شی گرایی
(استاد رویا یاقری)

Subject جلسه 1 (۹۸/۴/۱۷)
Date

فرآیند توسعه نرم افزار

" " " شامل مراحلی ست که برای توسعه یک نرم افزار باید پیموده شود.

برای این کار می توان از مدل های فرآیند توسعه استفاده کرد. این مدل ها می تواند شامل:

۱- مدل آبشاری یا خطی: در این مدل مراحل زیر به صورت ترتیبی اجرا می شود

جمع آوری نیازمندی ها ← تحلیل ← طراحی ← پیاده سازی ← آزمون

ابتدا همه درخواست ها از مشتری دریافت می شود سپس این درخواست ها تحلیل و مستند سازی می شود.

تیم طراحی بر اساس اطلاعات تحلیل، طراحی را انجام می دهد و سپس تیم پیاده سازی با توجه به نتایج طراحی کدنویسی می کند. از آن جایی که این مدل مدت طولانی از زمان تحلیل درخواست مشتری طول می کشد تا نتیجه به کاربر تحویل داده شود در اغلب موارد نتیجه مورد رضایت کاربر نهایی نیست.

۲- تکرار و توسعه: در این مدل یک پروژه بزرگ به چندین زیر پروژه تقسیم می شود و هریک از زیر پروژه ها به طور مستقل، فازهای فرآیند توسعه را طی می کنند و در نهایت نتیجه هریک از زیر پروژه ها با هم ادغام شده و خروجی نهایی به دست می آید

نتیجه 1 ← آزمون ← پیاده سازی ← طراحی ← تحلیل ← جمع آوری پروژه 1

نتیجه 2 " " " " " " پروژه 2

Subject
Date

در این روش چنانچه نیاز به تغییر در یک سمت باشد فقط زیر پروژه ای که آن سمت را شامل می شود ، دچار تغییر و تکرار می شود

۳- افزایشی (تکاملی) : در این مدل ابتدا با درخواست ها (نیازهای اولیه مشتری) یک مدل اولیه پیاده سازی می شود ، سپس با افزایش نیازمندی ها مدل اولیه تکامل می یابد

از آن جایی که در طراحی سی کرا نیازی نیست که در ابتدا فرآیند طراحی کاملی داشته باشیم یعنی دائما میتوانیم در چرخه حیات نرم افزار (پروژه) طراحی را تغییر دهیم پس به دنبال مدل فرآیند افزایشی-تکاملی می رویم

متدولوژی توسعه نرم افزار :

متدولوژی توسعه یک سری مدارد و چارچوب کاری است که همراه فرآیند توسعه وجود دارد

متدولوژی ها در حالت کلی به ۳ دسته تقسیم می شوند : ۱ سیاخت یافته ۲ سی کرا و ۳ فرآیند کرا

برای رسیدن به یک طراحی سی کرا ما از متدولوژی های سی کرا استفاده می کنیم . در متدولوژی سی کرا می بایست تحلیل سی کرا و طراحی سی کرا داشته باشیم

۱. تحلیل سی کرا : برای رسیدن به تحلیل سی کرا ، می بایست دو گام زیر را طی کنیم :

۱- گوش دادن به صحبت های مشتری و نوشتن آن

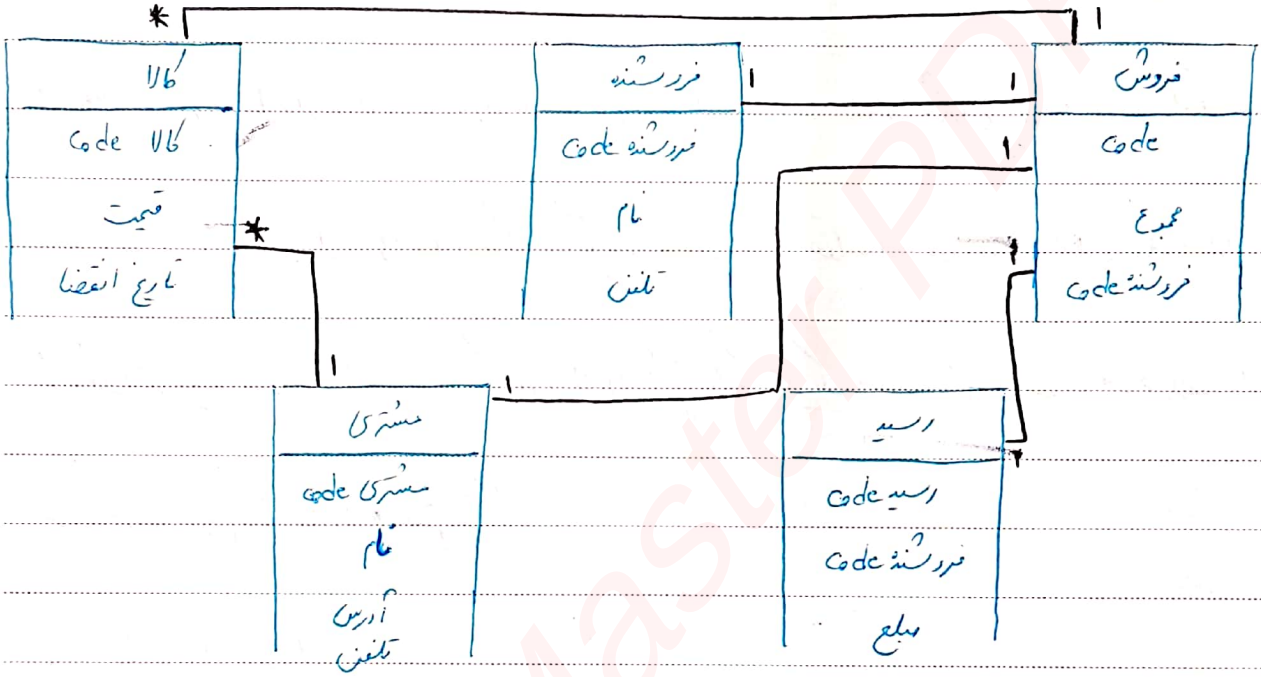
۲- ایجاد مدل دامنه از روی صحبت های مشتری ، [redacted]

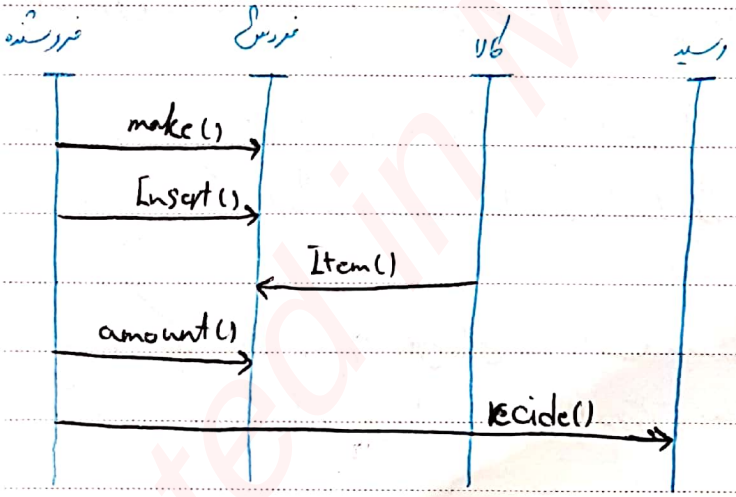
برای تعیین مدل دامنه کافی ست زیر اسامی خط بکشیم .

Subject
Date

۲. دیاگرام توالی: برای رسیدن به دیاگرام توالی کافی است ارتباطات بین اساسی در فاز تحلیل را پیدا کنیم (توصیف کنیم)

مثال: یک مشتری با سبد خریدش به پیش فروشنده می آید و فروشنده یک فروش جدید را شروع می کند. فروشنده تمام خریدهای مشتری را وارد می کند، سیستم مجموع سفارشات را محاسبه می کند، مشتری پرداخت می کند و سیستم رسید مشتری را پرینت می کند و سیستم موجودی را به روز می کند.





Subject
Date

انتخاب واحد یک عمل است که در سیستم انجام می‌شود پس متن نیست

زبان مدل‌سازی یکنواخت یا UML :

UML یک زبان مدل سازی است که برای تحلیل و طراحی سیستم‌های مبتنی بر شی به کار می‌رود. از UML برای مشخص کردن، تصویرسازی، ساخت و مستندسازی استفاده می‌شود.

از UML می‌توان قبل از ساخت یا بازساخت نرم افزار استفاده کرد. UML شامل تعدادی عنصر گرافیکی است که از ترکیب آن‌ها نمودارهای UML شکل می‌گیرد. هدف از استفاده از نمودارهای مختلف UML ارائه دیدگاه‌های گوناگون از سیستم است. باید توجه کنید که مدل UML آنچه را که باید یک سیستم انجام دهد توضیح می‌دهد ولی چیزی درباره نحوه پیاده سازی سیستم نمی‌گوید.

ویژگی های زبان UML :

۱- مدل UML آنچه که یک سیستم باید انجام دهد را توضیح می‌دهد و با نحوه پیاده سازی کاری ندارد

۲- مدل UML باعث تسهیل ارتباط میان اعضا پروژه یا بین تولیدکنندگان مختلف می‌شود که این کار را از طریق تبادل گزارش‌های هفتگی و مداوم بین اعضا پروژه انجام می‌دهد.

۳- به دلیل اینکه UML ترکیبی از زبان های مختلف است به منظور حفظ سازگاری و جمع‌آوری خصوصیات ثبت آن‌ها با گیر کمی پیچیدگی دارد. (مدل های آن را می‌توان مستقیما به انواع زبان‌های مختلف ارتباط داد)

۴- استفاده از UML تا حد زیادی هزینه‌های آموزش نظیر آموزش و استفاده مجدد از ابزارها را در هنگام ایجاد تغییر در سازمان و طرح‌ها کاهش می‌دهد.

Subject
Date
//

۵_ مهندسی رو به جلو و مهندسی معکوس از مهم ترین قابلیت های UML است . منظور از مهندسی رو به جلو نگاشت مدل های UML به یک زبان های برنامه نویسی است . و منظور از مهندسی معکوس این است که شما بتوانید از یک برنامه زبان شی گرا مدل های UML معادل آن را بدست آورید.

۶_ مدل UML مستقل از متدولوژی یا فرایند نرم افزار است به این معنا که برای استفاده از UML نیاز به استفاده از یک متدولوژی خاص وجود ندارد.

مفاهیم پایه در رشنال رز (Rational Rose) 8

از نرم افزار رشنال رز برای پیاده سازی مدل های UML استفاده می کنیم

مفهوم Use case 8

هر سرویسی که سیستم در اختیار کاربران قرار می دهد یک Use case محسوب می شود که به صورت بیضی نمایش داده می شود

به عنوان مثال در ATM برداشت وجه یا گرفتن موجودی Use case محسوب می شود

مفهوم Actor :

هر کس که با Use case کار می کند یک Actor است . در واقع Actor کسی است که Use case در جهت سرویس دادن به آن عمل می کند.

شناسایی Actor ها اولین قدم برای رسم Use case دیاگرام است.

از دیدگاه کلی Actor ها به دو دسته تقسیم می شود : ۱_ Actor های اصلی که معمولا انسانی هستند که از سیستم اطلاعات می گیرند یا به آن اطلاعات تزریق می کنند ۲_ Actor های فرعی ، Actor هایی هستند که مستقیما با سیستم در ارتباط نیستند ولی وجودشان برای فعال نگه داشتن سیستم ضروری است

از دیدگاه دیگر می توان گفت Actor ها به ۳ دسته تقسیم می شوند :

Subject
Date

۱- کاربران سیستم ۲- سیستم های دیگر ۳- زمان : از زمان می توان به عنوان Actor استفاده کرد، در حالت هایی که در یک زمان مشخص، قرار است عمل مشخص صورت گیرد.

مفهوم سناریو :
سناریو در واقع متنی است که فعالیت های use case را به طور کامل توضیح می دهد.

مفهوم Stereo Type :
از stereotype برای طبقه بندی انواع مدل ها استفاده می شود. که دارای انواع زیر است

۱- Actor : کسی که با سیستم در ارتباط است

۲- Boundry : به معنای user interface (واسط کاربری) است

۳- Controly ها اشیاء کنترلی هستند یعنی هرکجا در تحلیل قصد نمایش اشیاء کنترلی را داشته باشیم استفاده می کنیم

۴- Entity اشیائی هستند که در سیستم وجود دارند مثل سن بلیت در سیستم صدور بلیت

۵- Table اگر از جدولی از پایگاه داده استفاده شود برای نمایش آن از این نوع استفاده می کنیم.

PqPCO

Subject
Date

انواع View در نرم افزار رستوران زن:

۱. use case view: به تشریح رفتار سیستم از دیدگاه کاربر می پردازد که در آن Actor ها و use case ها و رفتار متقابل آن ها نمایش داده می شود

۲. logical view: آنچه را که باید پیاده سازی شود در این دید قرار می گیرد. عناصر اصلی این دید، کلاس ها و صفت ها و متدهای آن ها است و نیز ارتباط میان کلاس ها نمایش داده می شود.

۳. Component view: در این دید کلاس های ایجاد شده در مرحله قبل به Component تبدیل می شود و نیز ارتباط میان Component ها مشخص می شود. Component در واقع یک زیر سیستم است که شامل مجموعه ای از کلاس هایی هست که از لحاظ منطقی با هم سازگاری داشته و در کنار هم قرار می گیرند

۴. Deployment view: در این دید نگرشی از فعالیت های موجود با سخت افزار سیستم نمایش داده می شود.

Subject جلسه ۲ (۹۸/۰۴/۲۴)
Date

رسم نمودار Use Case :

ما در نمودار Use Case به دنبال نیازمندی های کاربران هستیم . پس در گام اول باید به استخراج نیازمندی ها بپردازیم .

نیازمندی ها به ۲ دسته : ۱- کاربردی ۲- غیرکاربردی تقسیم می شوند . و در اینجا منظور ما نیازمندی های کاربردی است .

برای استخراج نیازمندی ها باید محدوده سیستم مشخص شود یعنی مشخص شود چه Actor هایی با سیستم در ارتباط هستند .

بعد از تعیین محدوده باید مشخص شود که هر کاربر (Actor) از سیستم چه می خواهد .

در گام آخر روابط بین Use case ها و Actor ها را مشخص می کنیم .

توجه داشته باشید که نمودار مورد کاربری ترتیب اجرای موارد کاربری را مشخص نمی کند .

۱- نیازمندی ها
- کاربردی
  - مبتنی بر هدف
  - مبتنی بر فعالیت
  - مبتنی بر توانایی
  - مبتنی بر سرویس
- غیر کاربردی

۲- محدوده سیستم

۳- تعیین روابط

Subject
Date

انواع روابط در نمودار Use case ؟

۱- رابطه انجمنی : هرگاه دو الحان جهت سازی با یک دیگر یک رابطه طولانی مدت و ماندگار داشته باشند به طوری که رابطه آن ها فقط محدود به زمان مبادله یک سرویس خاص بین آنها نباشد، این رابطه را انجمنی می گویند. در نمودار use case ، رابطه بین Actor ها و use case ها از نوع انجمنی است که آن را با خط ممتد نشان می دهند. که جهت این خط جهت ارتباط را نشان می دهد.

۲- رابطه وابستگی: هرگاه یک الحان جهت سازی برای انجام یک سرویس خاص با الحان دیگری ارتباط برقرار کند. و پس از برطرف شدن نیاز آن ارتباط قطع گردد، رابطه بین این دو الحان از نوع رابطه وابستگی است. در نمودار use case رابطه بین دو use case از نوع وابستگی است که آن را با خط چین جهت دار نشان می دهیم (- - - - -)

رابطه وابستگی خود به ۲ دسته تقسیم می شود.

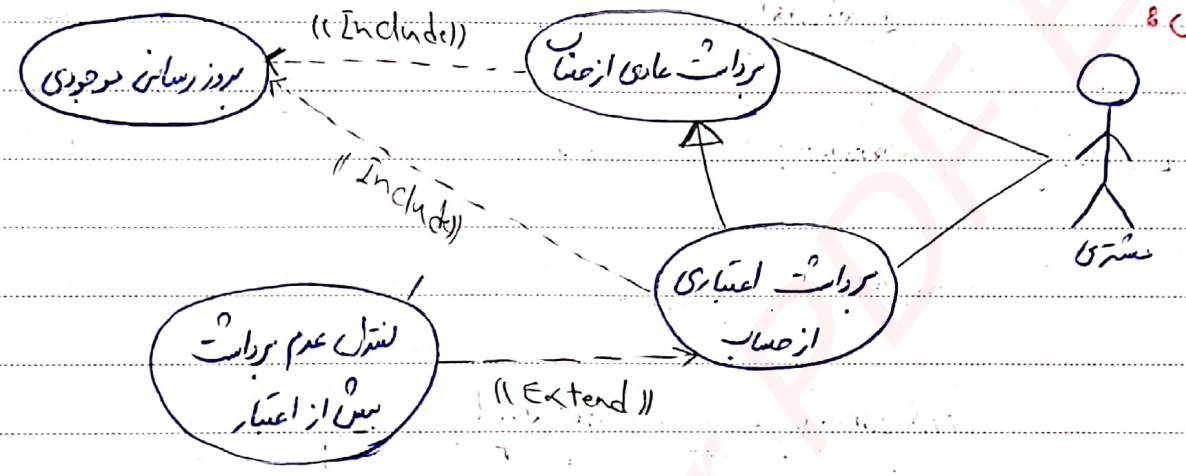
۱) رابطه Include : use case ها در حالت کلی به ۲ دسته اصلی و فرعی تقسیم می شوند. به اصلی همیشه اجرا می شود اما use case فرعی تحت شرایط خاص اجرا می شود. اگر چند use case اصلی دارای قسمت مشترکی باشند که به اندازه کافی بزرگ بوده به طوری که بتوان آن را یک مورد کاربری مجزا در نظر گرفت. آن گاه این قسمت مذکور را از درون تمامی use case های اصلی بیرون کشیده. و به عنوان یک use case مجزا تعریف می کنیم رابطه بین این use case جدید با use case های اصلی از نوع Include است ((Include))

۲) Extend : اگر در داخل چند use case اصلی (مورد کاربری اصلی) عملیات مشترکی باشد که در شرایط خاصی اجرا شود آن گاه این عملیات را از داخل use case های اصلی بیرون کشیده و به عنوان یک use case مجزا تعریف می کنیم. رابطه بین این use case جدید و use case های اصلی از نوع Extend است ((Extend))

Subject
Date

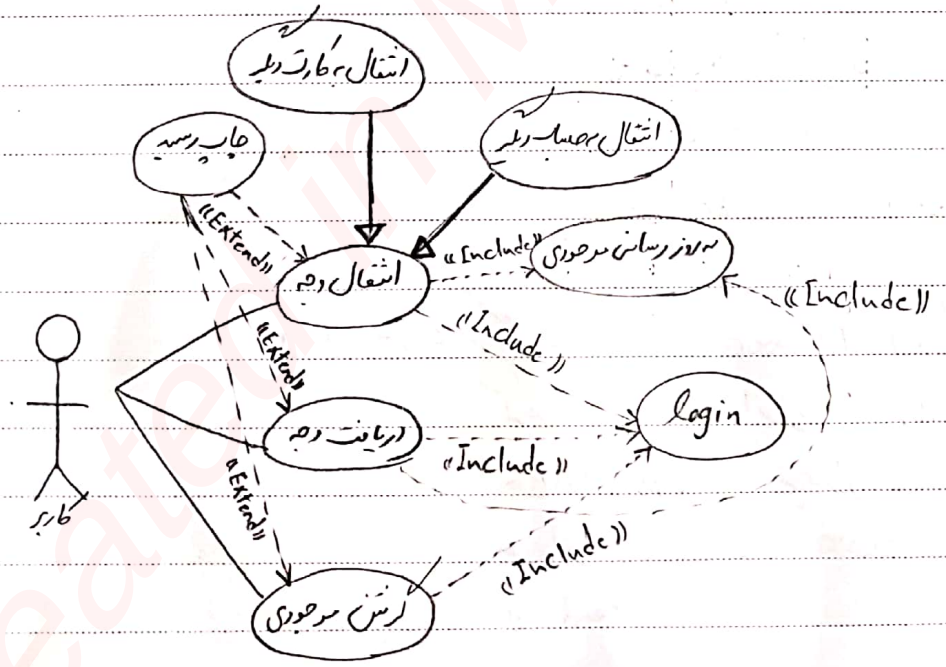
۳. رابطه تعمیم (رابطه وراثت): این رابطه بین یک use case با use case یا یک Actor با Actor کشیده می‌شود که بیان‌گر مفاهیم وراثت است (←)

مثال:



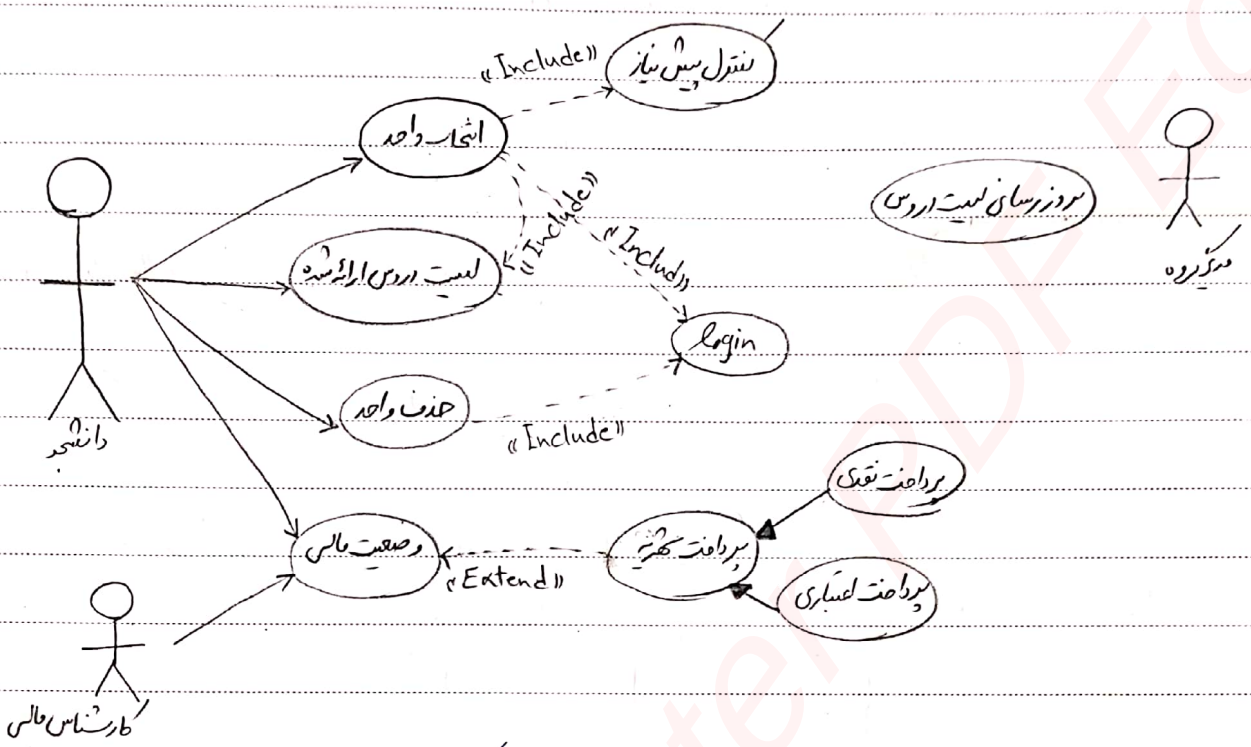
مثال: سیستم ATM را در نظر بگیرید نمودار use case آن را رسم کنید.

ابتدا کاربر کارت خود را وارد کرده، سپس عملیات login را انجام می‌دهد، در صورت صحیح بودن رمز امکان انتخاب عملیات های برداشت وجه، انتقال وجه و یا گرفتن موجودی را فراهم می‌کند. در نهایت کاربر با گرفتن کارت خود از سیستم خارج می‌شود

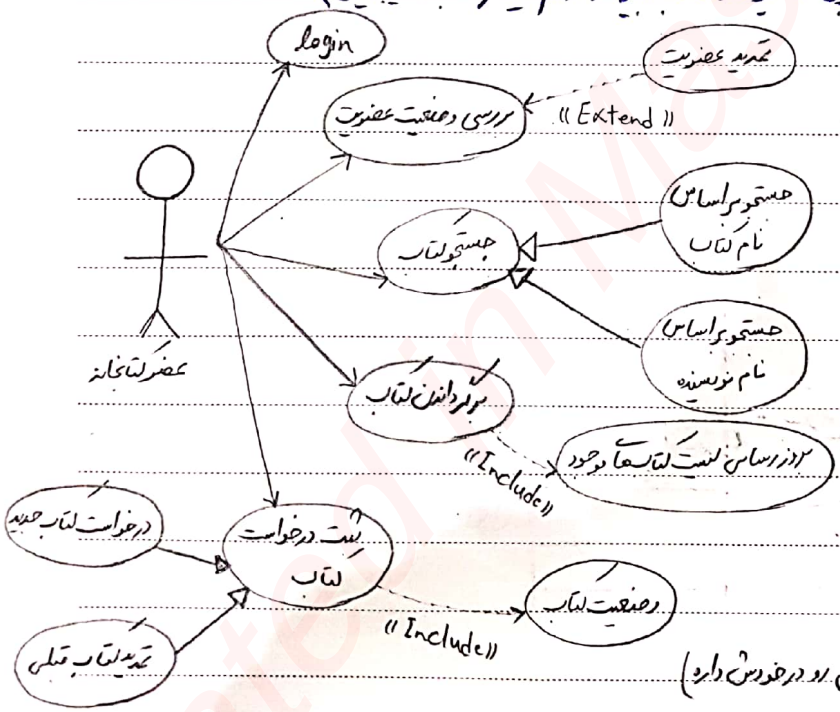


Subject
Date

مثال: نمودار مورد کاری سیستم ثبت نام دانشگاه را بنویسید



مثال: نمودار use case یک کتابخانه و کاری که بخواهد کتاب بگیرد را رسم کنید (کتابخانه دیجیتال)



بعضی فلش‌ها:

Include ← حتماً از کجا اولی به دومی میریم (اولی به دومی رو در خودش داره)

Extend ← شاید برم (حتمایت)

Subject
Date

Sequence

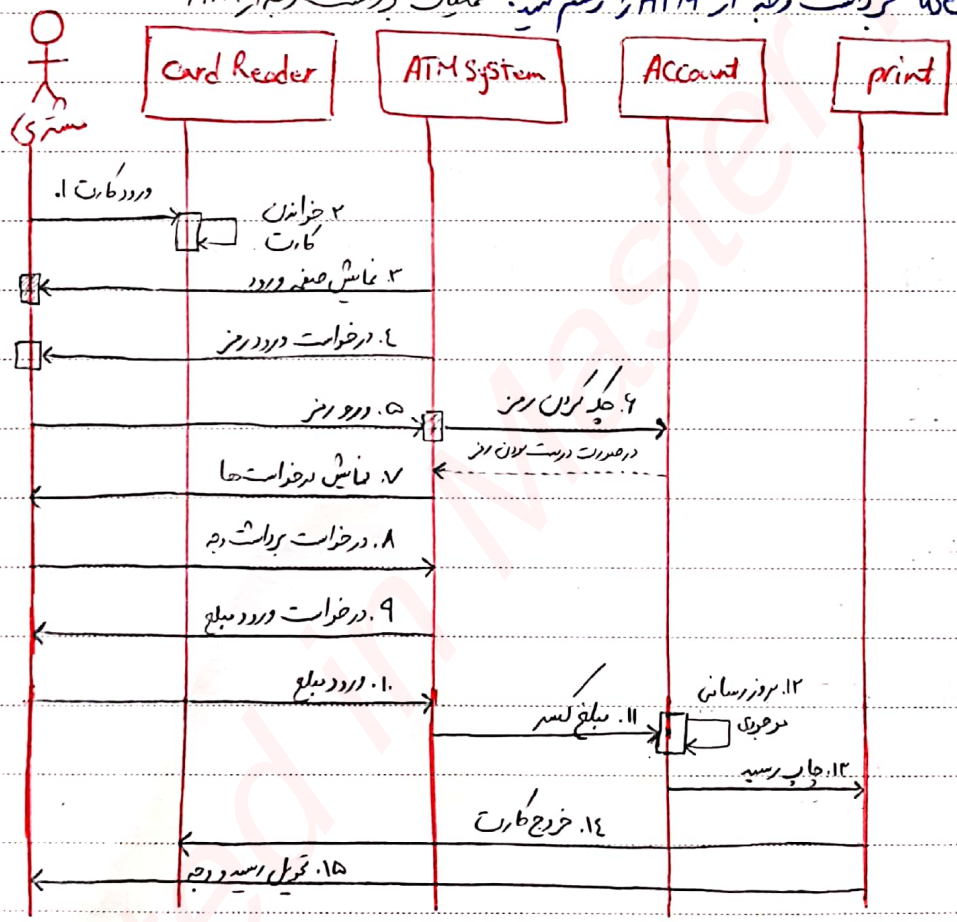
نمودار تعاملی (نمودار توالی)

نمودارهای تعاملی برای نشان دادن چگونگی عملیات در یک use case استفاده می‌شود. این نمودار زمانی مفید است که کسی بخواهد روند منطقی یک سناریو را بازبینی کند. نمودار توالی شامل موارد زیر است:

۱. object ۲. پیغام

با استفاده از یک پیغام، یک object می‌تواند از object دیگری عملیات خاصی را درخواست کند

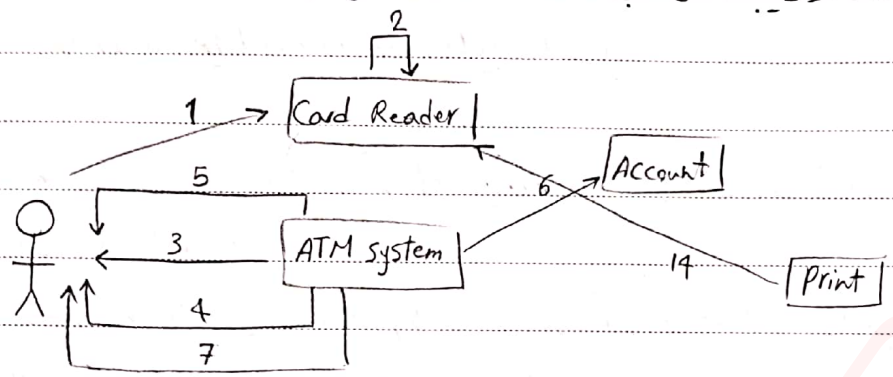
مثال: نمودار توالی use case برداشت وجه از ATM را رسم کنید. عملیات برداشت وجه از ATM



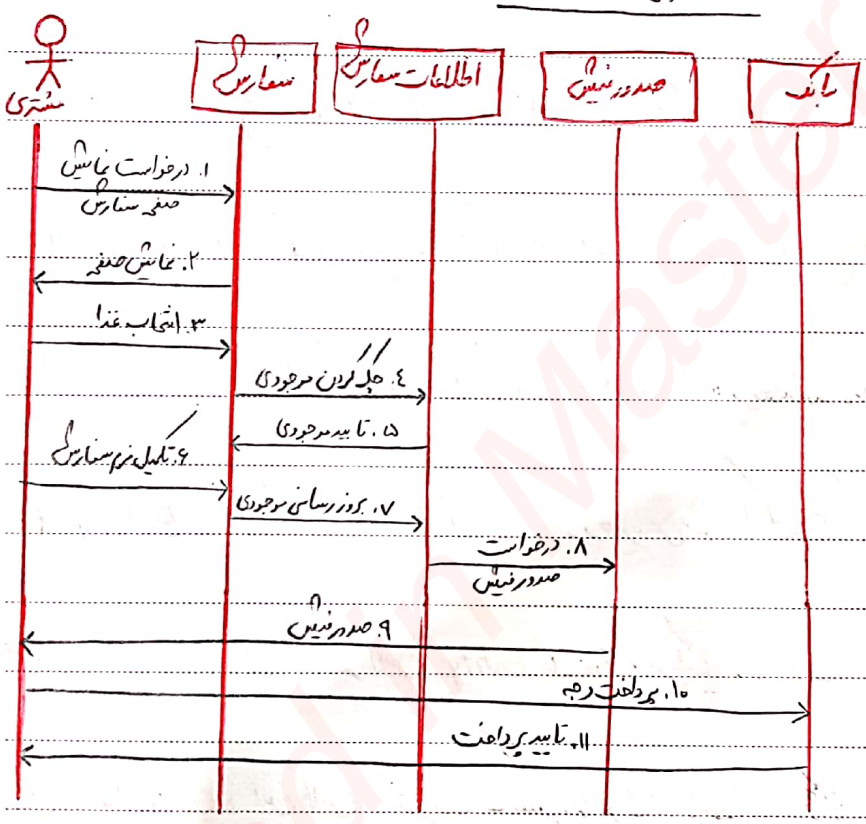
نمودار تعاملی (۲) (نمودار همکاری):

در نمودار همکاری نیز مانند نمودار توالی، object ها و پیغام ها وجود دارد با این تفاوت که در این نمودار تنها بر روی همکاری و تعاملات بین object ها تمرکز کرده و ترتیب زمانی اجرای تعاملات در نظر گرفته نمی‌شود

مثال: مثل بنل



مثال: نمودار سفارش غذا



Subject جلسه ۳ (۹۸/۴/۳۱)
Date

نمودار کلاس:

نام کلاس
ویژگی ها
عملیات

نمودار کلاس برای نمایش کلاس ها و ارتباط بین آنها استفاده می شود.

هر کلاس شامل نام کلاس، ویژگی ها و عملیات است.

کلاس مجموعه‌ای از اشیاء است که دارای ویژگی های مشترک هستند. یک مدل خوب برای یافتن کلاس ها جریان رویداد use case ها است.

وقتی براساس بالای نمودار sequence نگاه می کنیم یا یک Actor است و یا یک کلاس و یا یک شیء یک کلاس است.

پس با دسته بندی آنها می توانیم کلاس های موجود را پیدا کنیم، از طرف دیگر براساس فانکشن های وارد شده بر هر کلاس می توان متدهای آن کلاس را بدست آورد.

Stereo type کلاس ها (طبقه بندی کلاس ها):

۱) Boundry: کلاس هایی هستند که روی حاشیه بین سیستم ها و دنیا جهان قرار دارند. که شامل فرم ها، گزارش ها و واسط شما با سخت افزار مثل چاپگر هستند و با علامت ○— نشان داده می شوند.

۲) Entity: این کلاس اطلاعاتی که در حافظه ذخیره می شود یا نگهداری می کنند. (موجودیت ها)

○ هر کلاس Entity یک جدول در پایگاه داده دارد.

۳) Controly: این کلاس مسئولیت هماهنگی را بر عهده دارد. ◎

PAPCO

Subject :

Year. Month. Date. ( )

صفت های کلاس :

صفت محدوده متغیری است که می‌تواند شئ یک کلاس بگیرد. هر کلاس صفت‌ها و عملیات را در خود کپسوله می‌کند، بدین معنا که این صفات پنهان از دیگر کلاس‌ها قرار دارند.

به دلیل کپسوله بودن صفت های درون یک کلاس باید تعریف کنیم که چه کلاس هایی برای مشاهده و تغییر صفت ها اجازه دسترسی دارند.

۴ گزینه برای قابلیت رویت یک صفت وجود دارد:

① public : صفت برای تمامی کلاس های دیگر قابل رویت است. علامت (+)

② private : صفت برای کلاس های دیگر قابل رویت نیست. (-)

③ protected : فرزندان هر کلاس به صفت آن دسترسی دارند. (#)

حفاظت شده

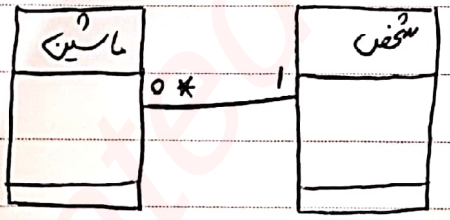
④ Pakage : صفت حالت عمومی برای کلاس های موجود در همان بسته را دارد. (~)

انواع روابط بین کلاس ها :

۱) رابطه انجمنی : رابطه معنایی بین کلاس هاست که به یک کلاس امکان می‌دهد تا درباره‌ی صفات و رفتار عمومی کلاس دیگر اطلاعات لازم را بداند.

برای هر رابطه انجمنی می‌توان درجه‌ی کاردینالتی مشخص کرد.

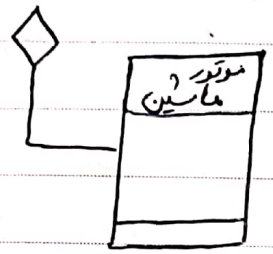
مثل رابطه ماشین با یک شخص



Subject :

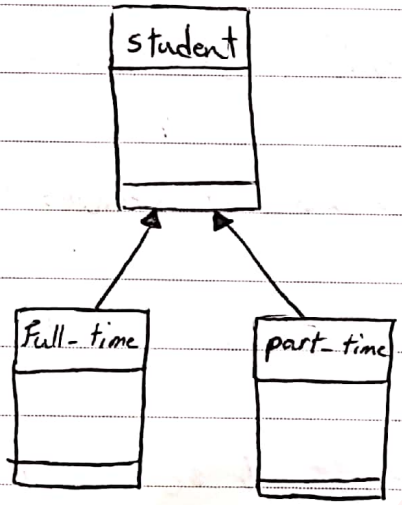
Year. Month. Date. ( )

2) رابطه تجمعی : در بعضی از موارد یک کلاس از تعدادی کلاس های خرد تشکیل می شود.
رابطه بین هریک از کلاس های خرد با کلاس های کل از رابطه تجمعی است.
مثل رابطه موتور با ماشین



موتور داخل ماشین

③ رابطه وراثت : یکی از ویژگی های شی گرایی سلسله مراتب و بسته بندی شی ها است. این رابطه ای سلسله مراتب را می توان بین دو کلاس پدر و فرزند دید.
که یک کلاس می تواند صفت ها و عملیات را از کلاس پدر به ارث برد.
مثل:



ASEM4N

Subject :

Year. Month. Date. ( )

نحوه اعلان کلاس :

```
class نام کلاس
{
    private ;
    public ;
    protected;
}
```

مثال 1 :

```
class test 1
  {
    private
        int a;
        };
    main()
    {
    test 1 ab;
    cout << ab.a;
    }
```

خروجی ندارد

غلط است

خروجی : ✗

* چون private تعریف شده نمی‌تونه از کلاس‌های دیگه استفاده کنه پس اشتباه است.

نقطه داخل همون تابع تعریف می‌شود

Subject :

Year. Month. Date. ( )

```
class test 2
{
Private:
    int x;
public;
int F() {
x=1
Cout << x;
}
}
```

=

```
class test 2
{
private:
int x;
public:
int F();
};
int test2 :: f(){
x=1;
Cout << x;
}
```

```
main (){
Test2 ab;
ab.F();
}
```

خروجی دستور دوم : 1

نکته 8 هنگام تعریف کلاس اعضای خصوصی (private) را مقدار دهی نکنید.

Subject :

Year. Month. Date. ( )

سازنده (Constructor)

متد هم نام با کلاس را سازنده می‌گویند، این متد وقتی اجرا می‌شود که شیئی از آن کلاس ایجاد شود.

مثال ۲: خروجی تابع زیر را بدست آورید.

```
class c {
private:
int x;
public:
c() {
cout << 'a';
 }
};
main() {
c ob;
}
```

خروجی: a

کانستراکتور هم‌نام کلاس است

مثال ۳:

```
class test3 {
private:
int x;
public:
test3 (int=3);
int f();
};
int test3 :: f() {
④ x++;
return x;
}
```

متغیر x مقدار عددی ۳ انتساب می‌دهیم

(متغیر کلاس توسط کانستراکتور مقدار دهی می‌شود)

کانستراکتور تعریف میشود

تابع

```
Constructor ② test3 :: test3 (int n) {
x = n;
}
① main() {
(شیئی می‌سازد) Test3 ob;
③ cout << ob.f();
}
```

Constructor اجرا می‌شود

تابع را صدا می‌زند

خروجی cout

در آخر به مرحله ④ بازمی‌گردد

خروجی: 4

(روند اجرای این سؤال مشخص شد به ①, ②, ③, ④)

* در امتحان روند اجرا هم مشخص شود کامل است

ASEMAN

Subject :

Year. Month. Date. ( )

مثال ۴:

```
class test 4 ( ) {
private :
 int x :
 public :
  Test 4 (int =0)
  void f( );
 };
Test 4 :: test 4 (int a)
 }
  x=a;
 }
  void test 4 ::f( )
  main ( )
  {
  test 4 ob (2)
  ob.f ( );
  }
```

خروجی: 2

تابع f تعریف شده و خروجی ندارد

int (a) = (int 2)

چون خروجی مقدار 2 را داده

پس به Constructor مقدار 2 داده

می شود یعنی int (2)

ASEM4N

Subject :
Year. Month. Date. ( )

مثال ۵:

خروجی: 6 و a

اولین خروجی 6 و بعد a خروجی دوم است.

```
class test 5 {
  public :
    test 5(){
        cout << 'a' ;
    }
  test 5 (int N) {
        cout << N;
    }
};
main(){
  Test 5 ob1(6);
  Test 5 ob2 ;
}
```

دوتا کانستراکتور داریم

در صورتی که ما هیچ ورودی نداشته باشد، خروجی کاراکتر a است

در صورتی که یک ورودی داشته باشد مقدار خروجی برابر با ورودی است

6a

خروجی 6 و a

دوتا Constructor دارد.

در اینجا پس شی تعریف کردیم یکی مقدار ۶ میگیرد یکی مقدار ندارد

مخرب (destructor):

متد همنام با کلاس که با (~) شروع می‌شود را مخرب می‌گویند که هنگام از بین رفتن شی اجرا می‌شود.

تابع مخرب دارای پارامترهای ورودی و خروجی نمی‌باشد، اما سازنده می‌تواند ورودی داشته باشد.

مثال ۶:

```
class test 6 {
  public :
  Test 6(){
        cout << 'a' ;
      }
  ~ test 6 (){
        cout << 'b' ;
};
```

اول Constructor میسازه a چاپ میشه

```
main(){
  ① Test6 ob;
  cout << 'c';
}
```

اول با این a چاپ می‌شود

بعدش cout می‌کند c را

و پس چون } بسته شده در مرحله بعد مخرب اجرا می‌شود

و b چاپ می‌شود

ASEMAN

Subject :
Year. Month. Date. ( )

مثال ۳۷

```
class test 7{
public:
test 7(){
cout << 'a';
}
~test 7(){
    cout << 'b';
    }
};
main(){
test 7 ob 7;
test 7 ob 2;
{
  test 7 ob 3;
}
cout << 'c';
}
```

کانستراکتور (تابع سازنده) فراخوانی می‌شود

از کلاس test 7 یک ob تعریف می‌کنیم → a

یک ob دیگر → a

کانستراکتور (تابع سازنده) فراخوانی می‌شود

→ a

مرگ ob 3 → b

دیستراکتور فراخوانی می‌شود

→ c

به ازای تعداد ob هایی که در پرانتز اصلی تعریف شده با بستن پرانتز تابع مخرب فراخوانی می‌شود

ob 1 مرگ → b

ob 2 → b

پس b, b چاپ می‌شود

ASEMAN

Subject :
Year. Month. Date. ( )

انتساب اشیاء به یکدیگر:

مثال ۸:

```
class test 8{
  public:
  int x;
  void f(){
  cout<<++x;
  }
};
```

مثل از متغیر x را چاپ کند اینکه یک واحد به آن اضافه می‌کند

```
main()
{
Test 8 ob1;
Test 8 ob2;
ob1.x = 1;
ob2 = ob1;
ob2.f();
}
```

دوتا داده تعریف می‌کنیم بدین صورت و آنها را مقدار دهی می‌کنیم که ob1 برابر با مقدار 1 است و ob2 برابر ob1 است و سپس برای شی ob2 تابع f را فراخوانی می‌کنیم

خروجی: 2

ارسال شی به تابع:

مثال ۹:

```
class test 9{
  private:
    int i;
    int j;
  public:
  Test 9 (int n){
    i=n;
  }
  int f(){
  j=i;
  return j;
  }
};
```

دوتا متغیر

سازنده ورودی n که برابر با i متغیر را می‌گیرد

تابع f را طوری تعریف می‌کنیم که مقدار j برابر با متغیر i است j را برمی‌گرداند

```
void k(Test9 ob)
{
  cout << ob. f();
}
main(){
Test 9 ob1(2);
k (ob1);
}
```

تابع k را تعریف می‌کنیم که ورودی آن یک شی از کلاس است توسط شی مربوطه کلاس تابع خروجی آن کلاس را چاپ می‌کند

در برنامه اصلی شی با مقدار اولیه 2 از کلاس تعریف می‌کنیم، سپس تابع k را با ورودی شی ob1 فراخوانی می‌کنیم

نکته: تابع سازنده هنگام ارسال شی به تابع اجرا نمی‌شود و تابع مخرب هنگام از بین رفتن شی در زمان پایان اجرای تابع اجرا می‌شود.

Subject :

Year: Month. Date. ( )

مثال ۱۰:

```
class test 10{
 private:
 int i;
 int j;
 public;
 test 10 (int n){ ⟶ int(n) = (int 2)
    i=n;
 }
 int f(){
    j=i;
    return j;
 }
 ~ test 10(){
       Cout<<'b';
       }
};
void k(test 10 ob){
   Cout<< ab.f();
 }
 main(){
Test 10 ob1(2); ⟶ Constructor اول باعث می‌شوند int(2) شود و 2 چاپ می‌شود
k(ob1); ⟶ 2 چاپ
                ⟶ b چاپ
Cout<<'a'; ⟶ خودش یعنی a چاپ می‌شود
}  ⟶ b چاپ می‌شود
```

خروجی: 2bab

j=2 ← (2)(b)ab ← مخرب

ASEMAN

Subject :

Year. Month. Date. ( )

تابع با خروجی از نوع شیء؟

مثال ۱۱:

```
class test 11 {
  public:
   int i;
};
test 11 k(){
    test 11 ob;
    ob.i = 2;
    return ob;
    }
```

```
main(){
test 11 ob 1;
    ob 1 = k();
cout << ob 1.i ;
}
```

خروجی: 2

یک کلاس هم می‌تواند به عنوان خروجی یک تابع باشد، هم به عنوان ورودی

مقداردهی متغیرهای مخصوص با تابع عمومی

مثال ۱۲:

```
class test 12 {
  private:
   int x, y ;
  public:
void setprice (int a , int b){
    x=a , y=b ;
    }
void Display(){
        Cout << "item 1" << x;
        Cout << "item 2" << y;
int main(){
 test 12 k;
 k. Setprice (2,5);
 k . Display();
}
```

* ۲ تا متغیر a و b می‌گیرد که a میرزه توی x و b توی y

خروجی‌ها: item1 = 2 ، item2 = 5

Subject :
Year. Month. Date. ( )

مدیریت دهی متغیرهای عمومی و خارج از کلاس

مثال ۱۳ :

خروجی : 7

```
class test 13 {
  public :
  int x, y ;
  int Area ( ){
    return x+y ; }
};
int main ( ){
Test 13 R1 ;
R1.x = 2 ;
R1.y = 5 ;
  Cout << "Area" = << R. Area ( )
}
```

تابع دوست (friend Function) :

تابع دوست در داخل یک کلاس تعریف می‌شود ولی جزء اعضای کلاس نیست ، یعنی شئ از کلاس نمی‌تواند این تابع دوست را فراخوانی کند ، بلکه این تابع دوست است که به اعضای خصوصی کلاس می‌تواند دسترسی داشته باشد

مثال ۱۴ :

```
class test 14 {
  private :
    inte ;
  public :
test 14 (int x){
    c = x ;
  }
friend int L ( test 14 ) ;
};
int L (test 14 ob){
  Cout << ob.c ;
}
main ( ){
  test 14 ob1 (3) ;
  L (ob1) ;
}
```

خروجی : 3

ASEMAN

Subject :

Year. Month. Date. ( )

مثال ۱۵:

```
class test 15 {
  private :
    int x;
public :
Test 15 (int) {
    x=i ;
}
friend f( test 15 z ){
    Cout<<z.x;
}
k(){
  Cout <<x;
}
};
```

```
main ( ){
test15 ob(2)
f(ob) ;
ob.k();
}
```

طبق تعریف تابع دوست داریم:
تابع friend جزئی از کلاس حساب می‌شود.

Test 15 ob(2); *

obf( );

اگر این را بدهد غلط است

کلاس دوست:

کلاسی که دوست کلاس دیگر است و می‌تواند به اعضای خصوصی آن دسترسی باشد.

مثال ۱۶:

```
class test 16{
    private :
      int t;
    public :
test 16 (int x){
    i =x;
}
 friend class d;
};
```

```
class d{
 public :
int f(test 16 ob){
 Cout <<ob.1 (3);
d ob 2;
 ob 2. f(ob1);
}
```

خروجی: 3

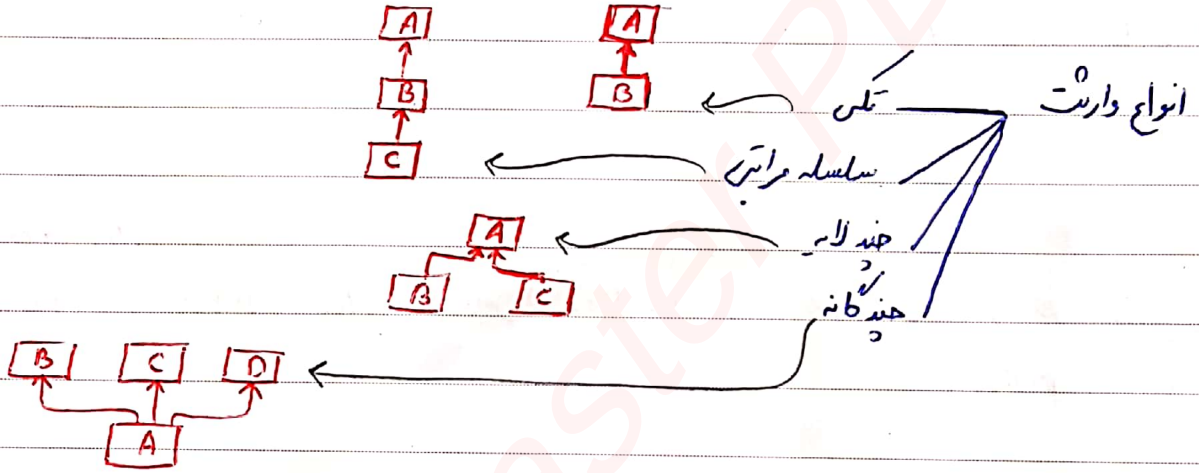
Subject : جلسه ٤ (٩٨/٥/٧)
Year. Month. Date. ( )

کپسوله سازی ← کلاس: صفات و عملیات را در داخل خود کپسوله میکند.

سلسله مراتب ← وراثت

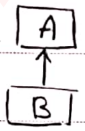
چاچولاریتی

تجرید




```
class A {
};
class B : نوع ارث بری A
{

}
```




```
classB : publicA
{
};
```

public
private
protected
← نوع ارث بری

Subject :
Year. Month. Date. ( )

نوع اول public :

تمام اعضای public کلاس A به عنوان اعضای public کلاس B محسوب می‌شود و تمام اعضای protected کلاس A به عنوان اعضای protected کلاس B محسوب می‌شود

```
class B : public A
{
};
```

نوع دوم private :

تمام اعضای public و protected کلاس A به عنوان اعضای private کلاس B محسوب می‌شود

```
class B, private A
{
};
```

نوع سوم protected :

تمام اعضای public و protected کلاس A به عنوان اعضای protected برای کلاس B محسوب می‌شود

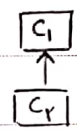
```
class B : protected A
{
};
```

مثال ۳:

```
class C1 {
    public :
        int f (int n) {
        return n;
        }
};
class C2 : public C1{
    public :
        int g(int n){{
            return n;
        }
}
```

```
main (){
    C2 ob;
    Cout << ob.f(1);
    Cout << ob.g(2);
}
```

جواب : خروجی ۱ و ۲



ASEMAN

Subject
Date

مثال ۲:

```
class a{
  public:
    a(){                    ← سازنده پدر
        cout << '1';
    }
   ~a(){                    ← مخرب پدر
        cout << '2';
    };

class b: public a{
  public:
    b(){                    ← سازنده فرزند
        cout << '3';
    }
   ~b(){                    ← مخرب فرزند
        cout << '4';
    }
};
main(){
  b ob;
}
```

(علامت ~ ← مخرب)

نکته: در فراخوانی سازنده‌ها ابتدا سازنده پدر فراخوانی می‌شود
سپس سازنده فرزند فراخوانی می‌شود

در فراخوانی مخرب‌ها:
ابتدا مخرب فرزند فراخوانی می‌شود
سپس مخرب پدر فراخوانی می‌شود

خروجی: 1342

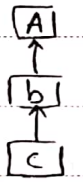
اول سازنده‌ها ←
① سازنده پدر a ← ①
③ سازنده فرزند b ← ②

دوم مخرب‌ها ←
④ مخرب فرزند ← ③
② مخرب پدر ← ④

Subject
Date

مثال ۳ :

```
class a{
   public :
     f(){
        cout << '1';
        }
   };
class b : public a{
          public :
           g(){
             cout << '2';
              }
   };
class c : public b{
        public :
          h(){
             cout << '3';
            }
};
main(){
        c ob1;
 1    ob1.f();
 2    ob1.g();
 3    ob1.h();
      b ob2;
 1    ob2.f();
 2    ob2.g();
```



نوع : ارث سری چندلایه

خروجی : 12312

دسترسی هر کلاس

class a : f()

class b : f(), g()

class c : f(). g(). h()

Subject
Date

مثال ۲:

```
class x {
    public:
        x() { cout << '1'; }
       ~x() { cout << '6'; }
};
class y: public x {
        public:
            y() { cout << '2'; }
           ~y() { cout << '5'; }
};
class z : public y {
        public:
            z() { cout << '3'; }
           ~z() { cout << '4'; }
```

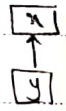
```
main() {
  z ob;
}
```

خروجی 123456

z به y و y به x دسترسی دارد

```
main() {
  y ob;
}
```

خروجی: 1256



y فقط به x دسترسی دارد

Subject
Date

مثال ۵ :

```
class C1{
    protected :
        int i ;
public :
C1 (int x){
        i = x ;
        Cout << i+1 ;
        }
};
class C2 : public C1{
        public:
        C2 (int x){
                Cout << i-1 ;
                }
};
main (){
        C2 ob(3);
}
```

* عرض protected همست
متغیر من i برای
شناخته شده است و
در Cout چاپ میشه
اگر private بود اصلا
نمیشناخت که بخواد چاپ بشه

خروجی : 42

Subject :

Year. Month. Date. ( )

مثال ۶ :

```
class x {
    public:
        x() { cout << 'a'; }
       ~x() { cout << 'b'; }
};
class y : public x {
    public:
        y() { cout << 'c'; }
       ~y() { cout << 'd'; }
};
main() {
    x ob1;
    y ob2;
}
```

خروجی : a a c d b b

ob1 برای سازنده x ← a

ob2 برای سازنده x و y ← a, c

ob2 برای مخرب x و y ← d, b

ob1 برای مخرب x ← b

} ← objectها اینجا ترم میشه

ob1 چون اینجا ترم میشه پس در آخر b رو مینویسیم

* زمانی که ob ها ترم میشن مخرب‌ها رو مینویسیم پس

ASEMAN

۱۸

Subject :
Year. Month. Date. ( )

مثال ۷:

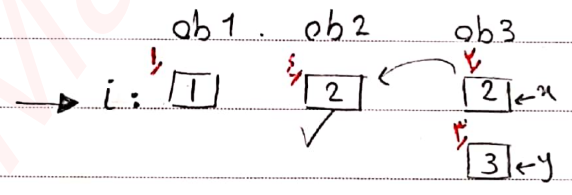
* ob1 و ob2 تفکیک ندارند
ob3 دو تا دارد

```
class x{
    public:
        int i;          } ob1
};                      } ob2
class y: public x{
    public:             } ob3
        int i;
};
main(){
x ob1;
x ob2;
y ob3;
ob1.i = 1;
ob3.x :: i = 2;
ob3.i = 3
ob2 = ob3
cout << ob2.i;
}
```

(آنکه از x به ارث می‌برد) ← ob3.x :: i = 2; — نامی که مربوط به کلاس x است

ob2 = ob3 ⟶ روش هم کلاس

خروجی: 2



Subject :

Year. Month. Date. ( )

اشاره گر به شیء:

می‌توان یک اشاره‌گر به یک شیء تعریف کرد و توسط آن به اعضای شیء دسترسی پیدا کرد. برای دسترسی به اعضای شیء از عملگر ← باید استفاده کرد.

مثال:

```
class c{
    private:
        int x;
    public:
        c(int y){
            x=y;
    }
    int f(){
        return x;
    }
    main(){
    c ob(2);
    c *p; ← اشاره گر به نام p
    p= &ob; ← آدرس ob در p قرار می‌گیرد
    Cout << p→f();
    }
```

خروجی: 2

Subject :

Year. Month. Date. ( )

مثال ۹:
[اشاره‌گر به شیء]

```
class c{
    public:
     int x;
     void f(){
        cout << 'a';
};
main(){
 c ob;
 c *p;
 p = &ob;
 ob.f();
 p->f();
}
```

ob تابع f را فراخوانی کرده
p اشاره‌گر f را فراخوانی کرده
پس خروجی a۲ دارد

خروجی : aa

مثال ۱۰:
[اشاره‌گر به عضو کلاس]

```
class c{
  public:
     int x;
     c(int a){
        x = a;
     }
};
main(){
    c ob(1);
    int c::*p;
    p = &c::x;
    cout << ob.*p;
}
```

x = 1

اشاره‌گر به متغیر x دسترسی پیدا کنه

ob.x معادل

مقدار متغیر x

p → 1

خروجی : 1

ASEMAN

Subject :

Year. Month. Date. ( )

مثال ۱۱:
[اشاره‌گر به توابع یک کلاس]

```
class c{
  public:
    int x;
    c(int a){
      x=a;
    }
  int f(){
    return ++x;
  }
};
main(){
c ob(1);
int c::*p();
p=&c::f;
cout<<(ob.*p)();
}
```

اشاره‌گر به تابعی از کلاس c
p به تابع f اشاره می‌کند
x=1
خروجی: 2

دستیابی به عضو عمومی شیء از طریق اشاره‌گر:
می‌توان آدرس عضو عمومی شیء را به اشاره‌گری نسبت داد و از طریق آن اشاره‌گر به آن دست یافت

مثال ۱۲:

```
class c{
  public:
    int a;
};
  main(){
    c ob;
    int *p;
    p=&ob.a;
    *p=3;
    cout<<*p;
}
```

خروجی: 3

اشاره‌گر p آدرس a را دارد
اشاره‌گر به عضو عمومی (a)
اشاره‌گر
محتوای آدرس p را 3
محتوای آدرس p را چاپ کند
ob.a

Subject :
Year. Month. Date. ( )

Polymorphism , virtual function — توابع مجازی و چندشکلی

منظور از Polymorphism (یعنی مورفیسم) استفاده از یک نام در چندین تابع مختلف است که کامپایلر با توجه به ورودی آنها تشخیص می‌دهد که کدام تابع را باید اجرا کند.

مثال ۱۳:

```
int f(int a){
    return ++a;
}
char f(char b){
    return ++b;
}
main(){
    cout << f(5);
    cout << f(A);
}
```

f(5) → 6 → سراغ اولی
f(A) → B → سراغ دوم
خروجی: 6B

مثال ۱۴:

```
int f(int a){
    return ++a;
}
char f(int b){
    return ++b;
}
main()
cout << f(1);
cout << f(2);
```

برنامه خطادار چون ورودی هر دو تابع integer هست و نمیدونه کدوم رو صدا بزنه

ASEM4N

Subject : جلسه آخر (۹۸/۵/۱۴)
Year. Month. Date. ( )

چند ریختی
- در زمان کامپایل ← overloading
  - Function overloading
  - operator overloading
- در زمان اجرا ← virtual Function (تابع مجازی)

انواع چند ریختی (poly morphism) :

۱_ چند ریختی در زمان کامپایل

۲_ چند ریختی در زمان اجرا

چند ریختی در زمان کامپایل :

چند ریختی در زمان کامپایل را سربارگذاری یا overloading نیز می‌گویند که به دو دسته Function overloading و operator overloading تقسیم می‌شود.

در Function overloading یک تابع می‌تواند انواع مختلفی را به عنوان ورودی دریافت کند و به نوع خاصی محدود نمی‌شود.

مثل تابع print(arg) که آرگومان آن می‌تواند از نوع integer یا character باشد.

و از operator overloading یک عملگر برای چند نوع داده‌ای استفاده می‌شود. مثل عملگر جمع که هم برای عدد و هم برای character استفاده می‌شود

Subject :

Year. Month. Date. ( )

چند ریختی در زمان اجرا:

چند ریختی در زمان اجرا را تابع مجازی یا virtual Function می‌گویند که در آن تابعی که در کلاس پایه تعریف شده و مجددا در کلاس مشتق تعریف می‌شود تابع مجازی می‌گویند. که مربایه در زمان اجرا خودش تشخیص می‌دهد کدام تابع مد نظر است.

مثال:

```
class person{
virtual void who(){ cout<<"I'm person"; }
};
class P1: public person{
    void who(){ cout<<"I'm P1"; }
};
class P2: public person{
    void who(){ cout<<"I'm P2"; }
};
main(){
person *p' = new P1;
person *p'' = new P2;
p'->who();
p''->who();
```

I'm P1

I'm P2

Subject :

Year. Month. Date. ( )

کلاس انتزاعی:

کلاسی که حداقل دارای یک تابع مجازی محض یا خالص باشد را کلاس انتزاعی می‌گویند

مثال:

```
class C{
  public:
  virtual void f()=0
};
class C2 public C1{
  public:
  void f(){ cout<<'2'; }
};
main(){
  C1 ob1;
}
```

تابع مجازی محض (خالص) Abstract

✗ خطا دارد

+ از کلاس انتزاعی نمی‌توانیم شیئی بسازیم

مثال: خروجی این مثال : BB

```
class B{
  public:
virtual void F(){cout<<"B";}
};
class D: public B{
  public:
  void F(){ cout<<"D"; }
int main()
{
B b;
B *P;
P = &b;   ⟶ B B
P->F();
D d;
P = &d;   ⟶ B D
P->F();
}
```

اگر همین صورت سؤال تغییر کند [خروجی = BD]

وقتی کلاس D مشتق شده از B است

اگر کلمه virtual نوشته نشده باشد، وقتی تابعی تابع مشتق داشته باشد

اون تابع مشتق را صدا می‌زند

ولی اگر virtual باشد خود تابع را صدا می‌زند یعنی virtual آن تابع را بالاتر می‌کند

یک زوج از کلاس d ساختیم. اشاره‌گر تابع D را صدا می‌زند

Subject :

Year, Month. Date. ( )

سربارگذاری عملگرها:

استفاده از عملگرها برای انواع داده‌ای مانند int و Double نیاز به تعریف ندارد، اما چنانچه بخواهیم عملگری را بر روی اشیاء یک کلاس اعمال کنیم نیاز به تعریف (سربارگذاری) عملگر داریم. برای سربارگذاری از تابع operator استفاده می‌کنیم.

مثال: سربارگذاری عملگر جمع مساوی (+=):

```
class T {
  private:
    int x,y ;
  public,
    void set ( int a , int b
    {
     x=a  , y=b; }
    void  operator + = (T p)
    {
      x = x + p.x
      y = y + p.y ;
    }
  void f()
{ cout << x "\t" << y ;}
int main () {
     T m , n ;
     m. set (1,2) ;
     n. set (4.5);
     m + = n ;
     m. f() ;
```

خروجی: 57 ← جمع کردن نظیر به نظیر

دو تابع m و n را از T فراخوانی کرده

x ها با هم و y ها با هم جمع می‌شوند: 1+4=5 ، 2+5=7

روی شیء m و n عملگر += انجام شده

روی شیء m با x و y تابع f

1 4 5

Subject :

Year. Month. Date. ( )

مثال: سربارگذاری عملگر جمع:

```
class c{
  private:
    int x,y;
  public:
    c(int a, int b) {x=a; y=b}
    c(){ };
    c operator +(c S)
    {
      x=x+ S.x
      y=y+ S.y;
    }
Void show()
      cout << x<<y
}
    main(){
c   a(1,2);
c   b(4,5);
c   d;
d = a+b;
d. B show();
```

خروجی: 57

کانستراکتور اول

اولین obj که میسازیم a که (1,2) است

دومی هم b است

کانستراکتور دوم

[@ ostadroyabagheri] کانال کلاس استاد
[RoyaBagherieng@yahoo.com] ایمیل استاد

Subject:
Year. Month. Date. ( )

مثال در سربارگذاری ++ :

```
classT {
  private:
    int x,y;
  public:
    T(int a, intb) {x=a ,y=b ;}

T operator ++()
{
  x+=1
  y+=1
return T(x,y);
void show(){
  cout<<x<<y;
}
int main()
{
 T ob (1,5);
  ++ob;
  ob.show();
}
```

خروجی 26

امتحان: base case ها - تعریف سازنده - تعریف مخرب انواع ارث بری
تعریف + کد + ماتریس پاور پوینت
نمودارها به عنوان سؤال اضافه

"پایان"

ASEMAN