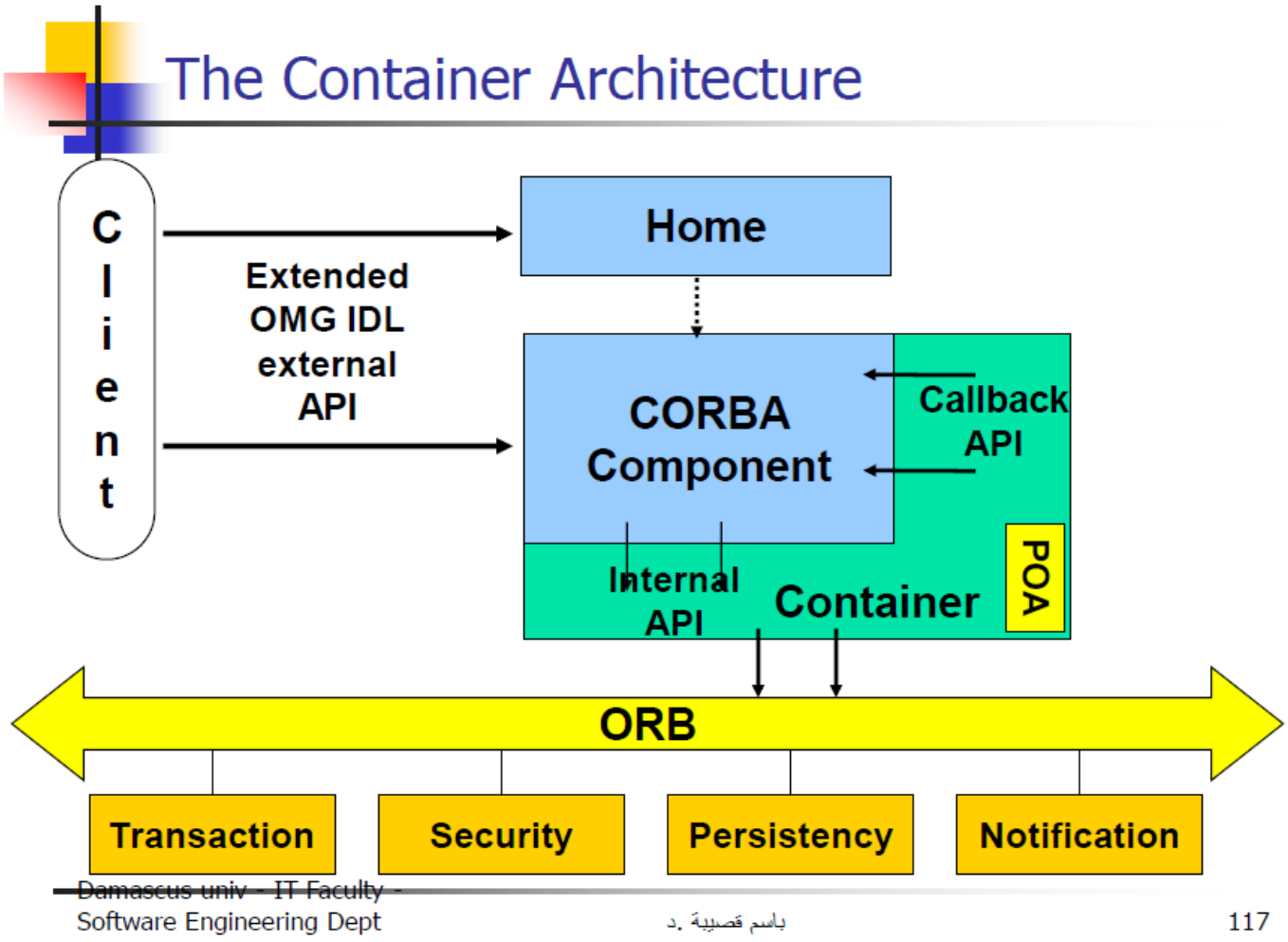
سنتحدث في هذه المحاضرة عن :

1. تتمة المكونات البرمجية.
2. Aspect.

تكلمنا سابقاً أن facet و event sink لهم خصوصية لأنهم يحتاجوا إلى تنفيذ، وعندما قاموا بعمل الهرمية الخاصة بالمكونات البرمجية، قد صرحوا بضرورة عمل implementation وهم class ويوجد function للتنفيذ تقوم بتنفيذ الـ function الذي تقدمه interface أو تنفيذ function الخاص بالـ notification.



لدينا عنصر مهم بالـ component اسمه container وهو الذي يقدم التقنيات ويدير حياة الأغراض البرمجية، له أنواع دخل CCM ، إما أن تكون كل function داخل class واحد، أو عمل class لكل interface ويتم التعامل معه كـ object.

Container View

- A container encapsulates one or several POAs
- A container manages one kind of component
  - `entity`: persistent, primary key, and explicit destruction
  - `process`: persistent, no key, and explicit destruction
  - `session`: exists during a session with the client
  - `service`: exists during an invocation
  - `EJBsession`, `EJBentity`: for EJBs
  - `Empty`: user-defined policy
- References are exported through *FactoryFinder*, *Naming*, or *Trader* services

مزايا الـ container

1. تزويد المكون البرمجي بالمكتبات اللازمة: يتم تزويد المكون البرمجي بالمكتبات من خلال كتابة اسم المكتبة (استدعائها) داخل ملف XML ويتم تحميلها عند الإقلاع.
2. اعتراض الـ method: (الفائدة إضافة خدمات)

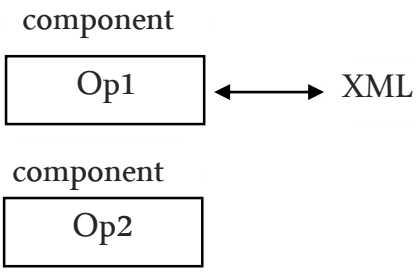
عند تنفيذ أي غرض برمجي، الذي يقوم بتنفيذ خدمات النظام فإن الـ container يقوم باعتراض الـ method فهذا يسمح بإضافة خدمات غير وظيفية مثل (... security).

Transaction

وبتعريفها العام هي تنفيذ لعدد من العمليات op1, op2, op3

يتم القيام بأي مناقلة من خلال تعريف الـ operation وسلوكهم داخلها.

لكن هنا الأمر يتم بالعكس:



حيث عند تعريف الـ component يوجد ملف XML وهو ملف خاص بالإعدادات configuration أقوم بإعطاء خواص لكل method، وبالتالي بدلاً من بناء transaction بالطريقة التقليدية سنقوم بذكر كل method كيف سيتم استخدامها إذا كانت ضمن transaction.

خلاصة: لا أخلق آليات لإدارة transaction وإنما الـ method بحد ذاتها كيف تتصرف عندما تكون ضمن الـ transaction. ولها عدة خواص:

Transactions

- Container-managed at the operation level
  - NOT_SUPPORTED
  - REQUIRED
  - SUPPORTS
  - REQUIRES_NEW
  - MANDATORY
  - NEVER
- Self-managed using the `Components::Transaction::UserTransaction` API which is mapped to CORBA transactions

Damascus univ - IT Faculty - Software Engineering Dept — د. باسم قصيبة — 122

- Not_supported: إذا تم تحقيق الـ method أولاً، لا يؤثر على الـ transaction.
- Required: تنفيذ الـ method هام في الـ transaction.
- Required_New: كل الاستدعاءات داخل الـ transaction تعالج بشكل تكراري إذا أخفقت.

(أنصح بمعرفة الفروق بينهم بشكل أفضل لأن لم يتم توضيحهم أكثر ضمن المحاضرة).

ملاحظة:

عند تعريف الـ method لن تكون داخل transaction بشكل مباشر، إنما ستكون مستدعاة داخل operation ثانية فإن كانت required فكل method داخلها يطبق عليها سياسة الـ transaction.

➢ كيف سيكون تدخل الـ container هنا ؟!

عند استدعاء الـ operation1 الخاص بالـ component، أذهب لملف XML الخاص بالـ component نفسه، واقرأ خواص الـ transaction، ويتم تحديد سلوكه من خلال الخواص. يفتح Transaction له وكلما استدعى operation1 يقرأ خواصه.

➢ كيف تتم عملية إدارة الـ Transaction هنا ؟!

تتم من خلال الاستفادة من *CORBA non functional service* لها مكتبة خاصة نقوم باستخدامها وهي لإدارة الخدمات الغير وظيفية.

(*Real time, security, event notification, life time ...*).

بهذه الطريقة تمت إضافة المهام الغير وظيفية إلى المكونات البرمجية.

تلخيص:

تم الاستفادة من اعتراض الـ container للـ method فيطبق الـ configuration الخاص بكل method وبالمقابل أنا آخذ السلوك ضمن الـ Transaction.

ملاحظة: هذا الأمر غير مدعوم لكل أنواع الـ component، فهو ضمن CORBA و fractor osgi .

صدر نموذج جديد للـ component وهو هام للغاية.

- **SCA (Service Component Architecture)**

وهي دمج بين الـ component والـ web service.

على اعتبار أن web service تعامل على أنها component لأنها خدمة متاحة للجميع.

التحزيم الخاص بالتنفيذ *packaging of implementation*

بعد الانتهاء من كتابة الـ component والـ config الخاص به، يجب أن نقوم بعملية تحزيم (تجميع) لمجموعة المكونات المرتبطة مع بعضها ويتم ذلك عن طريق أداة رسومية (tool) ويتم توليد ملف XML يوضح العلاقات

<component C1>

ويتم ذكر interface الخاص به <uses username = C2>

وهنالك أداة تقرأ ملف التجميع *Assembly* وتضع كل component على أي حاسب موجود (ما هو *IP server component*).

وبالتالي بعد قراءة ملف XML فإن الـ component1 موجود مثلاً على الجهاز 1 ، فيتم التخاطب مع *server component* الخاص به ويتم عمل *instance* من هذا الـ *component* مع توفير كل ما يحتاجه المكون البرمجي ليعمل (*c++, ...*).

هذه المعلومات متوفرة بملف الـ XML .

ويتم تحديد مكان المكون الثاني والثالث والرابع، وبعدها تكون عملية الـ *connection* بينهم (تبادل مراجع) ومن ثم يتم التنفيذ.

ملاحظة:

- عملية الـ *deployment* مؤتمتة.
- يجب أن يكون ضمن الأجهزة التي تحوي المكونات البرمجية *server component*.

انتهت جميع أفكار المكونات البرمجية، تم التنبيه لأهمية (آلية *container* لاعتراض الـ *method*) .

مقدمة:

أتت الـ Aspect لتعزيز مفهوم الـ separation of concerns (وتم التعرف عليها داخل MDE-meta modeling).

وهذا المفهوم يقول أن كل وحدة في البرنامج (unit) يجب أن يكون لها هدف(وظيفة) واحدة قدر الإمكان، وهذا يوفر سهولة الاستخدام.

ما الفرق بين البرمجة غرضية التوجه والـ aspect ؟

البرمجة غرضية التوجه جاءت الفلسفة الخاصة بها تجميع الخواص المشتركة مع بعضها البعض (common concerns).

كان هنالك مشكلة بالـ*oop* (غير تبادل المراجع) لدينا بعض المهام في النظام تسمى (cross cutting concerns) خدمات تقطع كل unit النظام مثل الـ security.

مثال:

موقع فيه معلومات حساسة، وكل تعديل على *DB* يطلب اسم المستخدم وكلمة المرور، وعلى فرض تم تغير سياسة الـ security ونريد زيادة التحقق من خلال بصمة اليد.

لتحقيق السابق سنقوم بفتح الكود وكل ولوج لـ*DB* سيتم إضافة هذه الخدمة وهذه عملية مرهقة.

إنّ الـ Aspect تلتقط cross cutting مثل مبدأ البرمجة غرضية التوجه، ويعملان سوياً لأنها تعتمد على *model* غرضي التوجه.

Terminology:

Aspects include a definition of where they should be included in a program as well as code implementing the cross-cutting concern.

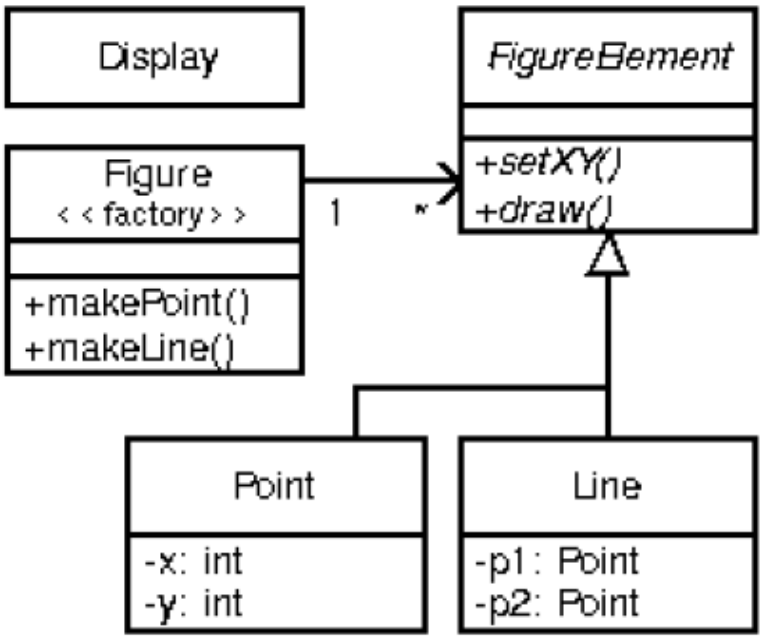
- A ***join point*** is a well-defined point in the program flow "where something happens", examples:
  - When a method or a constructor is called or executed.
  - When an exception is thrown.
  - When a variable is accessed or updated.
  - When an object is initialized.
- A ***pointcut*** is a group of join points.
- ***Advice*** is code that is executed at a pointcut.
- ***Introduction modifies*** the members of a class and the relationships between classes.

مفردات الـ Aspect

- **Join point:** إن الـ model غرضي التوجه يسمح لك باستدعاء method وبناء غرض تعديل قيمة حقل، قراءة قيمة حقل، وكل ما سبق يسمى عند Aspect: Join point وهي كل الأحداث الواردة في برنامج غرضي التوجه (listiner الخاص بالـ system).
- **Point cut**: وهي تجميع لـ method الهامة.

مثال من موقع (AspectJ أو JAspect "هما مكتبتين مختلفتين").

Example: The Figure Element example:



لدينا ضمن المخطط الموضح أعلاه صفين تتم وفقهم العمليات، فيكون تطبيق Point cut كالتالي:

- A pointcut named move that chooses various method calls:

```
pointcut move():
    call(void FigureElement.setXY(int,int)) ||
    call(voidPoint.setX(int))               ||
    call(void Point.setY(int))              ||
    call(void Line.setP1(Point))            ||
    call(void Line.setP2(Point));
```

- عندما نستدعي من أي غرض من نوع Point← set X، إن aspect يتدخل عندما يكون لديك استدعاء set X لأي غرض من نوع point.
- أو عند استدعاء set Yلغرض من نوع Point← تمت عملية join point (تجميع) عند استدعاء set Y لغرض من نوع Figure Element. أو يمكن استدعاء figure*(..)
- إن * تعني أي method ضمن figure، وممكن استخدام هذه الميزة مثلاً * create وبالتالي نقصد كل الـ method التي تبدأ بـ create أريد أن أعاملهم بطريقة معينة.
- (..) مهما كانت الباراميترات.

هكذا أصبح بالإمكان التقاط مجموعة من الأحداث مع بعضهم البعض.

- ***Advice*** **:** يمكنه التدخل قبل أو بعد

- Advice (code) that runs before the move pointcut:

```
before(): move() {
System.out.println("About to move");
}
```

- Advice that runs after the move pointcut:

```
after(): move() {
System.out.println("Just successfully moved");
}
```

مثلاً بالـ security لازم يدخل قبل، وبالتالي advice هي قطعة الكود التي ستطبق قبل حدث معين.

Aspect مثل container تستمع لأحداث بالنظام.

فنلاحظ أن هذا الحدث هو مصنف ضمن فئة معينة وأعلم ماهي advice المكتوب ويتحدد التنفيذ (قبل، بعد) على أساسه.

ملاحظة:

Befor دائماً لـ security، After مثال شهير loging (سجل العمليات).

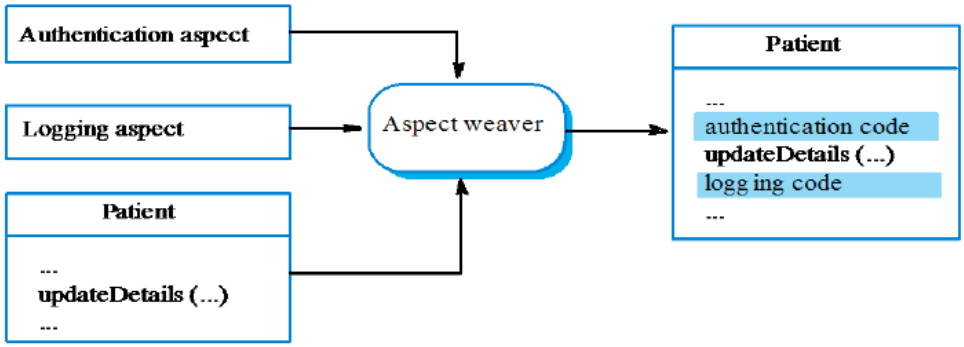
إن Aspect أخذت شهرة كبيرة.

أي شيء cross cutting concern داخل النظام ← أقوم بعمله advice وأعلم أين سيدخل، وعندما أريد التعديل أعدل على advice.

➢ كيفية عمل Aspect (Aspect weaving)

How to implement and execute? (Aspect weaving)

- Aspect weaver is an Aspect compiler that process source code and weave the aspects into the program at the specified point-cuts.
- Three approaches to aspect weaving:
  - Source code pre-processing.
  - Link-time weaving.
  - Dynamic, execution-time weaving: Aspect weaver watches over the execution and weaves to it.



1. **Source code pre_processing** وهي الأبسط والأوضح.

الميثود المهتم فيه هو updateDetails داخل patient

pre_processing يقوم بفتح كود النظام ويأخذ *method* (updateDetails) ويضع في مقدمتها advice وفي آخره advice ويترجم ويعيد لك النظام، وبالتالي عند تنفيذ النظام فإنه يتم تنفيذ كل شيء.

وعند تعديل aspect فإن النظام يعدل بشكل فوري.

➢ مساوئ الـ aspect

ممكن لنفس الميثود أن يكون له أكثر من aspect أي شخص *A* قام بعمل aspect وعرف متحول *c*، وشخص آخر *B* قام بعمل aspect مع متحول *c* (*int c / float c*) هنا سيحصل تعارض في كومبايلر الجافا، ومن هنا تأتي أهمية وجود هندسة البرمجيات بمستوى أعلى لتتم عملية الضبط.

2. **Link_time weaving**
3. **Dynamic** أكثر شيوعاً.

والمثال الأشهر ضمن الجافا هو *Byte code* بحيث يمكنني ضمنه أن اتسمّع *listening* من خلال *JVM* هنالك بعض الخدمات التي تقوم بإعلامي ما الذي يحدث بالبرنامج أثناء التنفيذ (تم استدعاء *op1* ثم *op2* ..) أي يعطي سلسلة التنفيذ.

وبالتالي هذا يساعد في عمل *dynamic model*.

ملاحظة:

Aspect و *JVM* يعملان ككتلة واحدة حيث عند استدعاء *op1* يقوم الـ *Aspect* باستدعاء *advice* دون التدخل بكود النظام ويقوم بالمعالجة الخاصة به.

مثال:

على فرض لدينا نظام هو عبارة عن *point* (شاشة مراقبة في أبراج طائرة)، مثلاً تطور البرج وكان هنالك أشخاص آخرون يقوم بالمراقبة وأنت لا تريد تعديل كود النظام. (شاشة - *point*)

تريد عمل *extension*، نقوم بكتابة *Aspect* كالتالي:

- Point observing system:

```
aspect PointObserving {
   private Vector Point.observers = new Vector();
   public static void addObserver(Point p, Screen s) {
      p.observers.add(s);
   }
   public static void removeObserver(Point p, Screen s) {
      p.observers.remove(s);
   }

   pointcut changes(Point p): target(p) && call(void Point.set*(int));
   after(Point p): changes(p) {
      Iterator iter = p.observers.iterator();
      while ( iter.hasNext() ) {
         updateObserver(p, (Screen)iter.next());
      }
   }

   static void updateObserver(Point p, Screen s) {
      s.display(p);
   }
}
```

اسم *Aspect* : هو *point observing*

Private هو *advice*

Vector point.observer : أطبق نموذج *subject observer*

Add observer: نضيف على هذه الـ *point* ← *screen* معينة، وحتى أستطيع أن اختار بعض النقاط لشخص وبعض النقاط لشخص آخر

p.observers.add(s) : إضافة على *point class* شعاع *vector* من *observer* وتم إضافة *Add observer*، إذا ضفت *screen* ضيفها على هذه *point* ، وهذا كله وانت خارج الكود.

Public void removeobserver: لإزالة الـ*subject*.

Pointcut change: عندما يحصل تغير على *point* (عند مركبتها يتغير *x,y*) وهي *Pointcut* مكتوبة سابقاً.

Target(b): الهدف هو النقطة.

Call(point.set.)*: *SetX* او *SetY*.

بعد أن يحصل التغير نقوم بعمل *iterator* لزيارة جميع عناصر الـ*observer* وأقول لهم *updateobserver(p,screen.next)*.

هكذا تم تمديد النظام دون الدخول في البنية الداخلية ويتطلب فقط معرفة في التصميم الخاص بالنظام.

➢ ما اهتمام الـ*Software Engineering* بالـ*Aspect*

إن هندسة البرمجيات هي عملية ضبط لعملية معينة.

مثل ضمن الـ*oop* الهندسة تقدم مراحل تضبط عملية التصميم ليكون جيد (جمع متطلبات ..)، وهنا نفس الحالة يجب تحليل النظام لمعرفة هل هذه *Aspect* أو لا.

وهنالك عدة نظريات ومنها:

- *Jacobson*: الذي قام بتصميم وتعريف الـ*usecase* فقد قال أنه كل *usecase* من نوع *extend* هو مرشح ليكن *Aspect*.
- عند النظر في مخطط الحالات *usecase diagram* فعندما نجد أنه أكثر من خدمة في النظام يستخدم من أكثر من شخص (*Actor*) يعتبر *cross cutting concerns*.

المطلوب في الامتحان

البرمجة وليس هندسة البرمجيات، ويكون النموذج ليكن نظام تريد القيام بتعديل أكتب *Aspect* المناسب للتعديل.

- **الفرق بين *dynamic* و *source code pre_processing***
  - *source code*: يتم بحيث تحشر الكود الخاص بالـ*Aspect* داخل *core* النظام ويتم توليد نظام جديد.
  - *Dynamic*: يقوم بالاستماع للأحداث مثل *java, jvm*.

ملاحظة:

JVM تساعد في توصيف الأنظمة ورسم أتومات وصفي لها *activity diagram*.

نموذج امتحاني:

نظام *Chating*

لدينا *cordinator* للنظام و *node*.

الـ*node* تسجل عند الـ *cordinator* .

cordinator يأخذ قائمة من *node*

عند إرسال رسالة *send msg*

نظام الـ*chat* عند إرسال رسالة يذهب لكل *node* ويستدعي ميثود *display* لنفس الرسالة عند الكل.

قم بتوسيع التطبيق باستخدام *aspect* لحتى عند إرسال الرسالة يلحق معها الزمن والتاريخ.

نأخذ الباراميتر ونعدل عليه، أعمل *advice*

pointCut → cordinator.sendmsg(string).Target(cordinator)

يرد هنا خطأ شائع ويتم اختيار *node*، ال *node* هم الأجهزة والتعديل يجب أن يكون بال*server*.

وهنا الشيء الذي يحتاج التعديل *Msg* فيتم إضافة لها *time,date* و *befor*

وال*display* تكون عند ال*node*.

أعدل على الباراميتر ويؤخذ التعديل.

ملاحظة: ضرورة الرجوع إلى سلايد ال*aspect* لدراسة الكود كاملا .

تم بفضل الله وعونه الانتهاء من مقرر هندسة البرمجيات3 _القسم النظري

نتمنى أن نكون قد قدمنا ما بوسعنا لذلك وأن تكونوا قد استفدتم من هذه المحاضرات

نسأل الله تعالى أن يتقبل منا هذا العمل خالصاً لوجه الكريم

وما كان من صوابٍ فمن الله...وما كان من خطأ فمن أنفسنا

فريق هندسة البرمجيات3 _القسم النظري

Eman Zyadeh _ Enas Alhalabi _ Hamida Abo Rshed