Contents

Chapter 1

Virtual Functions and Late Bindings

1.1 Programmazione orientata agli oggetti

La programmazione ad oggetti è diversa ad altri tipi di programmazione (tipo askle che è un linguaggio funzionale, o l'approccio imperativo, ovvero un approccio in cui si dice al computer cosa fare passo passo attraveroso un a serie di comandi) è quella di rappresentare un mondo un cui degli elementi (oggetti) interagiscono tra di loro.

La programmazione ad oggetti è diversa ad altri tipi di programmazione (tipo askle che è un linguaggio funzionale, o l'approccio imperativo, ovvero un approccio in cui si dice al computer cosa fare passo passo attraveroso un a serie di comandi) è quella di rappresentare un mondo un cui degli elementi (oggetti) interagiscono tra di loro.

Alcuni concetti fondamentali della programmazione ad oggetti sono:

- **Classe**: è un modello, uno schema, una struttura che definisce le proprietà e i comportamenti comuni degli oggetti di quel tipo. Una classe può essere vista come un "progetto" o un "blueprint" per creare oggetti
- **Ereditarietà**: è un meccanismo che consente a una classe di ereditare le proprietà e i comportamenti di un'altra classe. La classe che eredita è chiamata "classe derivata" o "sottoclasse", mentre la classe da cui si eredita è chiamata "classe base" o "superclasse". L'ereditarietà permette di creare gerarchie di classi e di riutilizzare il codice.
- **Polimorfismo**: è la capacità di un oggetto di assumere diverse forme o comportamenti a seconda del contesto in cui viene utilizzato. In altre parole, il polimorfismo consente a un oggetto di essere trattato come un'istanza di una classe base, ma di eseguire il comportamento specifico della sua classe derivata. Si divide in:
  - **Polimorfismo ad hoc**: si riferisce alla capacità di una funzione o di un metodo di avere lo stesso nome ma comportarsi in modo diverso a seconda del tipo o del numero di argomenti passati. Questo è spesso realizzato attraverso l'overloading dei metodi.
  - **Polimorfismo di sottotipo**: ovvero la capacità di un oggetto di una classe derivata di essere trattato come un'istanza della sua classe base. Questo è spesso realizzato attraverso l'overriding dei metodi.
  - **Polimorfismo parametrico**: si riferisce alla capacità di una funzione o di un metodo di operare su tipi generici, consentendo di scrivere codice che può essere riutilizzato con diversi tipi di dati. Questo è spesso realizzato attraverso l'uso di generics o template. Ad esempio

1.2 Funzioni virtuali

Definition 1.2.1: Funzione virtuale

una funzione si definisce virtuale un metdo non statico di una classe base che può essere **ridefinita** (overridden) in una classe derivata

1.2.1 Overriding

Definition 1.2.2: Overriding

l'ovverinding è la metodologia tramite la quale vado a sostiuire il corpo di un metodo con uno nuovo a runtime

l'overriding si applica in tali circostanze:

- il metodo overridden è una funzione virtuale
- la funzione nella classe derivata ha la stessa signature della funzione nella classe base

Si ricorda che la **signature** di una funzione è composta da:

- Nome della funzione
- parametri formali
- numero di parametri formali
- il *tipo di ritorno* non fa parte della signature

La signature di una funzione con il suo *tipo di ritorno* è detta **prototipo** della funzione.
I metodi statici non posso no essere sottoposti a overriding in quanto non sono associati ad un'istanza della classe ma alla classe stessa. se si dichiara, inoltre, nella classe derivata una funzione con la stessa signature ma con un tipo diverso il compilatore restituirà errore. In c++ se io dichiaro una classe derivata con lo stesso nome cma con una signature io precludo la visita della funzione della classe base. Si noti tale affermazione:

Note:

Quando si invoca una funzione virtuale per un oggetto il cui tipo è conosciuto a tempo di compilazione, il comportamento della funzione, dal punto di vista dell'utente, è lo stesso di quello di una funzione non virtuale.

Proposition 1.2.1 sulle funzioni virtuali

Quando una funzione virtuale f è invocata per un oggetto o della classe D derivata da B ci snoo tre possibilità:

- f è definita solo in D: viene invocata la versione di $D.f$ da o
- f è definita solo in B: viene invocata la versione di $B.f$ da o
- f è definita sia in B che in D: viene invocata la versione di $D.f$ da o

il vantaggio è che quando di usare funzioni è quello di manipolare gli oggetti tramite riferimenti o puntatori alla classe base. In questo modo, è possibile scrivere codice più generico e riutilizzabile, poiché non è necessario conoscere il tipo esatto dell'oggetto a cui si sta facendo riferimento.

Esempi

```
class B {
    public:
        virtual void f() { cout << "B::f" << endl; } // funzione virtuale
};

class D : public B {
    public:
        void f() override { cout << "D::f" << endl; } // overriding della funzione virtuale
};
int main() {
    B* b = new B();
```

```
    B* d = new D();

    b->f(); // Output: B::f
    d->f(); // Output: D::f (polimorfismo in azione)

    delete b;
    delete d;
    return 0;
}
```

Funzioni virtuali in Java e C++

Java in Javam tutti i metodi non statici sono virtuali di default, quindi non è necessario dichiararli esplicitamente come virtuali. Tuttavia, è possibile utilizzare la parola chiave `final` per impedire che un metodo venga sovrascritto in una sottoclasse. ad esempio

```
class B {
    public void f() { System.out.println("B::f"); } // funzione virtuale
}
class D extends B {
    @Override
    public void f() { System.out.println("D::f"); } // overriding della funzione virtuale
}
public class Main {
    public static void main(String[] args) {
        B b = new B();
        B d = new D();
        b.f(); // Output: B::f
        d.f(); // Output: D::f (polimorfismo in azione)
    }
}
```

C++ in C++ le funzioni non sono virtuali di default, quindi è necessario dichiararle esplicitamente come virtuali utilizzando la parola chiave `virtual`. Ad esempio

```
class B {
    public:
        virtual void f() { cout << "B::f" << endl; } // funzione virtuale
};
class D : public B {
    public:
        void f() override { cout << "D::f" << endl; } // overriding della funzione virtuale
};
int main() {
    B* b = new B();
    B* d = new D();
    b->f(); // Output: B::f
    d->f(); // Output: D::f (polimorfismo in azione)
    delete b;
    delete d;
    return 0;
}
```

1.2.2 Early binding

Definition 1.2.3: Early binding

l'early binding è il processo di associazione di una chiamata a funzione con la sua definizione durante la fase di compilazione del programma

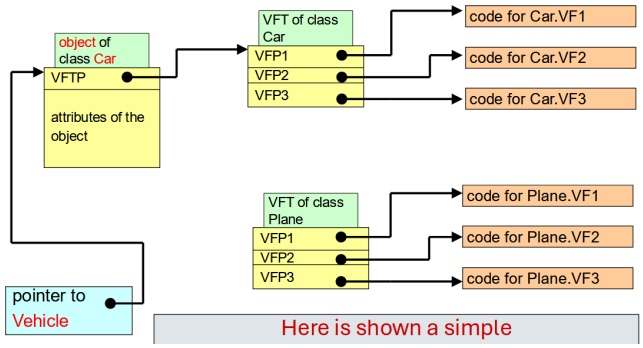
It's the method used by most non OO languages, and it's also the default behaviour for functions and methods in C++ (if not declared as virtual).

1.2.3 Late binding

Definition 1.2.4: Late binding

The correct address of the code to execute after a funciton call is deduced only at runtime.

Late binding can be implemented by simply using a VTABLE, or a table of virtual function pointers relative to each class:



Each object has a hidden pointer to the virtual funciton table (VTABLE) of it's class.
In short, the following steps are performed for every virtual funcion call:

1. The program looks for the VFTP in the object
2. It then searches the table for the correct VFP using the funciton name
3. If the pointer is found, the code pointed to is executed

The reason why not all function calls are virtual in most languages is because of eficiency, although in most cases the overhead generated by virtualization is negligible.

1.3 Examples

```
#include <iostream>
#include <string>
void prn( std::string s) {
    std::cout << s << "\n";
}
class A {
  public:
    virtual void f() { prn("A::f()"); }
    void g() { prn("A::g()"); }
    virtual void h() { prn("A::h()"); }
    void i() { prn("A::i()"); }
};
class B : public A {
```

```
  public:
    virtual void f() { prn("B::f()"); }
    virtual void g() { prn("B::g()"); }
    void h() { prn("B::h()"); }
    void i() { prn("B::i()"); }
};
class C : public B{
  public:
    virtual void f() { prn("C::f()"); }
    void g() { prn("C::g()"); }
    void h() { prn("C::h()"); }
    void i() { prn("C::i()"); }
};
class D : public A {
  public:
    virtual void h() { prn("D::h()"); }
    void m() {
        prn("D::m()");
        h();
    }
    virtual void g() { prn("D::g()"); }
};
class E : public D {
  public:
    virtual void f() { prn("E::f()"); }
    virtual void h() { prn("E::h()"); }
    void i() {
        prn("E::i()");
        m();
    }
    void g() { prn("E::g()"); }
};
class G : public E {
  public:
    virtual void f(int i) { prn("G::f(int)"); }
};

int main() {
    A anA;
    A *a;
    B aB;
    B *b;
    a = new B();
    a->f();
    a->g();
    b = (B*)a;
    b->f();
    b->g();
    A &arA = *new E();
    arA.i();
    E anE;
    anE.i();
    arA.g();
    D *apD = &anE;
    apD->g();
    A *apA = &anE;
```

```
    apA->g();
    G aG;
    aG.f();
    aG.f(3);
    aG.h();
    aG.i();
    aG.g();
    a = &aG;
    a->f();
    a->f(3);
    a->h();
    a->i();
    a->g();
    G *pG;
    pG = (G*)a;
    pG->f();
    pG->f(3);
    pG->h();
    pG->i();
    pG->g();
}
```

Given the C++ classes defined above, here are some interesting function calls to look at:

- a->f(): at runtime, a is a pointer to an object of class B. Seen as f is declared virtual in A it is overridden by the B class and "B::f()" is output.
- a->g(): because g is not declared as virtual in A, early binding is used and the function declared in A is used and "A::g()" is output.
- aG->f(): a function with signature f() is decalred in class G's superclass (E), but G also defines a function with name f but with a different signature (f(int)). Thus the homonymous function in the superclass is hidden and an error occurs.
- a = &aG; a->f(): f() is virtual in A, so late binding is used. f(int) is defined in G but its signatura doesn't match, so E's definition for f() is used instead.
- a = &aG; a->f(int): because A doesn't have any definition for a function with signature f(int), an error is thrown.

La programmazione ad oggetti è diversa ad altri tipi di programmazione (tipo askle che è un linguaggio funzionale, o l'approccio imperativo, ovvero un approccio in cui si dice al computer cosa fare passo passo attraveroso un a serie di comandi) è quella di rappresentare un mondo un cui degli elementi (oggetti) interagiscono tra di loro.

Alcuni concetti fondamentali della programmazione ad oggetti sono:

- **Classe**: è un modello, uno schema, una struttura che definisce le proprietà e i comportamenti comuni degli oggetti di quel tipo. Una classe può essere vista come un "progetto" o un "blueprint" per creare oggetti
- **Ereditarietà**: è un meccanismo che consente a una classe di ereditare le proprietà e i comportamenti di un'altra classe. La classe che eredita è chiamata "classe derivata" o "sottoclasse", mentre la classe da cui si eredita è chiamata "classe base" o "superclasse". L'ereditarietà permette di creare gerarchie di classi e di riutilizzare il codice.
- **Polimorfismo**: è la capacità di un oggetto di assumere diverse forme o comportamenti a seconda del contesto in cui viene utilizzato. In altre parole, il polimorfismo consente a un oggetto di essere trattato come un'istanza di una classe base, ma di eseguire il comportamento specifico della sua classe derivata. Si divide in:

- **Polimorfismo ad hoc**: si riferisce alla capacità di una funzione o di un metodo di avere lo stesso nome ma comportarsi in modo diverso a seconda del tipo o del numero di argomenti passati. Questo è spesso realizzato attraverso l'overloading dei metodi.
- **Polimorfismo di sottotipo**: ovvero la capacità di un oggetto di una classe derivata di essere trattato come un'istanza della sua classe base. Questo è spesso realizzato attraverso l'overriding dei metodi.
- **Polimorfismo parametrico**: si riferisce alla capacità di una funzione o di un metodo di operare su tipi generici, consentendo di scrivere codice che può essere riutilizzato con diversi tipi di dati. Questo è spesso realizzato attraverso l'uso di generics o template. Ad esempio

1.4 Funzioni virtuali

Definition 1.4.1: Funzione virtuale

una funzione si definisce virtuale un metdo non statico di una classe base che può essere **ridefinita** (overridden) in una classe derivata

1.4.1 Overriding

Definition 1.4.2: Overriding

l'ovverinding è la metodologia tramite la quale vado a sostiuire il corpo di un metodo con uno nuovo a runtime

l'overriding si applica in tali circostanze:

- il metodo overridden è una funzione virtuale
- la funzione nella classe derivata ha la stessa signature della funzione nella classe base

Si ricorda che la **signature** di una funzione è composta da:

- Nome della funzione
- parametri formali
- numero di parametri formali
- il *tipo di ritorno* non fa parte della signature

La signature di una funzione con il suo *tipo di ritorno* è detta **prototipo** della funzione.
I metodi statici non posso no essere sottoposti a overriding in quanto non sono associati ad un'istanza della classe ma alla classe stessa. se si dichiara, inoltre, nella classe derivata una funzione con la stessa signature ma con un tipo diverso il compilatore restituirà errore. In c++ se io dichiaro una classe derivata con lo stesso nome cma con una signature io precludo la visita della funzione della classe base. Si noti tale affermazione:

Note:

Quando si invoca una funzione virtuale per un oggetto il cui tipo è conosciuto a tempo di compilazione, il comportamento della funzione, dal punto di vista dell'utente, è lo stesso di quello di una funzione non virtuale.

Proposition 1.4.1 sulle funzioni virtuali

Quando una funzione virtuale f è invocata per un oggetto o della classe D derivata da B ci snoo tre possibilità:

- f è definita solo in D: viene invocata la versione di $D.f$ da o
- f è definita solo in B: viene invocata la versione di $B.f$ da o
- f è definita sia in B che in D: viene invocata la versione di $D.f$ da o

il vantaggio è che quando di usare funzioni è quello di manipolare gli oggetti tramite riferimenti o puntatori alla classe base. In questo modo, è possibile scrivere codice più generico e riutilizzabile, poiché non è necessario conoscere il tipo esatto dell'oggetto a cui si sta facendo riferimento.

Esempi

```
class B {
    public:
        virtual void f() { cout << "B::f" << endl; } // funzione virtuale
};

class D : public B {
    public:
        void f() override { cout << "D::f" << endl; } // overriding della funzione virtuale
};
int main() {
    B* b = new B();
    B* d = new D();

    b->f(); // Output: B::f
    d->f(); // Output: D::f (polimorfismo in azione)

    delete b;
    delete d;
    return 0;
}
```

Funzioni virtuali in Java e C++

Java in Javam tutti i metodi non statici sono virtuali di default, quindi non è necessario dichiararli esplicitamente come virtuali. Tuttavia, è possibile utilizzare la parola chiave `final` per impedire che un metodo venga sovrascritto in una sottoclasse. ad esempio

```
class B {
    public void f() { System.out.println("B::f"); } // funzione virtuale
}
class D extends B {
    @Override
    public void f() { System.out.println("D::f"); } // overriding della funzione virtuale
}
public class Main {
    public static void main(String[] args) {
        B b = new B();
        B d = new D();
        b.f(); // Output: B::f
        d.f(); // Output: D::f (polimorfismo in azione)
    }
}
```

C++ in C++ le funzioni non sono virtuali di default, quindi è necessario dichiararle esplicitamente come virtuali utilizzando la parola chiave `virtual`. Ad esempio

```
class B {
    public:
        virtual void f() { cout << "B::f" << endl; } // funzione virtuale
};
```

```
class D : public B {
    public:
        void f() override { cout << "D::f" << endl; } // overriding della funzione virtuale
};
int main() {
    B* b = new B();
    B* d = new D();
    b->f(); // Output: B::f
    d->f(); // Output: D::f (polimorfismo in azione)
    delete b;
    delete d;
    return 0;
}
```

1.4.2 Early binding

Definition 1.4.3: Early binding

l'early binding è il processo di associazione di una chiamata a funzione con la sua definizione durante la fase di compilazione del programma

Chapter 2

Virtual Functions and Late Bindings

2.1 Programmazione orientata agli oggetti

La programmazione ad oggetti è diversa ad altri tipi di programmazione (tipo askle che è un linguaggio funzionale, o l'approccio imperativo, ovvero un approccio in cui si dice al computer cosa fare passo passo attraveroso un a serie di comandi) è quella di rappresentare un mondo un cui degli elementi (oggetti) interagiscono tra di loro.

La programmazione ad oggetti è diversa ad altri tipi di programmazione (tipo askle che è un linguaggio funzionale, o l'approccio imperativo, ovvero un approccio in cui si dice al computer cosa fare passo passo attraveroso un a serie di comandi) è quella di rappresentare un mondo un cui degli elementi (oggetti) interagiscono tra di loro.

Alcuni concetti fondamentali della programmazione ad oggetti sono:

- **Classe**: è un modello, uno schema, una struttura che definisce le proprietà e i comportamenti comuni degli oggetti di quel tipo. Una classe può essere vista come un "progetto" o un "blueprint" per creare oggetti
- **Ereditarietà**: è un meccanismo che consente a una classe di ereditare le proprietà e i comportamenti di un'altra classe. La classe che eredita è chiamata "classe derivata" o "sottoclasse", mentre la classe da cui si eredita è chiamata "classe base" o "superclasse". L'ereditarietà permette di creare gerarchie di classi e di riutilizzare il codice.
- **Polimorfismo**: è la capacità di un oggetto di assumere diverse forme o comportamenti a seconda del contesto in cui viene utilizzato. In altre parole, il polimorfismo consente a un oggetto di essere trattato come un'istanza di una classe base, ma di eseguire il comportamento specifico della sua classe derivata. Si divide in:
  - **Polimorfismo ad hoc**: si riferisce alla capacità di una funzione o di un metodo di avere lo stesso nome ma comportarsi in modo diverso a seconda del tipo o del numero di argomenti passati. Questo è spesso realizzato attraverso l'overloading dei metodi.
  - **Polimorfismo di sottotipo**: ovvero la capacità di un oggetto di una classe derivata di essere trattato come un'istanza della sua classe base. Questo è spesso realizzato attraverso l'overriding dei metodi.
  - **Polimorfismo parametrico**: si riferisce alla capacità di una funzione o di un metodo di operare su tipi generici, consentendo di scrivere codice che può essere riutilizzato con diversi tipi di dati. Questo è spesso realizzato attraverso l'uso di generics o template. Ad esempio

2.2 Funzioni virtuali

Definition 2.2.1: Funzione virtuale

una funzione si definisce virtuale un metdo non statico di una classe base che può essere **ridefinita** (overridden) in una classe derivata

2.2.1 Overriding

Definition 2.2.2: Overriding

l'ovverinding è la metodologia tramite la quale vado a sostiuire il corpo di un metodo con uno nuovo a runtime

l'overriding si applica in tali circostanze:

- il metodo overridden è una funzione virtuale
- la funzione nella classe derivata ha la stessa signature della funzione nella classe base

Si ricorda che la **signature** di una funzione è composta da:

- Nome della funzione
- parametri formali
- numero di parametri formali
- il *tipo di ritorno* non fa parte della signature

La signature di una funzione con il suo *tipo di ritorno* è detta **prototipo** della funzione.
I metodi statici non posso no essere sottoposti a overriding in quanto non sono associati ad un'istanza della classe ma alla classe stessa. se si dichiara, inoltre, nella classe derivata una funzione con la stessa signature ma con un tipo diverso il compilatore restituirà errore. In c++ se io dichiaro una classe derivata con lo stesso nome cma con una signature io precludo la visita della funzione della classe base. Si noti tale affermazione:

Note:

Quando si invoca una funzione virtuale per un oggetto il cui tipo è conosciuto a tempo di compilazione, il comportamento della funzione, dal punto di vista dell'utente, è lo stesso di quello di una funzione non virtuale.

Proposition 2.2.1 sulle funzioni virtuali

Quando una funzione virtuale f è invocata per un oggetto o della classe D derivata da B ci snoo tre possibilità:

- f è definita solo in D: viene invocata la versione di $D.f$ da o
- f è definita solo in B: viene invocata la versione di $B.f$ da o
- f è definita sia in B che in D: viene invocata la versione di $D.f$ da o

il vantaggio è che quando di usare funzioni è quello di manipolare gli oggetti tramite riferimenti o puntatori alla classe base. In questo modo, è possibile scrivere codice più generico e riutilizzabile, poiché non è necessario conoscere il tipo esatto dell'oggetto a cui si sta facendo riferimento.

Esempi

```
class B {
    public:
        virtual void f() { cout << "B::f" << endl; } // funzione virtuale
};

class D : public B {
    public:
        void f() override { cout << "D::f" << endl; } // overriding della funzione virtuale
};
int main() {
    B* b = new B();
```

```
    B* d = new D();

    b->f(); // Output: B::f
    d->f(); // Output: D::f (polimorfismo in azione)

    delete b;
    delete d;
    return 0;
}
```

Funzioni virtuali in Java e C++

Java in Javam tutti i metodi non statici sono virtuali di default, quindi non è necessario dichiararli esplicitamente come virtuali. Tuttavia, è possibile utilizzare la parola chiave `final` per impedire che un metodo venga sovrascritto in una sottoclasse. ad esempio

```
class B {
    public void f() { System.out.println("B::f"); } // funzione virtuale
}
class D extends B {
    @Override
    public void f() { System.out.println("D::f"); } // overriding della funzione virtuale
}
public class Main {
    public static void main(String[] args) {
        B b = new B();
        B d = new D();
        b.f(); // Output: B::f
        d.f(); // Output: D::f (polimorfismo in azione)
    }
}
```

C++ in C++ le funzioni non sono virtuali di default, quindi è necessario dichiararle esplicitamente come virtuali utilizzando la parola chiave `virtual`. Ad esempio

```
class B {
    public:
        virtual void f() { cout << "B::f" << endl; } // funzione virtuale
};
class D : public B {
    public:
        void f() override { cout << "D::f" << endl; } // overriding della funzione virtuale
};
int main() {
    B* b = new B();
    B* d = new D();
    b->f(); // Output: B::f
    d->f(); // Output: D::f (polimorfismo in azione)
    delete b;
    delete d;
    return 0;
}
```

2.2.2 Early binding

Definition 2.2.3: Early binding

l'early binding è il processo di associazione di una chiamata a funzione con la sua definizione durante la fase di compilazione del programma

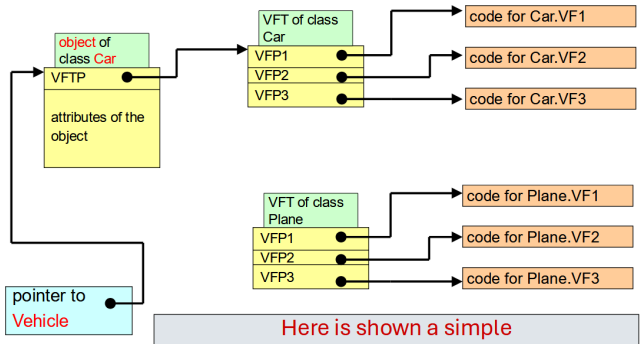
It's the method used by most non OO languages, and it's also the default behaviour for functions and methods in C++ (if not declared as virtual).

2.2.3 Late binding

Definition 2.2.4: Late binding

The correct address of the code to execute after a funciton call is deduced only at runtime.

Late binding can be implemented by simply using a VTABLE, or a table of virtual function pointers relative to each class:



Each object has a hidden pointer to the virtual funciton table (VTABLE) of it's class.
In short, the following steps are performed for every virtual funcion call:

1. The program looks for the VFTP in the object
2. It then searches the table for the correct VFP using the funciton name
3. If the pointer is found, the code pointed to is executed

The reason why not all function calls are virtual in most languages is because of eficiency, although in most cases the overhead generated by virtualization is negligible.

2.3 Examples

```
#include <iostream>
#include <string>
void prn( std::string s) {
    std::cout << s << "\n";
}
class A {
  public:
    virtual void f() { prn("A::f()"); }
    void g() { prn("A::g()"); }
    virtual void h() { prn("A::h()"); }
    void i() { prn("A::i()"); }
};
class B : public A {
```

```
  public:
    virtual void f() { prn("B::f()"); }
    virtual void g() { prn("B::g()"); }
    void h() { prn("B::h()"); }
    void i() { prn("B::i()"); }
};
class C : public B{
  public:
    virtual void f() { prn("C::f()"); }
    void g() { prn("C::g()"); }
    void h() { prn("C::h()"); }
    void i() { prn("C::i()"); }
};
class D : public A {
  public:
    virtual void h() { prn("D::h()"); }
    void m() {
        prn("D::m()");
        h();
    }
    virtual void g() { prn("D::g()"); }
};
class E : public D {
  public:
    virtual void f() { prn("E::f()"); }
    virtual void h() { prn("E::h()"); }
    void i() {
        prn("E::i()");
        m();
    }
    void g() { prn("E::g()"); }
};
class G : public E {
  public:
    virtual void f(int i) { prn("G::f(int)"); }
};

int main() {
    A anA;
    A *a;
    B aB;
    B *b;
    a = new B();
    a->f();
    a->g();
    b = (B*)a;
    b->f();
    b->g();
    A &arA = *new E();
    arA.i();
    E anE;
    anE.i();
    arA.g();
    D *apD = &anE;
    apD->g();
    A *apA = &anE;
```

```
    apA->g();
    G aG;
    aG.f();
    aG.f(3);
    aG.h();
    aG.i();
    aG.g();
    a = &aG;
    a->f();
    a->f(3);
    a->h();
    a->i();
    a->g();
    G *pG;
    pG = (G*)a;
    pG->f();
    pG->f(3);
    pG->h();
    pG->i();
    pG->g();
}
```

Given the C++ classes defined above, here are some interesting function calls to look at:

- a->f(): at runtime, a is a pointer to an object of class B. Seen as f is declared virtual in A it is overridden by the B class and "B::f()" is output.
- a->g(): because g is not declared as virtual in A, early binding is used and the function declared in A is used and "A::g()" is output.
- aG->f(): a function with signature f() is decalred in class G's superclass (E), but G also defines a function with name f but with a different signature (f(int)). Thus the homonymous function in the superclass is hidden and an error occurs.
- a = &aG; a->f(): f() is virtual in A, so late binding is used. f(int) is defined in G but its signatura doesn't match, so E's definition for f() is used instead.
- a = &aG; a->f(int): because A doesn't have any definition for a function with signature f(int), an error is thrown.

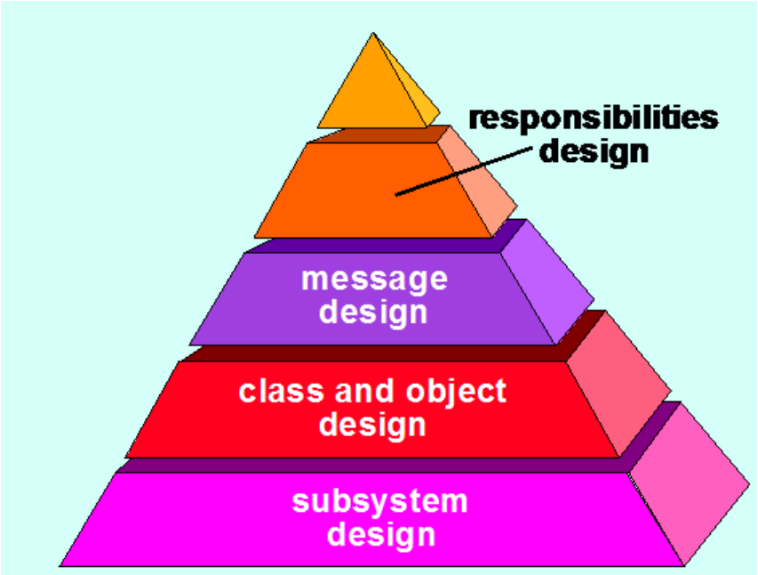
2.4 templates

2.4.1

2.5 Object Oriented Design in UML

Definition 2.5.1: Object Oriented Design

Object Oriented Design is the phase of software development that follow Object-Oriented Analysis (OOA), where the abstract models of the system are refined and elaborated into detailed specifications that can be directly implemented in an object-oriented programming language



In other words the OOD transform the "what" of OOA into the "how", adding technical details necessary for construction while preserving system correctness and maintainability

2.5.1 Design Issues

- **Decomposability**: The facility with which a design method helps the designer decompose a large problem into sub-problems that are easier to solve
- **Composability**: the degree to which a design method ensures that program components (modules), once designed and built, can be reused to create other systems
- **understandability**: the ease with which a program component can be understood without reference to other information or other modules
- **Continuity**: the ability to make small changes in a program and have these changes manifest themselves with corresponding changes in just one or a very few modules
- **protection**: a architectural characteristic that will reduce the propagation of side affects if an error does occur in a given module

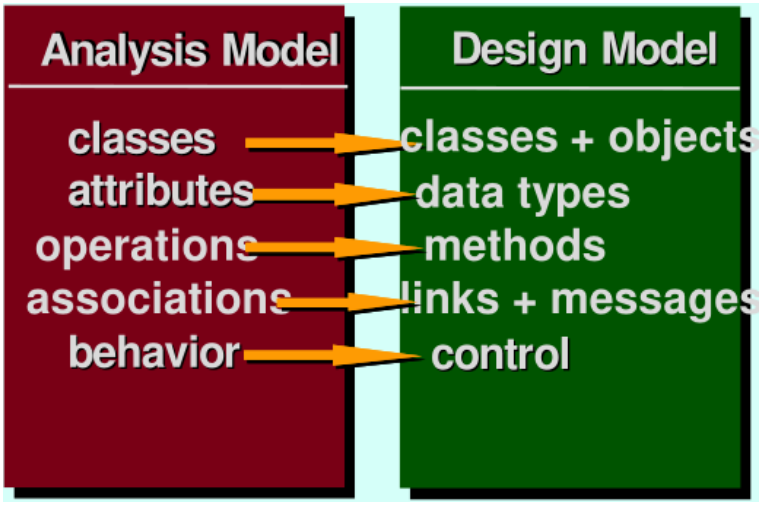
2.5.2 The four pilars for OOD

These are four components from the complete architecture of any well-designed OO sysyem:

- **Problem domain component**:the subsystems that are responsible for implementing customer requirements directly
- **Human interaction component**:the subsystems that implement the user interface (this included reusable GUI subsystems)
- **Task Management Component**: the subsystems that are responsible for controlling and coordinating concurrent tasks that may be packaged within a subsystem or among different subsystems;
- **Data management component**: the subsystem that is responsible for the storage and retrieval of objects.

2.5.3 OOA vs OOD

Core concept: OOD refines the abstract OOA model into a concrete, implementable design



what distinguishes OOD from OOA is:

- LEvel of detail:
  - Names are fixed
  - Fixed signatures for messages
  - Multiplicity & its realization
  - Visibility
  - Algorithms for methods
  - More detailed sequence/collaboration diagrams
- Additional diagram notations

  In OOD we have class diagrams, but they refined to match the design of the sysyem. In addition to class diagrams, we have several other diagrams:
  - **Structural Diagrams**
  - **Behavioral Diagrams**

2.5.4 Structural Diagrams for OOD in UML

In OOD there are two Structural diagrams:

- **Class Diagrams**: Their structure is the same as for OOA
- **Object Diagrams**:
  - They deal with objects, instances of classes
  - They are absolutely equivalent to class diagrams
  - Given this, we will not analyze them in deep

2.5.5 Behavioral Diagrams for OOD in UML

We'll study:

- **Sequence Diagrams**: Describe the interactions between objects by time ordering
- **Collaboration Diagrams**: Describe the interactions between the objects by organizations

Human interaction component

the subsystems that implement the user interface (this included reusable GUI subsystems);
In OOD I have class ddiagrams,but they are refined to match the design

2.5.6 Statechar diagrams vs interaction diagrams

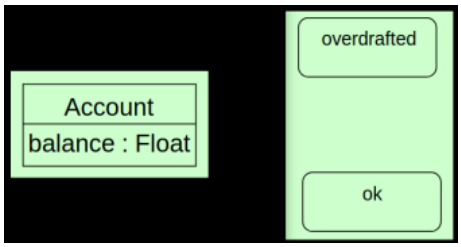
- **interaction diagrams**: show how obj interact

Statechart diagrams

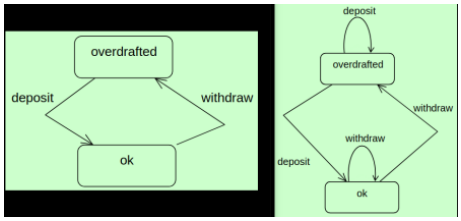
First of all we'll start with a def:

Definition 2.5.2: Object state

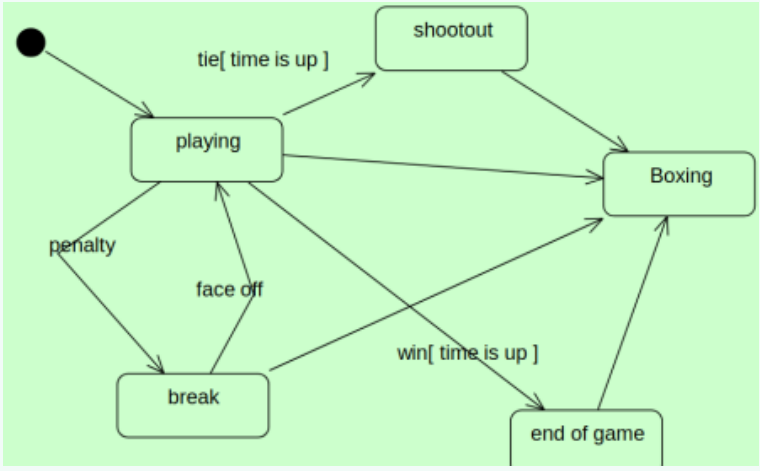
We define an *Object state* as the set of values that describe an object at a specific moment. The state is determined based on the attribute values.



A state can change because of an event (like the reception of a message) or can remain the same.

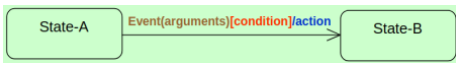


Example 2.5.1 (Hockey game)



State diagrams are represented through rectangles with rounded borders. Inside them are the state variables that uniquely describe the state, and they have associated *activities* or *actions*. An activity can be long and allows interruptions, while an action happens quickly.

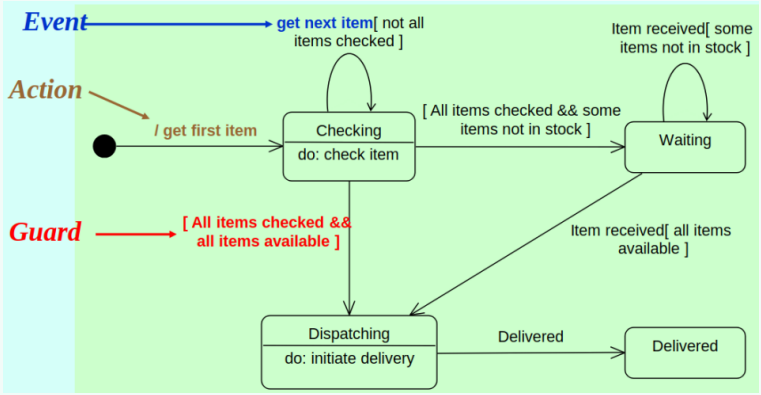
- **entry**: an action carried out at the beginning of the state
- an ongoing activity carried out while in the state (e.g., showing a window)
- an action carried out as a response to a specific event
- an action carried out at the end of the state



Based on an event, you can decide to move from one state to another. The state transition is described by an arrow.

- **event**: message to send
- **Guard condition**: the transition happens only when the guard is true. The guards for the exit transitions from a state are mutually exclusive.
- **Action**: a process that happens rapidly and cannot be interrupted.

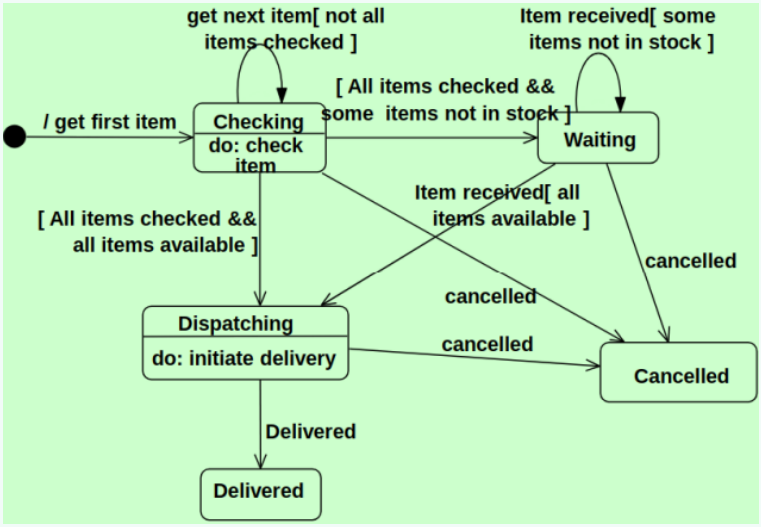
Example 2.5.2 (Order Management)



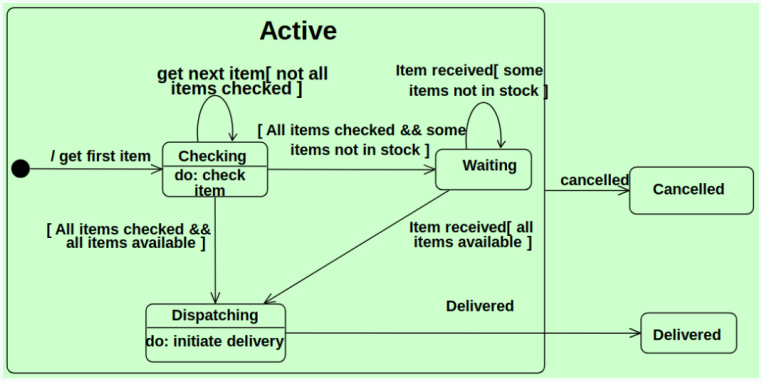
Now we wanna delete an order in any moment. you have two solutions:

- transitions from each state through a state "canceled"
- a super state and single transitions

Transition through "cancelled"



Super-state and sub-states



Statechart diagrams don't necessarily have to reference classes or objects, they can do that too refer to subsystems... In practice this is the most common way to use diagrams anyway statechart

2.5.7 Activity diagrams

Definition 2.5.3: Activity Diagrams

An *Activity Diagram* is a behavioral diagram in UML that describes the workflow of a system, showing the flow of control from activity to activity, including support for parallel processing and decision points

Purpose

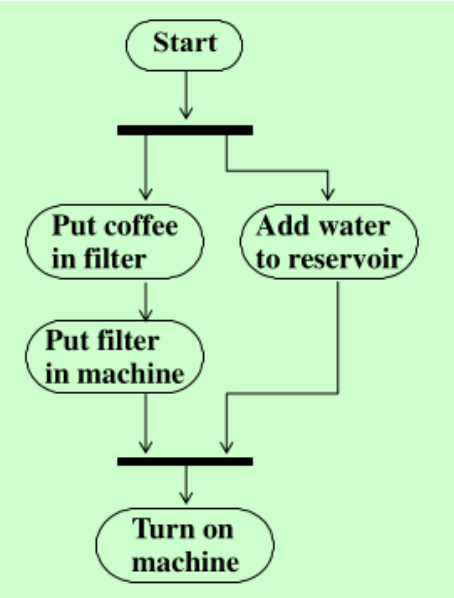
- Model business processes
- Describe algorithmic workflows
- Show concurrent/parallel activities
- Document complex procedural logic
- Visualize use case implementations

Key Characteristics

What activity represent? Two level of abstraction:

- **conceptual level**:
  - Task to be done
  - Business process step
  - High-level action
- **Specification/Implementation Level**:
  - Method on a class
  - Function execution
  - Code block

Example 2.5.3 (The coffie pot)



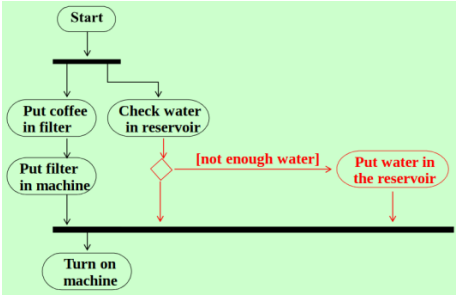
Sync bars

Activities can be carried out in parallel, in any order (fork). Arrived at a bar with multiple incoming arrows, all previous activities must be completed to continue (join). An activity diagram shows a partial order of activities.

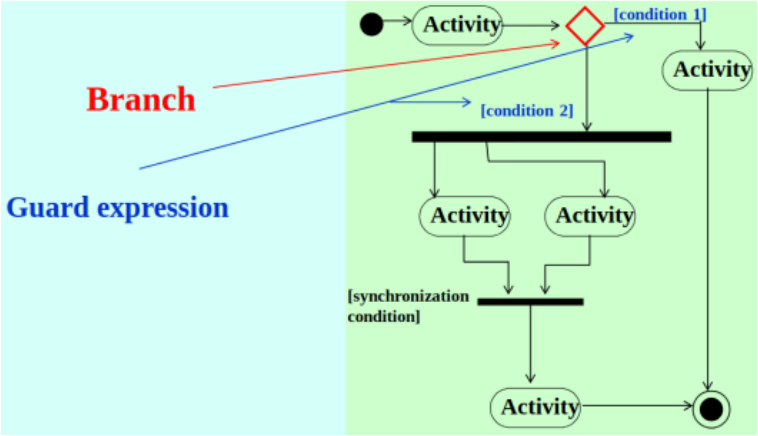
- **Fork (Parallel Split)** it is used for splitting into multiple concurrent flows that execute in parallel. Key points:
  - One incoming edge
  - Multiple outgoing edges
  - All outgoing flows start simultaneously
  - Activities execute concurrently (any order)
  - Synchronization bar (thick line)
- **Join (Parallel Merge)**, it is used for waiting all incoming parallel flows to complete before continuing. Key points:
  - Multiple incoming edges
  - One outgoing edge
  - Synchronization point
  - Flow continues only when ALL incoming flows complete
  - Synchronization bar (thick line)

conditions

In activity diagrams we can also define "conditions": two alternative parts that you can't execute in parallel



Structure:



2.5.8 Interaction Diagrams

Definition 2.5.4: interaction Diagrams

Interaction diagrams are behavioral diagrams in UML that model the dynamic aspects of a system by showing how objects collaborate and communicate through message exchanges to accomplish specific tasks or scenarios. They emphasize the flow of control and data between objects during runtime execution

Key charateristic

- Focus on "real" entities (Obj, not classes)
- Show msg e
- Can represent both sync and async Communication
- Bridge the gap between requirements and implmentation

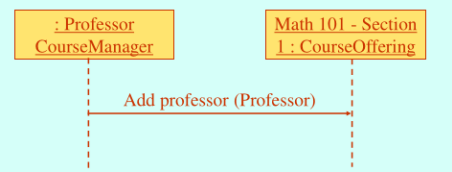
Interaction diagrams provide two different perspectives on the same interaction:

sequence diagrams

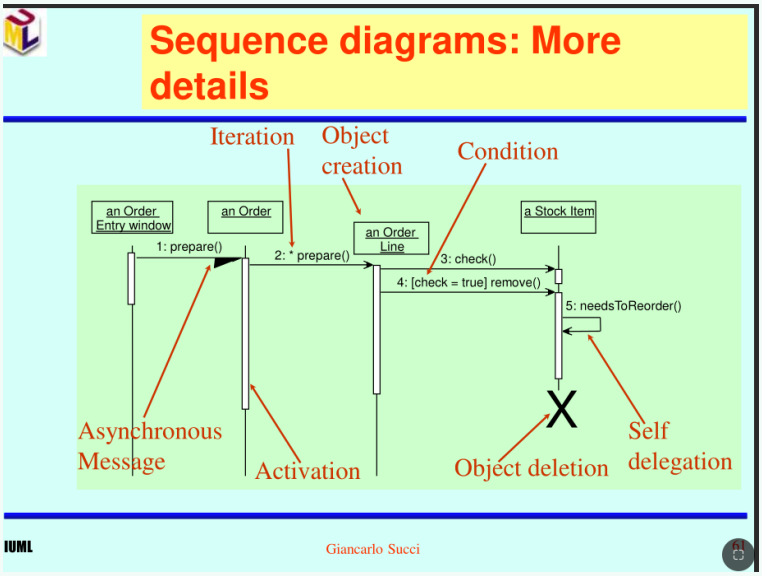
- emphasize *when* messagge are sent
- Shows object interactions arranged in time sequence
- Shows temporal ordering clearly
- Best for understanding flow over time

- Vertical timeline representation

Here we can see timelines:



Example 2.5.4 (detailed example)



Content of sequence diagrams

We have this content:

objects (participants) Lifelines:

- Vertical line extending from the obj
- represent the obj existence time
- contines 'till obj is destroyed

Messages There are two types of msgs:

- **Sysnc msgs**:
  - Full arrowhead: $\longrightarrow$
  - Caller wait for completion
  - blocking operation
  - most common in obj-oriented system
- **Async msgs (signals)**:
  - Half arrowhead (stick arrow): $\rightarrow$
  - Sender doesn't wait for completion

- Non-blocking operation
- Used for concurrent operations

they can do three functions:

- making a new thread
- making a new obj
- communicate with a thread already in execution

- **Return Messages**:
  - Typically shown as a dashed arrow: ⇢