المتطلبات غير الوظيفية

د.عمار الجوخدار

RB Informatics; 28/10/2019 تطبيقات الإنترنت

المرونة Flexibility

- هو مصطلح يطلق على الـ programمن وجهة نظر المستخدم النهائي، ويكون البرنامج flexible إذا سمح للمستخدم بتغيير سلوك محدد دون أن يعود للمبرمج (رواتب، ضرائب، مع الانتباه إلى تحديد طبيعة التغيير حيث أن هناك سلوك غير قابلة للتعديل إلا بالعودة للمبرمج).
- لابد من تحديد مستوى المرونة المطلوب في مرحلة المتطلبات والتحليل، هل هي على مستوى الحسابات أم على مستوى الإجراءات أو غيرها.
- المرونة دوماً محدودة أي ليس هناك مرونة بنسبة ١٠٠٪ فهي شيء متوّقع أثناء التحليل ، لاتتجاوز مجموعة من الاستمارات المحددة مسبقاً ضمن التحليل فبمجرد وجود تعبير رياضي أو حلقة for أو فتح table جديد بالdb نكون قد تجاوزنا مفهوم المرونة وانتقلنا إلى الscalability وهي قابلية التوسع التي تقع على عاتق المبرمج وليس الend user.
- يتم تحقيق المرونة برمجيا عبر تحويلها لمجموعة إعدادات و صلاحيات ، يستطيع المستخدم ضبطها والتحكم بها.
- يفيد الAspect في إضافة سمات جديدة عجزت مرونة التحليل عن تغطيتها، دون الحاجة لتعديل الرماز القديم بحيث نحول دون حدوث regression.
- كلما ازدادت المرونة تزداد التكلفة لأن حجم الكود البرمجي سيزداد وسيمر بحالات متعددة وستزداد مسارات الحلول وعدد مرات الاختبار وبالنهاية سيزداد تعقيد النظام وتنخفض نوعيته وأداؤه.

Audit, Alert, Tracing

- إمكانية معرفة مجريات الprogram بعد أن تم تطويره، (معرفة طلبات الزبائن عالserver وطلبات ال server عال db وبالعكس).
- يجب تحديد الطبقات التي نرغب بالتنصت عليها أثناء وضع المتطلبات مثل واجهة التخاطب أو قاعدة البيانات أو منطق العمل BL وتحدد المعلومات المراد جمعها نتيجة التنصت.

Audit: هي طريقة لمعرفة إذا كانت العمليات طبيعية أم لا (تبين أن أحدهم قام بعمل لايجوز له القيام به وأدّعى أنه لم يقم به).

Alert: وضع قاعدة ذكية تنبه أن ماحصل لايجب أن يحصل (استطاع أحد اختراق حساب والدخول عليه من حاسب آخر، يجب أن يقوم النظام بتنبيه المستخدم لعملية الدخول الغريبة للحساب لتحديد فيما إذا كانت اختراق أم لا).

Tracing: إمكانية العودة للوراء والتحقق من الآداء (مثلا بهدف زيادة الرواتب للموظفين) أو من تجاوز الصلاحيات.

- معلومات التنصت قد تفيد إما بحفظ LOG للقيام بعملية Tracing لاحقاً ، أو لتحليلها آنياً وإرسال تحذيرات Alert
- ومن أهم نماذج التصميم التي تفيد في التنصت هي AOP (Aspect Oriented Programming) وهي Design pattern موجودة بالجافا تخفف من حجم الكود عشرات المرات عدا عن سرعة التطوير والقدرة على تقسيم العمل بين أكثر من شخص ، وجدت منذ ٢٠ سنة وسنتحدث عنها بالتفصيل في المحاضرات القادمة .

إدارة المناقلات:(Transaction ACID)

تعريف المناقلة: نقول عن عملية أنها مناقلة Transaction إذا حققت أربع شروط:

- ❖ Atomicity: جميع العمليات الجزئية تتم معا أو لا.
- ❖ Coherence(التماسك): يجب أن تبقى البيانات منسجمة ومتجانسة مع بعضها البعض قبل وبعد المناقلة (ليس فقط نجاح أو فشل المناقلة).
  - منع المستخدم من حذف عملية والنظام لم يحذفها بعد من قاعدة البيانات .
  - منع إضافة مناقلة لمستخدم وهو لم يطلبها (شخص حول مبلغ مالي وهو بالحقيقة لم يحول شيء).
  - إرسال نتيجة لعملية لم تكتمل.
- ❖ Isolation(العزل): أن تكون نتيجة المناقلة مستقلة عن المناقلات الأخرى التي تتم على التوازي معها.
- ❖ Durability(الديمومة): نتيجة المناقلة لاتتغير سلبا أو إيجابا (لا يمكن أن يرسل تقرير بنجاح عملية ثم يرسل تقرير فشل بإتمامها).

مثال: عملية سحب مبلغ مالي من ATM.

أنواع المناقلات

-يوجد نوعين من المناقلات user managed transaction وcontainer managed transaction أو المناقلات التصريحية وهي الأسهل تحقيقا وأكثر مرونة.

المناقلات التصريحية :(أو المعالجة من قبل ال container وهي البيئة المشغلة للخدمات) من أجل كل تابع نحدد نوع المناقلة المرغوبة، وكل مناقلة تدعم أحد التصريحات التالية:

Required :تعني أن ينضم التابع عند طلبه لمناقلة أخرى مفتوحة أي يتم تنفيذه مع التابع الذي استدعاه بنفس المناقلة ،وإن لم توجد مناقلة مفتوحة (أي أن لتابع الذي استدعاه non transaction) يتم فتح مناقلة جديدة له.

Not Required: لا يتأثر التابع بوجود المناقلة أو بعدم وجودها (لا يحتاج إلى مناقلة ولا يضره بوجودها).

Required new: ينشئ التابع مناقلة خاصة به بغض النظر عن التابع الذي استدعاه.

Supported: إذا كان التابع المستدعي له مناقلة مفتوحة فإن التابع المستدعى ينضم إليها وإن لم يكن للتابع المستدعي مناقلة مفتوحة فلا يتم فتح مناقلة جديدة للتابع المستدعى.

Not Supported: التابع لا يدعم المناقلة لذلك يجب عدم استدعاءه بمناقلة وبحال كان للتابع المستدعى مناقلة يتم تعطيلها ريثما ينتهي التابع من التنفيذ ثم يتم استئناف المناقلة.

Never: يظهر التابع exceptionبحال تم استدعاءه وكان هناك مناقلة مفتوحة.

Mandatory: التابع يحتاج لمناقلة موجودة مسبقا وفي حال عدم وجودها يظهر exception.

Functional Scalability

- هو من أهم الخصائص الغير وظيفية لأي تطبيق.
- قد يؤدي أي تطوير أو تعديل على برنامج موجود إلى تراجع شيء كان محقق مسبقاً وقد يكون زمن التعديل على برنامج معين مساوٍ لزمن تطوير هذا البرنامج من الصفر.
- functional scalability:

هو القدرة على إضافة وظائف إضافية على البرنامج دون الاضطرار إلى تغيير شيء سبق وتم برمجته أي التصاعدية الوظيفية بمعنى أن يكون البرنامج قابل للتوسع وإضافة وظائف جديدة دون التأثير على الوظائف السابقة وبالتالي تصبح كل وظيفة بحد ذاتها قابلة للبرمجة من قبل أحد المبرمجين وهذه الوظائف تتكامل سوياً دون تدخل بشري ودون تراجع.

- AOP (Aspect Oriented Programming).

تعتبر أهم طريقة وجدت وأفضل طريقة

User Scalability

- القدرة على تحمل عدد متزايد من المستخديم دون العودة للبرمجة من جديد ودون تعديل العتاديات الحالية وفقط بإضافة عتاديات جديدة.
- هذا المفهوم مرتبط بالـ load balancing.
- في البنيان ثلاثي الأرتال 3Tiers يتم ذلك في الطبقة الوسطى BL Serverوالقابل للتكرار.
- لابد للمصمم من لحظ استهلاك الذاكرة والـ CPU لكل مستخدم.
- هناك أدوات خاصة بال bench markingومحاكاة عدة مستخدمين مثل:

Web Runner, JMeter, Open STA

وبعضها مجاني ويوجد غيرها.

تقوم هذه الأدوات بمحاكات عدد من المستخدمين.

إدارة الموارد Resource Management

تتضمن إدارة الموارد : إدارة (الذاكرة – cpu – band width – hard disk– database) ، لذلك نريد أعلى إدارة للموارد بحيث نستطيع أن نقوم بتشغيل آلاف المستخدمين على server واحد (حسب المقدرة) فعند القيام بذلك فسنستطيع أن نقوم ب scalability للتطبيق أي كلما وضعنا server فإننا نضيف على سبيل المثال ٥ الاف من المستخدمين لكل server ، أي الإدارة هي ضمان كل مستخدم أن لا يستهلك موارد أكثر من المخصص لذلك .

أي أن إدارة الموارد هي:
استهلاك أقل للموارد وعدم تجاوز الموارد المتاحة.
وبالتالي يجب معرفة لكل طلب request تأتي للنظام الذاكرة المستهلكة، HD ،cpu ،bandwidt
وبالتالي نحدد عدد المستخدمين الممكن أن يخدمهم النظام (المستخدمين المنافسين).
حسنا لكن كيف أستطيع أن أحدد على كم مستخدم يصلح هذا النظام أليس لذلك علاقة بما يقوم به كل مستخدم ؟
الكلام السابق ليس له معنى إذا كان كل مستخدم يقوم بعمل مختلف فيجب أن يكون كل طلب request يستهلك نفس الذاكرة ، نفس CPU
lazy list تقوم بتقسيم الطلبات requests إلى أجزاء صغيرة مثلا فهو نتائج البحث google
لا يتم عرضها جميعا بل كل جزء على حدا ومثال آخر في نظام بنك تقارير ضخمة نحتاج لطباعة أسماء العملاء تستغرق وقت طويل والنظام online فلا نسمح له القيام بهذه العملية ممكن أن نضع مخدم server جانبي و back up DB للقيام بهذه الأعمال.
thread pooling يضمن عدم عمل أكثر من n مستخدم على التوازي بحيث نضمن عدم انهيار النظام
session العمل على كل مستخدم بحجم ثابت من ال RAM فنحتفظ لكل مستخدم بمعلومات من حجم معين .
ملاحظة :هذه الصعوبات لم تكن موجودة في نموذج Client/Server لأنه يتم العمل على مستخدم واحد بينما برامج ال Web based مفتوحة لكل الناس.

Clustering, Load Balancing and Fault Tolerance

وهي تفيد في تقسيم مجموعة من الطلبات requests بين مجموعة من المخدمات ونميز بين حالتين هما:

- Load Balancing: ويكون في حال انقطاع أو توقف المخدم فهنا يجب على المستخدم المحاولة مرة أخرى.
- Clustering: هي مكلفة إلّا أنها تعتبر شفافة بالنسبة للمستخدم ، لأنه في حال توقف المخدم الذي يعمل عليه ينتقل لمخدم اخر دون أن يشعر لأن المخدمات تكون مشتركة بال session.

Availability

تقاس من خلال القانون التالي: $\frac{time\ to\ fail}{time\ to\ repair+time\ to\ fail}$

تعني fail: هذا البرنامج كل متى سيتوقف؟

مثلاً: إذا كان لدينا نظام تشغيل windows كل متى يحتاج restart؟

يكون ذلك وفق ما نقوم من عمليات عليه فاذا كانت البرامج التي نعمل عليها هي فقط .. exel,word فمثلاً تحتاج ٣ أيام أما إذا كانت عمليات برمجة وتطوير فممكن أن يحتاج عمل restart عدة مرات في اليوم والذي بدوره سيأخذ كمية من الوقت من إعادة الاقلاع واعادة المبرمج للنقطة التي توقف عليها.

إذاً الإتاحية لتكون عالية يجب أن يتحقق واحد من الأمرين إما أن يكون زمن *repair* قصير أو أن يكون زمن *fail* كبير.

يوجد وجهتين نظر للإتاحية:

من وجهة نظر الجهة الطالبة:

- يجب ألا يتوقف النظام أكثر من ساعة في اليوم أو في السنة أو ألا يتوقف أبدا.
- يجب ألا يتوقف في أوقات معينة مثل الدوام الرسمي.

من وجهة نظر المصمم (التنفيذ):

كيف سيؤمن availability عالية؟ يجب عليه أن يفكر بعد أشياء وهي:

-إدارة transaction.

-إدارة threads: أي ابقائها على قيد الحياة ولا يوجد طريقة لذلك سوى القيام ب continuer أي كلما مات thread يقوم continuer بإنشاء thread بديل ولكن عند انشاء threadبديل كيف سيستعمله البرنامج؟ هذا يسمى بل thread pooling: أي أن البرنامج لا يفتح threads بل يجب أن يعثر على مجموعة threads حيث عند احتياجه او موت thread يقوم بسحب واحد منها.
حيث في حال حدوث عطل عطل مع مستخدم فلا يتوقف النظام بل يتوقف النيسب المسؤول عن تخديم هذا المستخدم.

-load balancing أو clustering : تعطي نوع من التسامح مع الأعطال في حال كان على مستوى المخدمات.

-إذا كانت المشكلة Software ستوقف نيسب واحد ، واذا كانت Hardware ستوقف مخدم وحيد.

الاستعادة من الأعطال Fail Over

تعني Fail بأن النظام توقف نهائياً عن العمل أي لا يوجد continuer يُنشأ threads عوض عن التي ماتت ولا يوجد availability

fail over وهي القدرة أن نعيد تشغيل البرنامج ويكون ذلك بالقيام بعملية bulid او roleback لل transaction.

تلخيص:

-fail over : هو الانطلاق مرة ثانية بالنظام بشكل صحيح وذلك بعد توقفه نهائيا

-زمن إصلاح العطل : عندما يحدث عطل معين فكم الزمن اللازم لإعادة تشغيل البرنامج والقيام بعملية Back up لل DB والزمن اللازم لإعادة تشغيل application server.

-يؤثر على fail over عوامل عديدة:

عدد المخدمات – استعمل ACID بشكل صحيح – وجود Raplication لل DB – القيام ب Back up يومي لقاعدة البيانات

-أحيانا يمكن اللجوء لأعمال يدوية لإعادة إقلاع النظام حيث يقوم بعض المهندسين بتحليل المعطيات وتحديد المعطوب منها وإصلاحه آليا أو يدويا أو بالعودة للمعطيات الورقية .

Usability سهولة الاستعمال

- يبدأ التفكير بسهولة الاستعمال أثناء وضع المتطلبات وذلك:

بتحديد المستوى التعليمي للمستخدم النهائي – الزمن اللازم لإدخال معطيات محددة وتكرار الإدخال – اليومي أو الشهري أنواع التقارير المطلوبة – التنبيهات والتحذيرات – زمن التدريب المناسب لطبيعة عمل المؤسسة – سهولة الإلغاء والتعديل.

- يختلف مفهوم سهولة الاستعمال تبعا لنوع البرنامج (أي أنها نسبية) :

برنامج مصرفي لا يحتاج لسرعة في إجراءات العمليات بل على العكس يرغب دوما بالقيام بالعملية على مرحلتين إنشاء وتدقيق – برنامج شبكة اجتماعية لا بأس بالسرعة مقابل احتمال أكبر لأخطاء الإدخال

Accessibility

أي القدرة على الوصول للمعلومة حتى لو كان الإنسان يعاني من مشاكل معينة في النظر أو السمع أو الحركة أو غيرها.

على سبيل المثال:

- ألا يحتاج المستخدم لأكثر من زر لتحقيق وظيفة لأن المستخدم قد يكون لديه مشكلة في اليدين، بالتالي يمكن تثبيت زر الـ shift لمن لا يستطيع استخدام كلتا يديه.
- وجود المكبرة في نظام تشغيل windows – لضعيفي النظر.

Concurrent Access التنافس

متى تحدث؟

عند تخديم مستخدمين اثنين على نفس الموارد.

في مثال الـ Atm: من الممكن أن يدخل مستخدمين لنفس الحساب ويسحبان معا كمية أكبر من الموجودة.

إدارة الـ Concurrent Access تتم بعدة طرق:

- من خلال استخدام الأقفال (وبذلك نمنع التعديل الجديد حتى ينتهي القديم) وبالتالي يصبح التخديم كـ رتل يُخدَم الواحد تلو الأخر.
- ترك التخديم عشوائي (كترك دخول وخروج الزبائن عشوائيا) سيحدث لدي مشكلة deadlock.
- تنافس منظم.

Producer/Consumer

لحل التنافس بين المنتج والمستهلك يجب أن يتحقق ما يلي:

- يجب ألا ينتج منتجا قبل استهلاك المنتج الأخر.
- ألا يستهلك المستهلك منتجا قبل غير كامل الإنتاج.
- المستهلك لا يستهلك منتجا مرتين.

مثل برامج الألعاب:

الـ producer: يقوم بتوليد للحركة والـ consumer يقوم بالإظهار.

- العمل يتم بشكل تفرعي وإلا سيسير العمل بسرعة البطيء.
- يجري العمل على one producer لـ n consumer بحيث يضع ما ينتجه الـ producer في buffer على شكل رتل.

كيف نحل مسألة المنتج/المستهلك:

```
class Product{
int p[10];
int k=0;
  void produce() {
     for (int i = 0; i < 10; i++) {
     p[i]=k; } k++;
 }
  }
 void consume() {
 System.out.println(p);
}
void main(String[] args) {
    Product p=new Product();
 Producer pr=new Producer(p);
 Consumer c=new Consumer(p);
pr.start();
  c.start ;()}  }
```

```
class Consumer extends Thread{
   Product p;
 Consumer(Product p) {
this.p=p;
}
 public void run() {
  while(true) {
  p.consume();
 }
 }
}

class Producer extends Thread {
Product p;
 Producer(Product p) {
this.p=p;
}
 public void run() {
while(true) {
 p.produce();
  } } }
```

حل مشكلة استهلاك منتج غير كامل:

```
class Product {
int p[10]; int k=0;
 synchronized void produce() {
   for (int i = 0; i < 10; i++) {
  p[i]=k; } k++;
   }
   }
 synchronized void consume() {
  System.out.println(p);
} }
```

حل مشكلة استهلاك المنتج مرتين أو عدم استهلاكه باستعمال الانتظار الفعال:

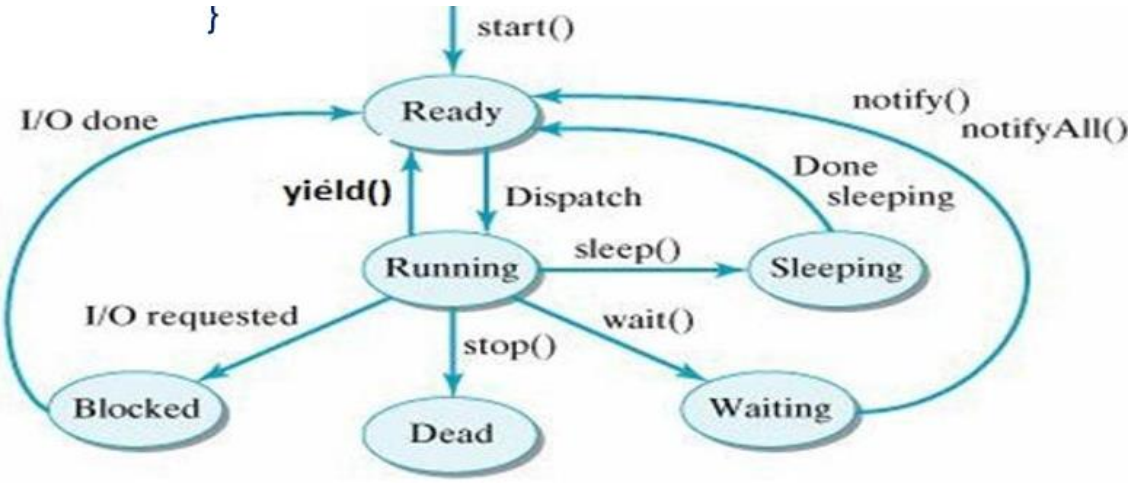
```
class Product{
  int p[10];
int k=0;
   boolean isAvailable=false;
synchronized void produce() {
     if (isAvailable) { return; }
 for (int i = 0; i < 10; i++) {
  p[i]=k; } k++;
 }
   isAvailable=true;  }

synchronized void consume() {
    if (!isAvailable) { return; }
 System.out.println(p);
  isAvailable=false;   }  }
```

حل مشكلة استهلاك المنتج مرتين أو عدم استهلاكه

باستعمال حالات الترقبMonitoring status

```
class Product{
 int p[10]; int k=0;
  boolean isAvailable=false;
    synchronized void produce() {
      while(isAvailable) { wait ;()}
    for (int i = 0; i < 10; i++) {
        p[i]=k; } k++;
 }   isAvailable=true;
notifyAll ;()}
synchronized void consume() {
   while(!isAvailable) { wait(); }
   System.out.println(p);
    isAvailable=false;
 notifyAll();   } }
```



انتهت المحاضرة ^^