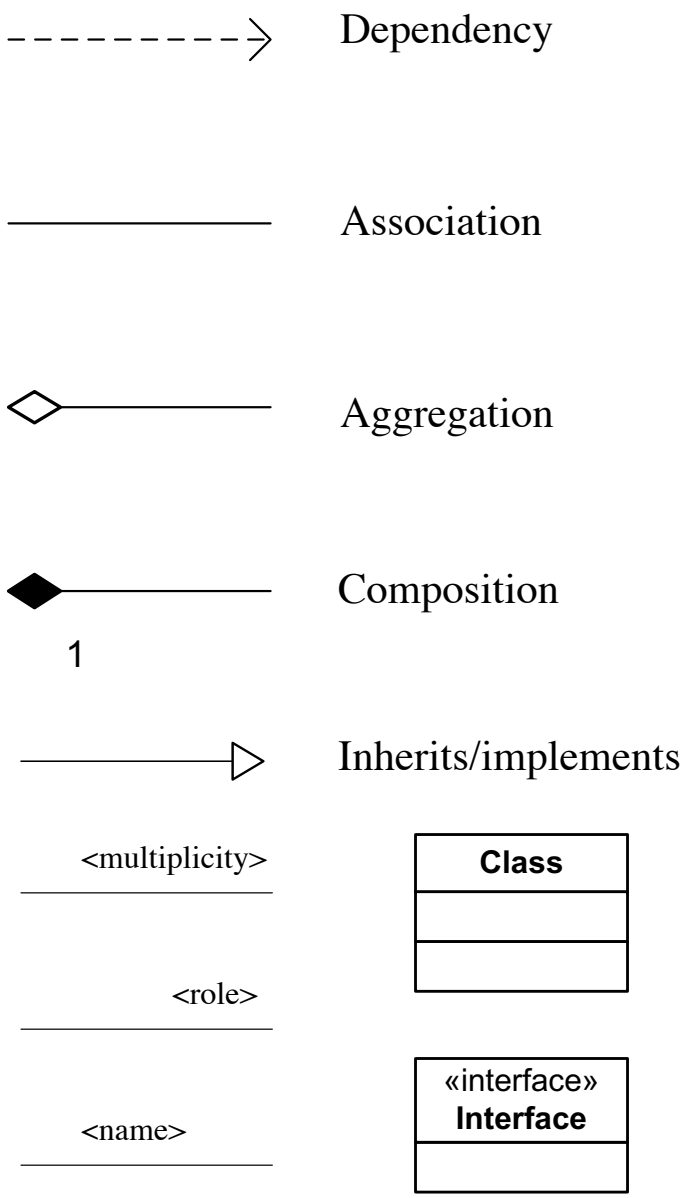

PHYSICIAN
heart
HEART
doctor
PATIENT
eyes: SET [...]
EYE
IN_PATIENT
OUT_PATIENT

Table 4-2: *Kinds of Relationships*

Relationship	*Function*	*Notation*
association	A description of a connection among instances of classes	
dependency	A relationship between two model elements	
generalization	A relationship between a more specific and a more general description, used for inheritance and polymorphic type declarations	
realization	Relationship between a specification and its implementation	
usage	A situation in which one element requires another for its correct functioning	

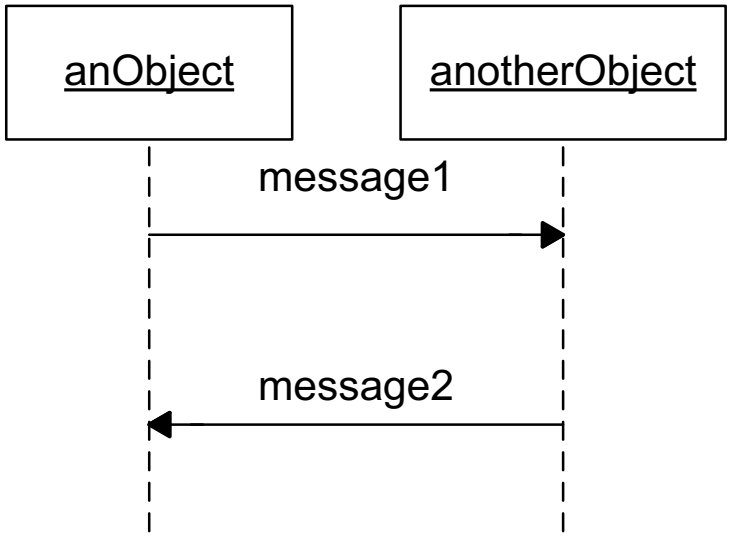
UML modelling

- Class diagrams show the static structure of classes, interfaces and the relationships between them

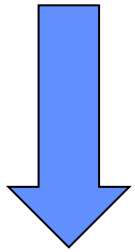


Sequence diagrams

- To emphasise time-ordering of method calls, object lifelines are used



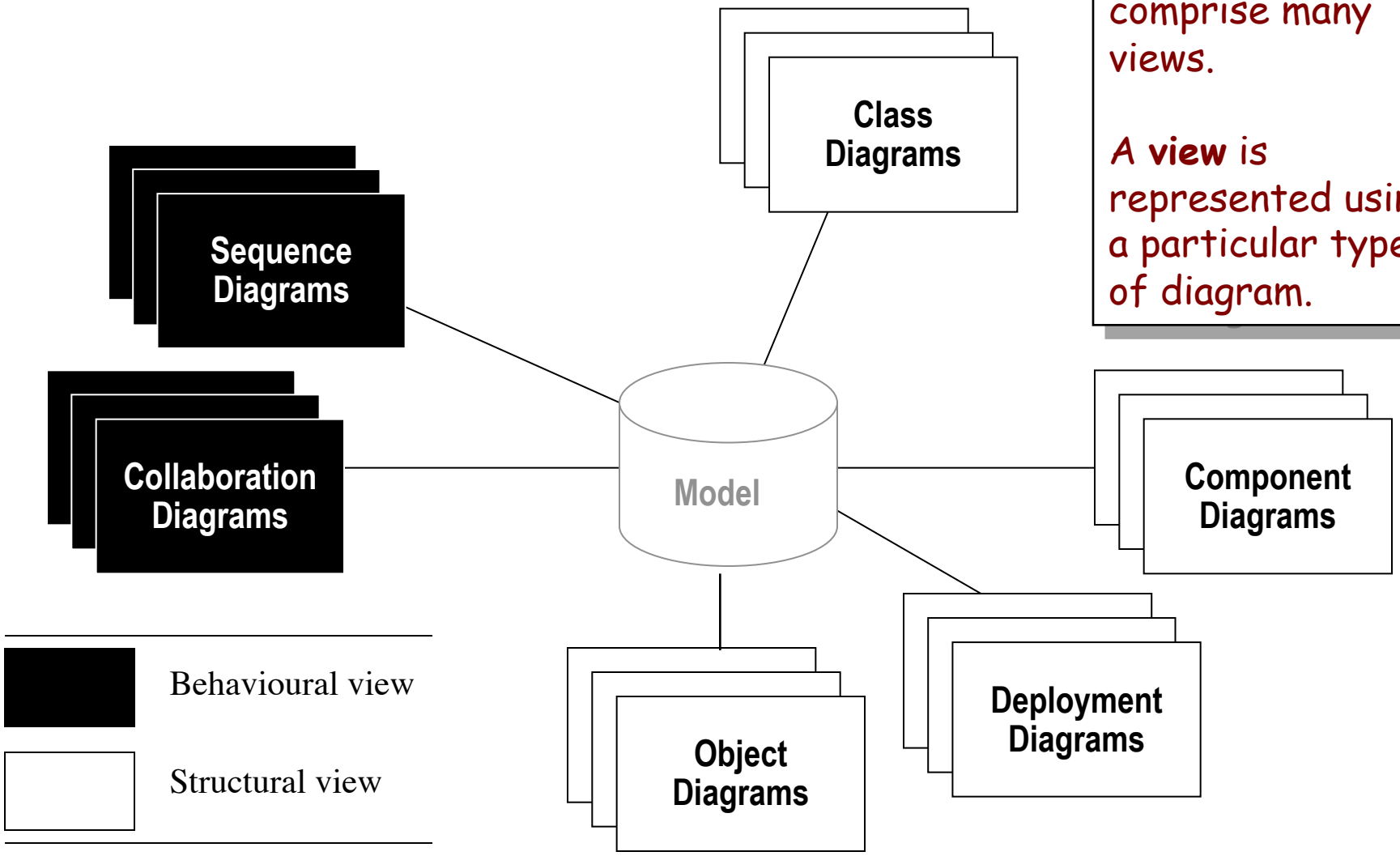
A lifeline is a dashed vertical line and represents the duration of an object's existence.



UML Diagrams

A **model** is a complete description of a system and may comprise many views.

A **view** is represented using a particular type of diagram.



Questions

- Provide a BON diagram for grouping items in the scene editor