



Basics of programming 3

Java and UML



Steps of SW development

- Requirements

- goal: defining what is expected

- Analysis

- goal: understanding problem domain

- Design

- goal: closing on implementation decisions

- Implementation

- goal: creating formal, machine language description

- Testing

- goal: assessing adherence to requirements



Models in SW development

■ Analysis

- focus on responsibilities and interactions
 - public methods are defined
 - attributes (if any) only specify responsibility

■ Design

- focus on internal structures
 - protected, package, private members
 - high-level structures (e.g. list, set) concretized

■ Implementation

- programming language constraints considered
 - implementation decisions only



Modelling

- High level description

- details are neglected on purpose

- essence of good modelling: what is relevant?
 - flexibility and universality are important

- must maintain abstraction level

- don't dive into details
 - knowing capabilities is still a must

- Formal language is needed

- to understand each other

- to understand past self



OO concepts in modelling

- Objects and classes
 - static and dynamic views
 - class level and object level views
- Types (“dumb objects”)
 - primitive and complex
 - abstraction is needed
- Associations and Inheritance
 - connections have abstraction level too



Static and dynamic views

- Static view
 - all or relevant components and their relationships
 - describes all kinds of connections
 - temporary connection (dependency)
 - constant (association, aggregation, inheritance, etc)
 - instance or class level
- Dynamic view
 - temporal behaviour of components
- Views are related
 - must be consistent



Modelling attributes (types)

- Abstract types

- name
- address
- age
- coordinates
- ...

- Java types

- String** name
- class Address**
- int** age
- class Coord**

- C++ types

- char*** (string) name
- struct** (class) **address**
- int** age
- struct coord**



Modelling attributes (types)

- Keep abstraction level high
 - e.g. use coordinates as type, not class
- Hide attributes
 - private or protected
 - use getter/setter methods for access
 - enables additional behaviour
- Type representation should be shy
 - should only consider internal consistency



Visibility in UML, Java and C++

- Similar concepts with small differences
 - private (-)
 - access for defining class only
 - package (~)
 - not in C++ (namespaces have “public” visibility)
 - no explicit notation in Java, access for package members
 - protected (#)
 - access for subclasses
 - in Java access for package members
 - public (+)
 - access for everybody

Handling Units

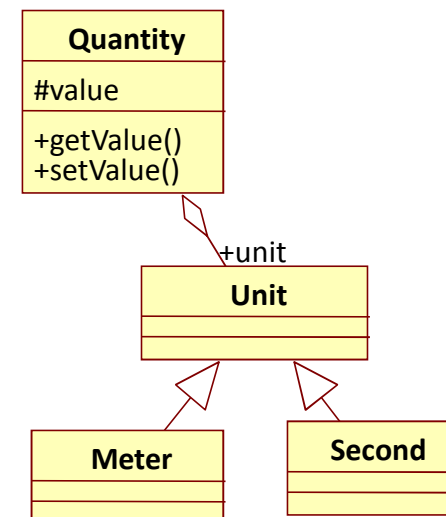
■ Classic approach

□ implicit units

- e.g. `double speed // [m/s]`
- good documentation is needed (c.f. mars probe)
- even in API: `Thread.sleep(long)`

■ Sophisticated approach

- explicit units
- helps conversion, logging, etc
- might be overkill





Associations between classes

- **Dependency**
 - temporal, lasts a single method call
- **Association**
 - long term, lasts multiple method calls
 - reference is stored
- **Aggregation / Composition**
 - life long
 - lifecycle management is needed



Association

- Notation

- solid line 
- arrows and crosses show navigability

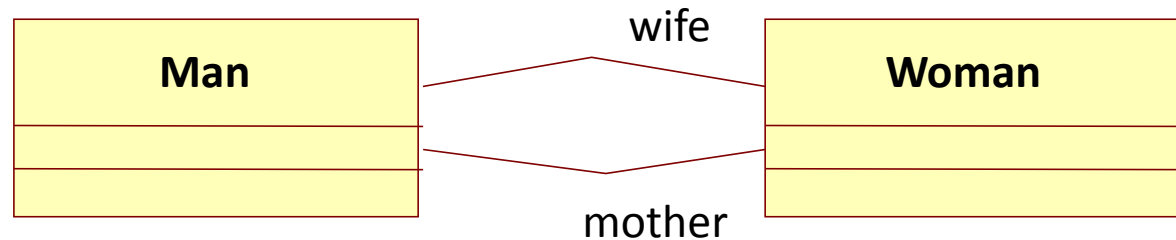
- In Java

- Storing reference
- Lifecycle is taken care of by GC

- In C++

- Storing pointer to object
- Lifecycle is taken care of by others
 - no delete in destructor

Association



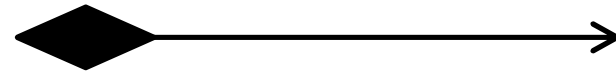
```
// JAVA
class Woman {}
class Man {
    Woman wife;
    final Woman mother;
}
```

```
// C++
class Woman {};
class Man {
    Woman* wife;
    Woman& mother;
public:
    ~Man() {}
};
```

Aggregation / Composition

■ Notation

- solid line with diamond



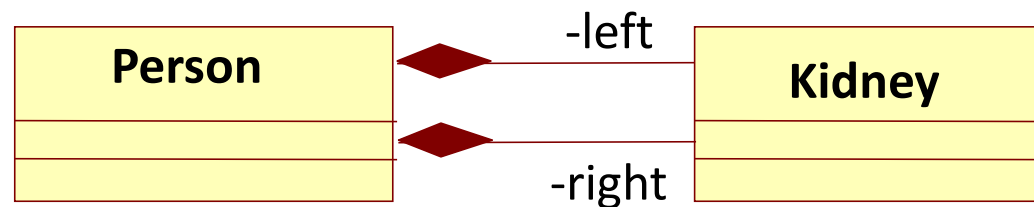
■ In Java

- Storing reference
- Lifecycle is taken care of by GC

■ In C++

- Storing pointer to object
 - Lifecycle must be taken care of
 - delete in destructor
- Storing object directly

Aggregation / Composition



```
// JAVA
class Kidney {}
class Person {
    Kidney left;
    Kidney right;
}
```

```
// C++
class Kidney {}
class Person {
    Kidney* left;
    Kidney right;
public:
    ~Person() {delete left;}
}
```

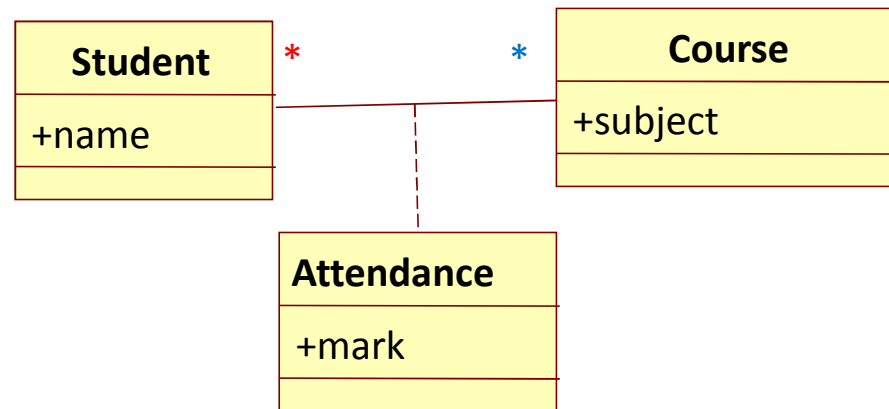



Association vs Aggregation

- Memory handling differ in implementations
 - pointers in C++
 - delete is needed somewhere ☹️
 - references in Java
 - GC is our friend 😊
- Association vs. Aggregation
 - Java: no difference on code level
 - analysis and design concepts reduced
 - C++: difference in life cycle management
 - maintains analysis and design decisions

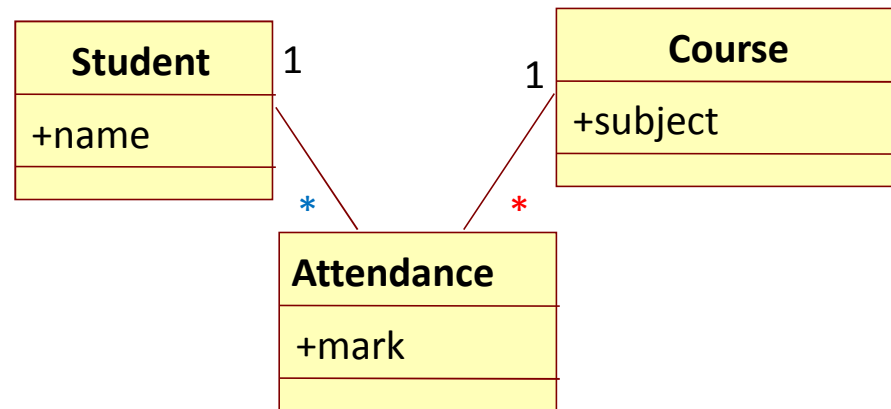
Association class

- Represents responsibilities of an association
 - not assignable to any party
 - *e.g. Student attends Course*
 - analysis level, OO languages usually do not support it



Association class

- Design and implementation
 - join class
 - 1-n multiplicity on Association class

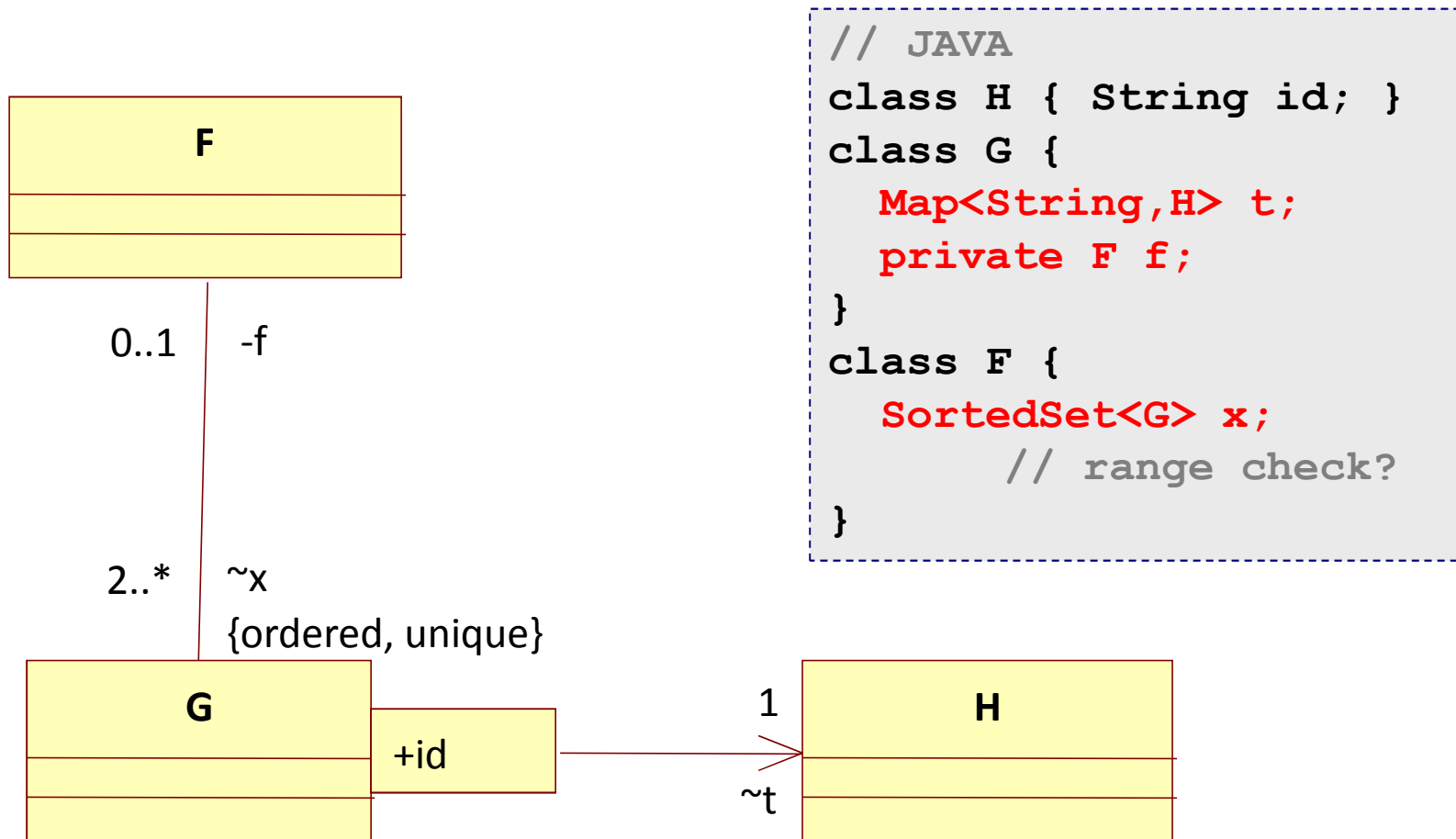




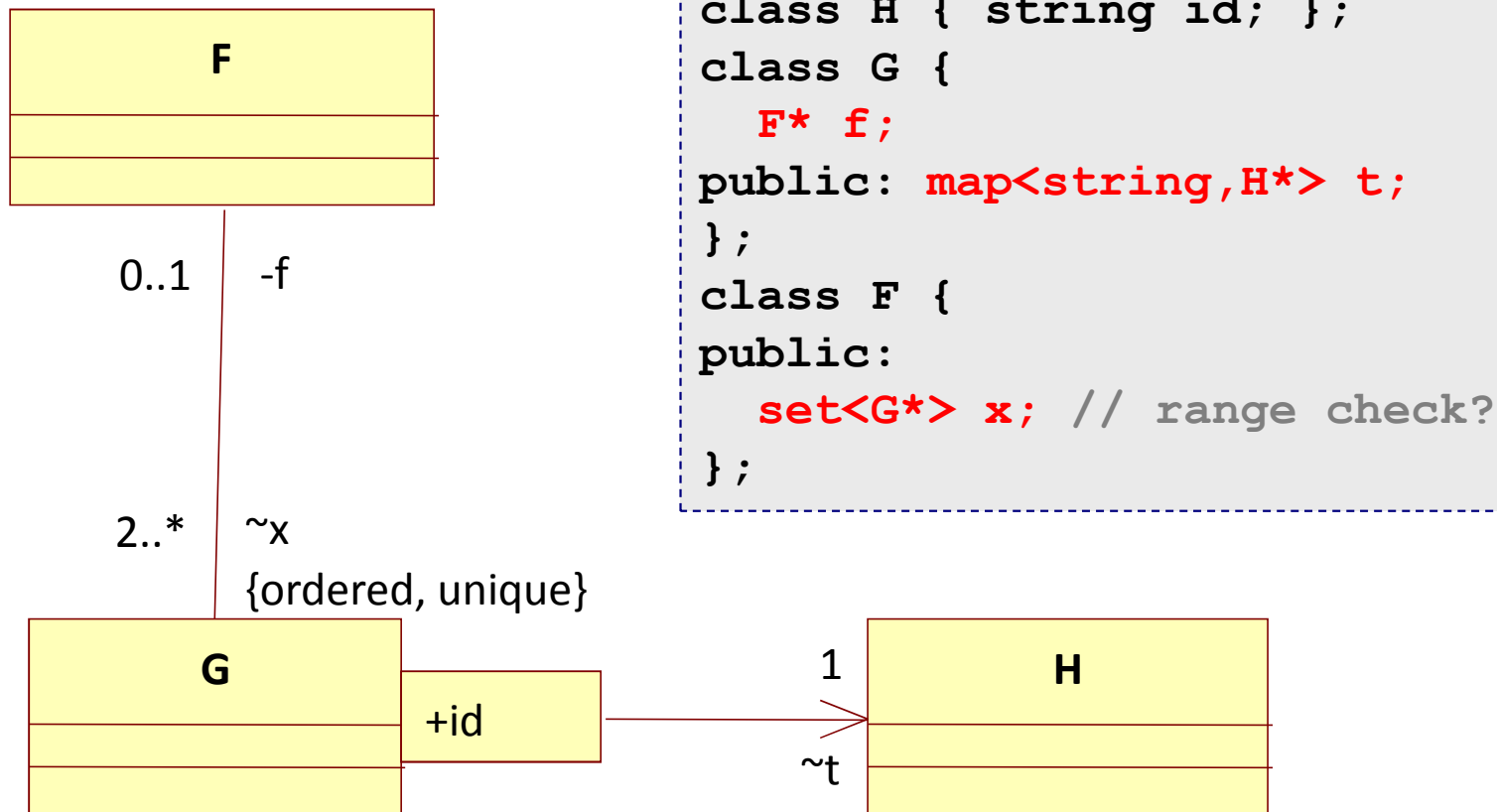
Associations' attributes

- Cardinality (multiplicity)
 - ordinal: 1 *reference*
 - optional: 0..1 *reference*
 - range: 2..5 *collection (range check?)*
 - unlimited: * *collection*
- Qualifier *associative array (map, etc)*
- Other
 - unique *set*
 - ordered *list*

Associations' attributes



Associations' attributes



```
// C++
class H { string id; };
class G {
    F* f;
public: map<string,H*> t;
};
class F {
public:
    set<G*> x; // range check?
};
```

Inheritance

- Notation

- solid line with triangle



- Java

- single inheritance between classes
- public inheritance only

- C++

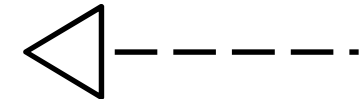
- multiple inheritance
 - virtual inheritance
- public and private inheritance
 - in UML, for private use stereotypes

Interfaces

■ Notation

- stereotype <<interface>>

- implementation: dashed line with triangle



- lollipop 

■ Java

- separate meta-type

- multiple inheritance for interfaces

■ C++

- no separate meta-type

- all methods pure virtual



Abstract classes vs. Interfaces

■ Interfaces

- method signatures only
- use when
 - implementations might vary, but are out of scope
 - access of functionality must be separated

■ Abstract classes

- might contain code
- use when
 - default implementation is needed in superclass
 - hook methods are needed in subclasses



Abstract classes vs. Interfaces

■ UML

- abstract class (method): name in *italic*
- interface: stereotype <<interface>> or lollipop

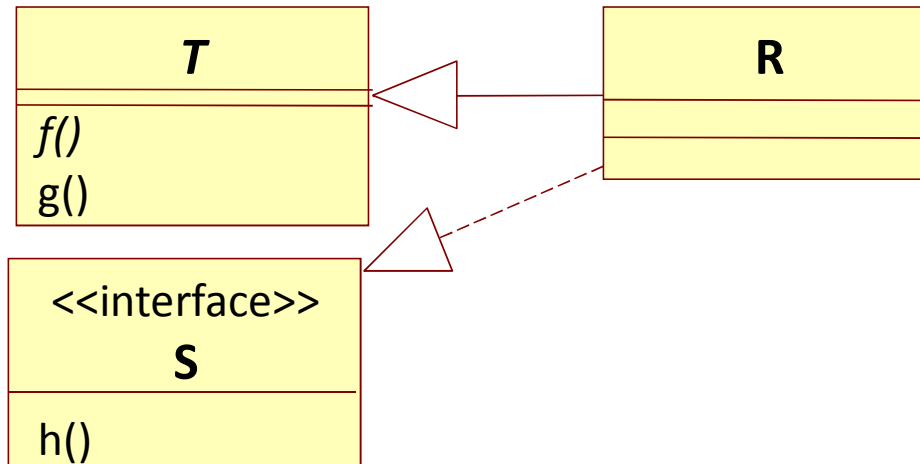
■ Java

- separate meta-types for abstract and interface
- multiple inheritance for interfaces
 - separate expected functionality with interfaces

■ C++

- no explicit abstract or interface type
- pure virtual methods, no distinction

Abstract classes and interfaces



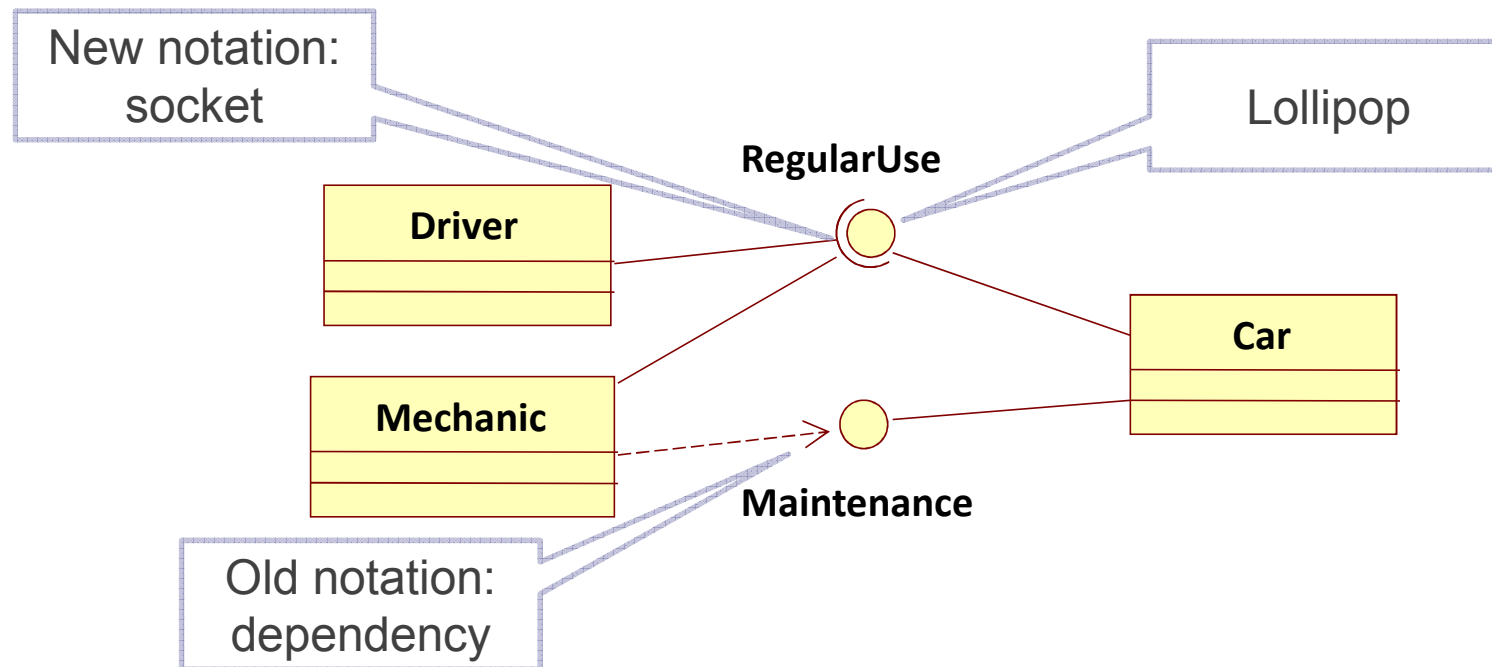
Visibility is omitted!

```
// JAVA
abstract class T {
    abstract void f();
    void g() { ... }
}
interface S { void h(); }
class R extends T
    implements S {...}
```

```
// C++
class T {
    virtual void f() = 0;
    void g() { ... }
};
class S {
    virtual void h() = 0;
};
class R : public T,
        public S {...};
```

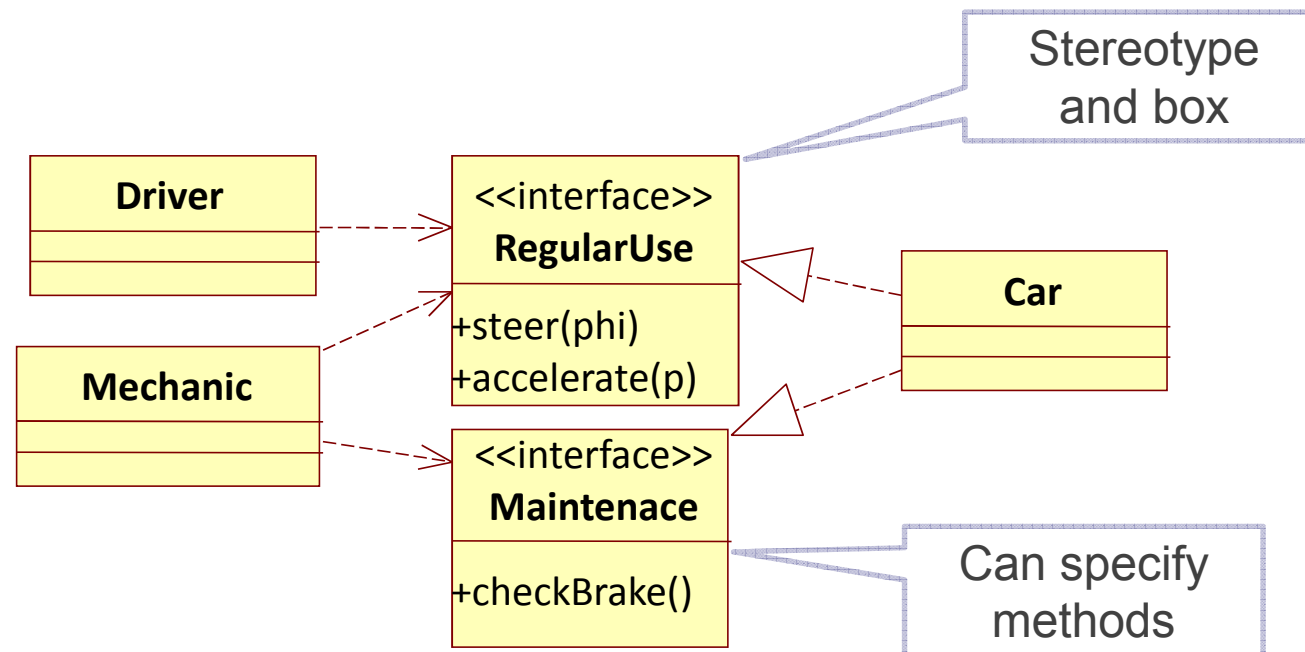
Interface for separating access

- Car is accessed differently by people
 - driver must not maintain car ☹️



Interface for separating access

- Car is accessed differently by people
 - driver must not maintain car ☹️





Inheritance: method override

- Overriding methods
 - modifies inherited behaviour
- UML best practice
 - in subclasses only show overridden methods
 - when superclass is visible on diagram
- Java best practice
 - use `@Override` annotation in subclass
- C++
 - method must be *virtual* in superclass



Behaviour

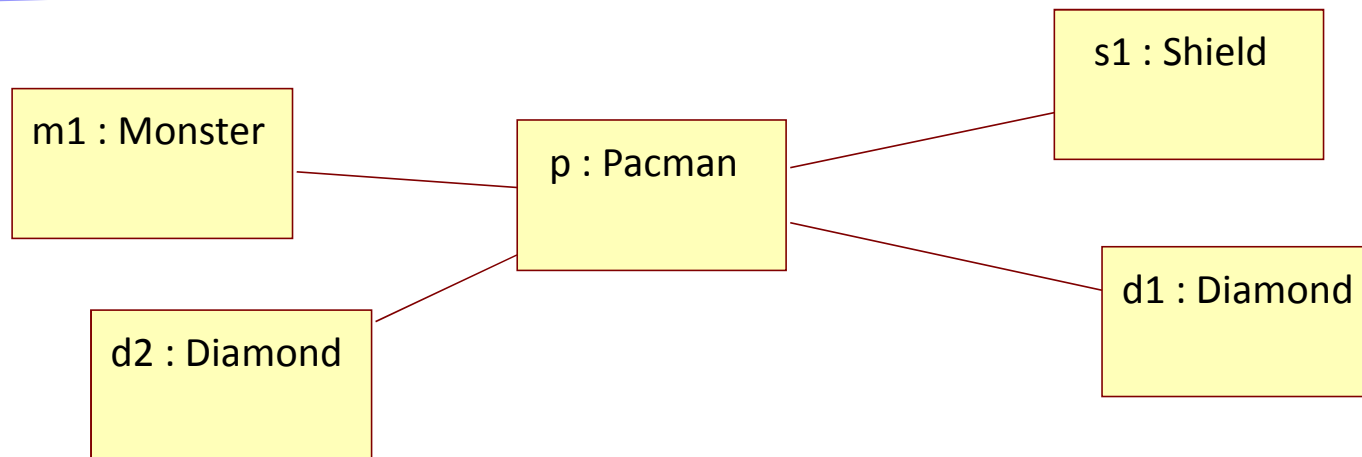
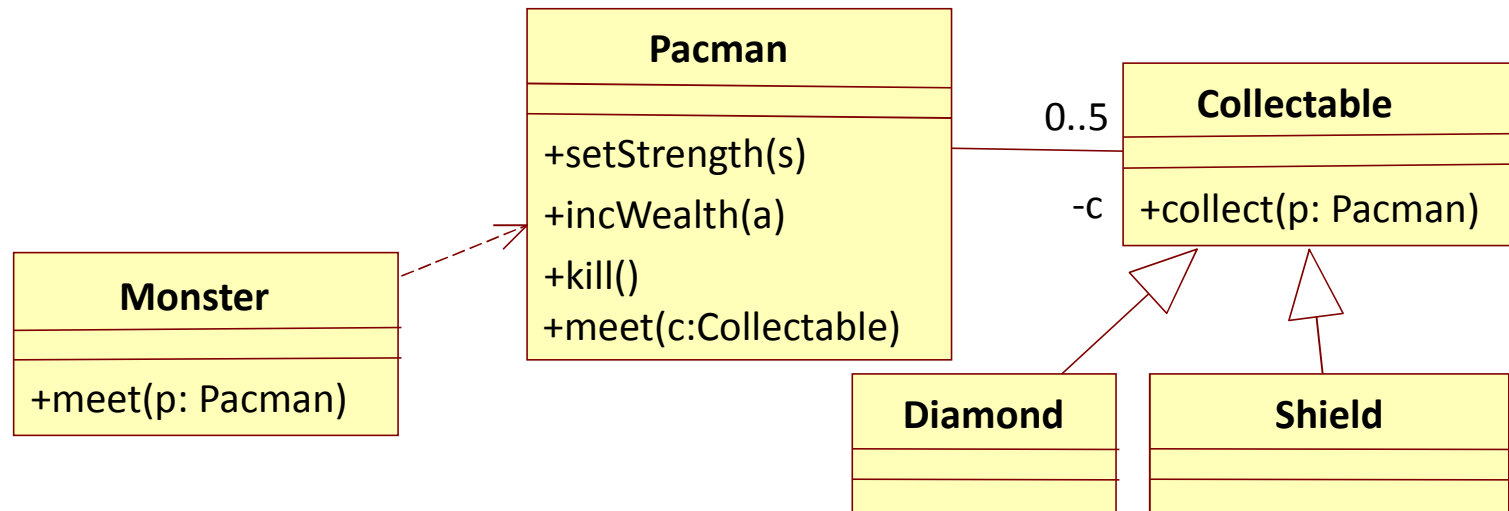
- Static structure is captured by class diagrams
 - show all connections
 - closely related to source code
- Behaviour is implemented in methods
 - methods are executed during runtime
 - method bodies are omitted from class diagram
- Internal behaviour (intra-object)
 - mostly state chart, seldom activity diagram, etc.
- External behaviour (inter-object)
 - interaction diagrams (sequence and communication)



Modelling behaviour

- Communication diagram
 - represents objects
 - and relationships between objects
 - relationship is static
 - if no communication, called object diagram
 - and communication between objects
 - communication is dynamic
- Objects are instances of classes
 - object diagram is an instance of class diagram

Class and Object diagram





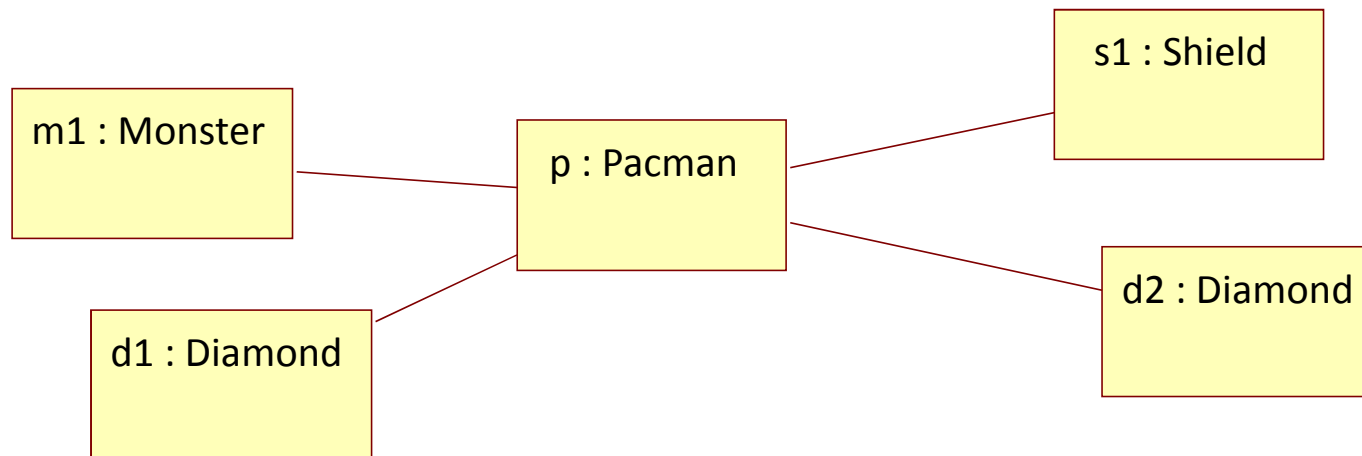
Java implementation (partial)

```
class Monster {
    public void meet(Pacman p) {
        p.kill();
    }
}
interface Collectable {
    boolean collect(Pacman p);
}
class Diamond implements Collectable
{
    double value;
    public boolean collect(Pacman p) {
        p.incWealth(value);
    }
}
class Shield implements Collectable
{...}
```

```
class Pacman {
    double strength;
    double wealth;
    public void
    meet(Collectable c) {
        if (n<5) if
            (c.collect(this))
            n++;
    }
    public void
    intWealth(double w) {
        wealth += w;
    }
    ...
}
```

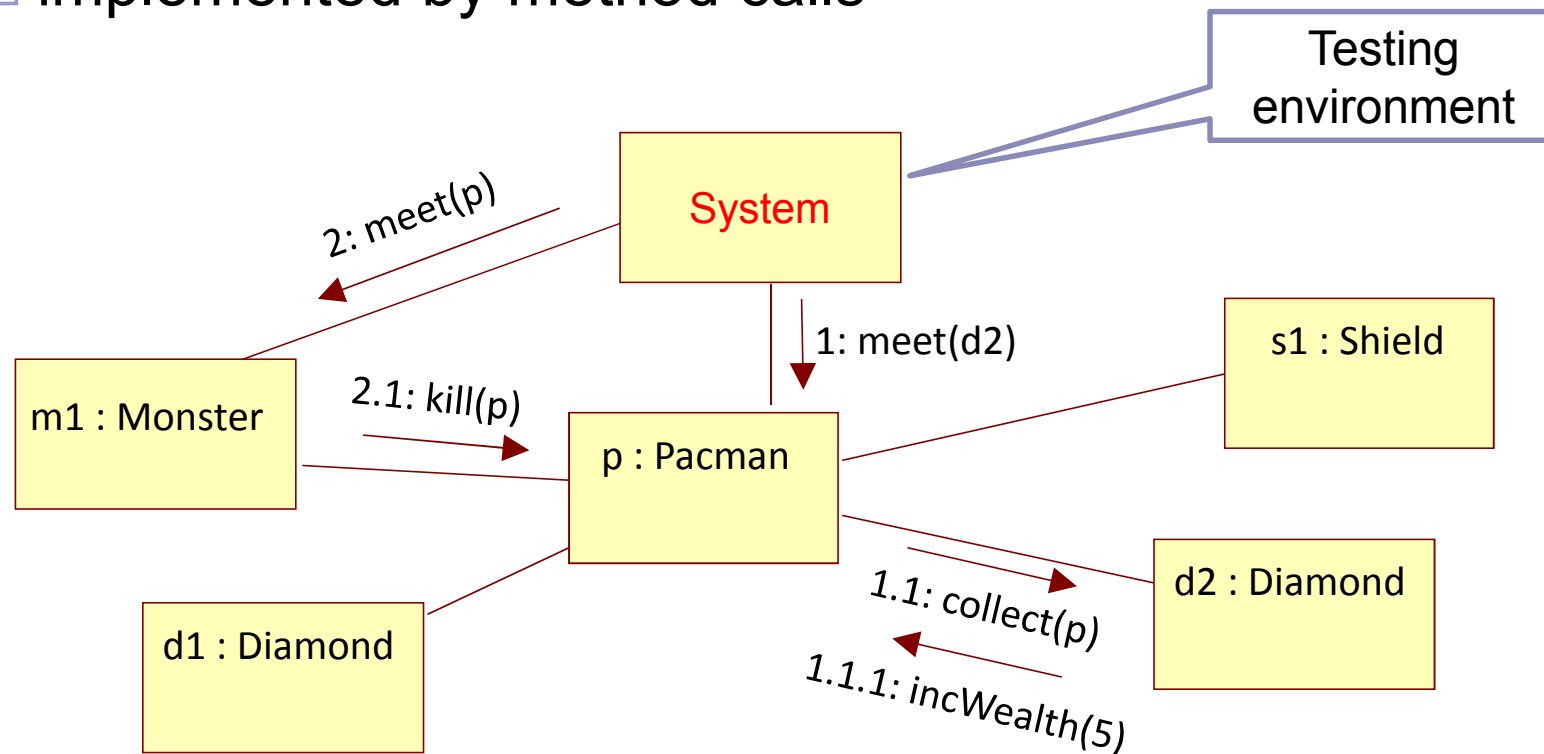
Java implementation (objects)

```
Monster m1 = new Monster();  
Diamond d1 = new Diamond();  
Diamond d2 = new Diamond();  
Shield s1 = new Shield();  
Pacman p = new Pacman();  
// setting up connections is omitted here
```



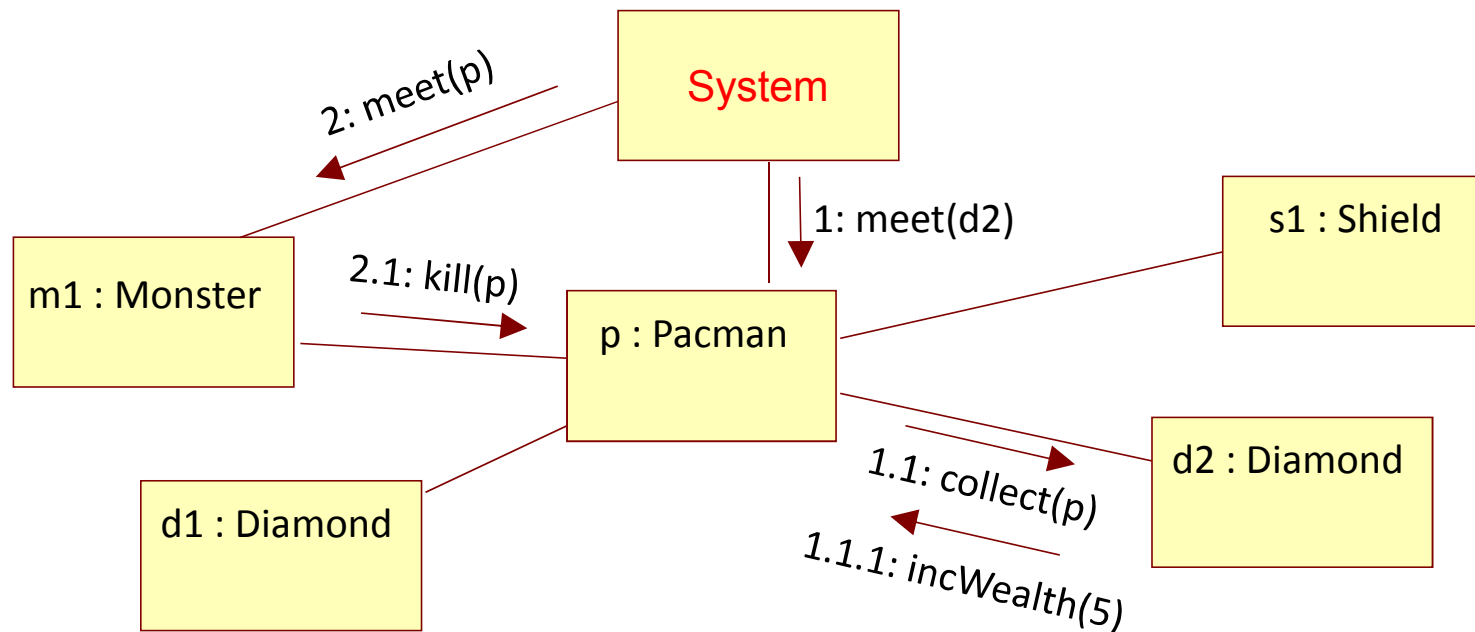
Communication (collaboration)

- Links may have messages
 - implemented by method calls



Communication implemented

```
// objects created, links initialized, then ...  
p.meet(d2);  
    // p.meet(d2) calls d2.collect(this)  
    // d2.collect(this) calls p.incWealth(5)  
m1.meet(p);  
    // m1.meet(p) calls p.kill();
```

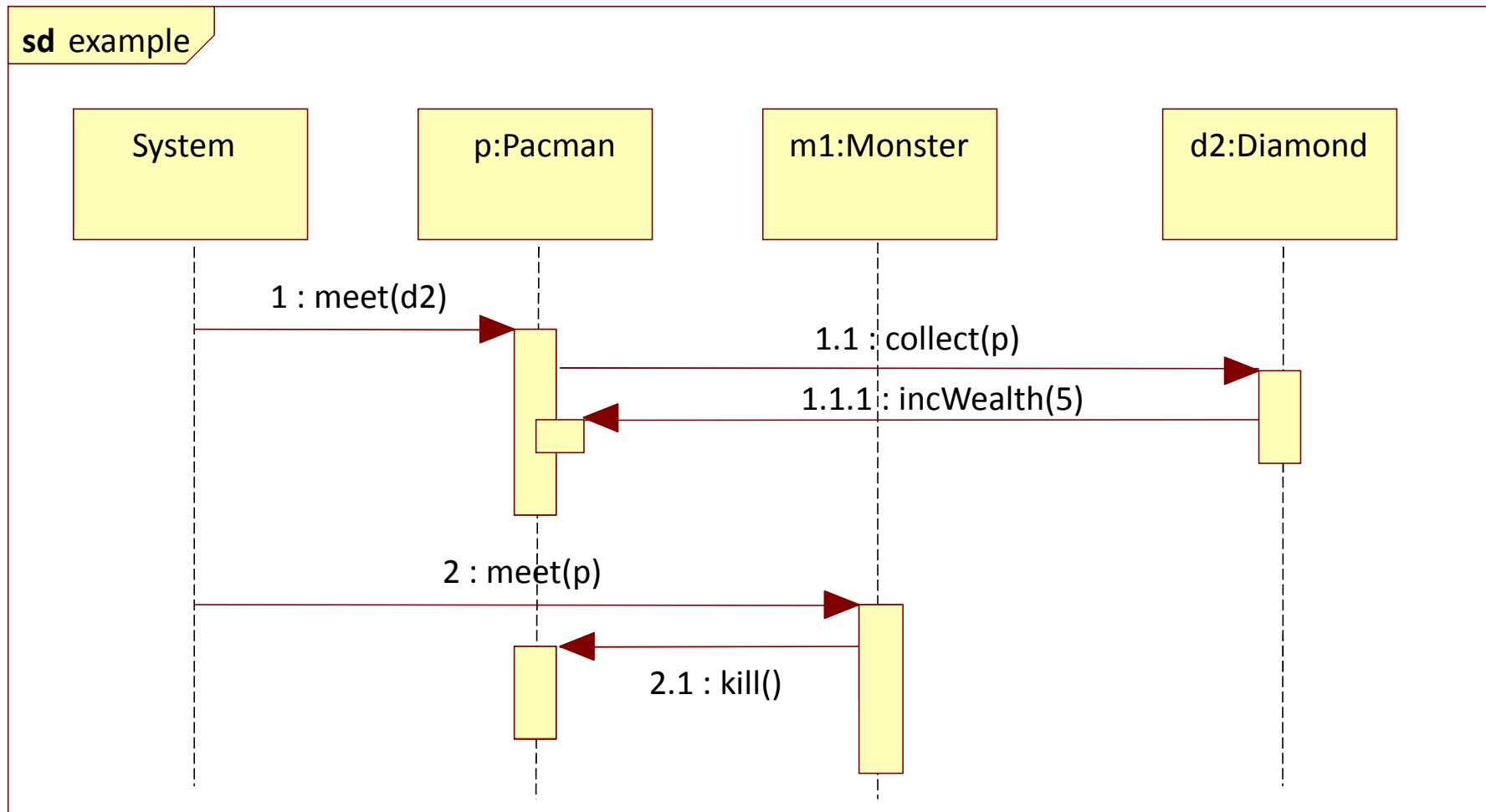




Modelling behaviour 2

- Sequence diagram
 - represents object life-lines
 - sequence of communication between objects
- Activity diagram
 - describes external and internal behaviour
- State chart
 - represent internal behaviour
 - stimuli are mostly method calls
 - state representation is arbitrary

Sequence diagram





How concepts correlate

- Class diagram
 - all methods should appear on interaction diagrams
- Sequence and communication diagrams
 - all messages should appear on class diagrams
- Source code
 - class structure resembles class diagram
 - method signatures, attributes, inheritance, associations
 - method bodies implement dynamics
 - interaction diagrams represent the execution of methods
 - state-charts and activity diagrams mostly for internals