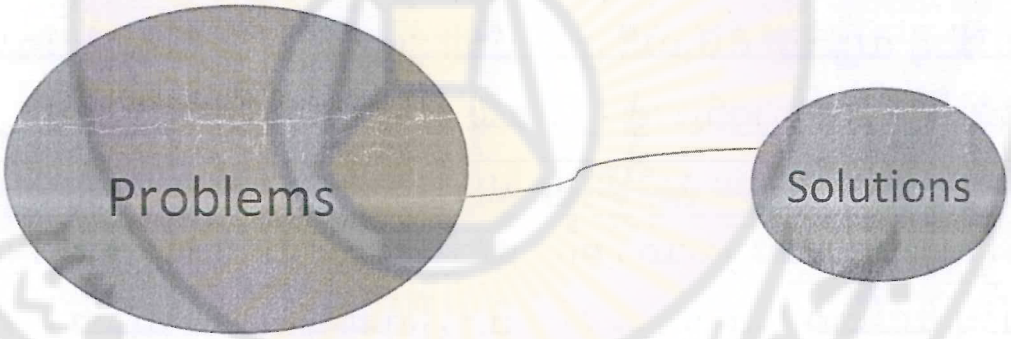
هندسة البرمجيات علم شارك بين مفهومي الهندسة والبرمجيات لذلك أول محاضرتين سوف تكون مفاهيم نظرية أساسية Terms و سنكتشف أن وظيفة مهندس البرمجيات هي إيجاد الحلول ورسم البنيان المعماري , و رسم البنيان المعماري سيحتاج لأدوات tools وبالعملي سنبدأ بها مباشرةً.

بدايةً مامعنى كلمة "هندسة" وما معنى كلمة " برمجيات ":

الهندسة Engineering :



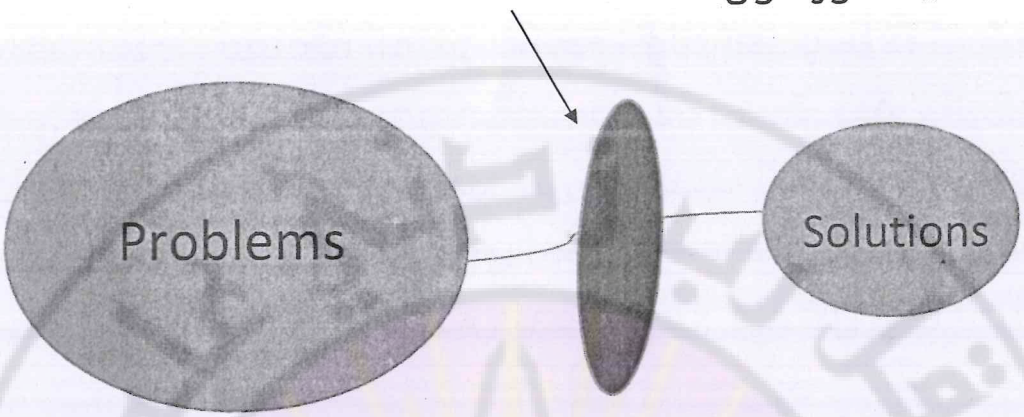
هي الطريقة التي نستطيع بها ايجاد الحل الأنسب من فضاء حلول لمشكلة ما موجودة في فضاء من مشاكل.

ومن هذا المنطلق نستطيع أن نقارن بين مهندس ناجح, مهندس جيد ومهندس عبقري فمن خلال نقطتين نستطيع ان نكون مهندسين ناجحين:

1- توسيع فضاء الحلول والإطلاع على حلول اكثر لكن هذا لا يكفي!

2- القدرة على ربط الحل المناسب مع المشكلة.

ويكون الحل مناسباً بأن يكون الأمثل, الأسرع, الأقل تكلفة والأدق ومنه علم الهندسة يتمحور حول هذا الفضاء:



والذي يحوي مجموعة من الطرق Methodologies والنظريات Theories وأُطر العمل Frameworks والأدوات Tools التي تضمن إيجاد الحل المناسب للمشكلة التي نواجهها.

البرمجيات Softwares :

هي كائنات عبارة عن مجموعة من التعليمات التي تهدف لقيادة الحاسب نحو إنجاز مهمة محددة.

ربط المفهومين:

بالنظر حولنا نجد أن البرمجيات أصبحت تستخدم في مجالات حياتنا كافة كـ برامج الطيار آلي, برامج مراقبة وتحكم بالقطارات, برامج بنوك, محاسبة, جامعات ....

ولدينا من جهة ثانية برامج أخرى كـحل المعادلات من الدرجة الثانية والأخيرة ليست بنفس السهولة للبرامج المذكورة أولاً حيث لها متطلبات كثيرة فأصبحت تعامل معاملة المنتج **Product** ولم تعد وسيلة أو شئ سهل التنفيذ بل يجب أن تتمتع بمواصفات محددة (دقة, ضمن الوقت المحدد, تكلفة دنيا) وبذلك نجد العديد من المنتجات المختلفة والتي تخدم نفس الهدف **Purpose** كمثال برنامجي Word و LaTeX ونظامي Linux و Windows ونحن كمستخدمين ما الذي يجعلنا نختار ونفضل هذا المنتج عن ذاك ؟؟

لذلك تحوّلت العملية من تقنية إلى **Business** وعند تحسين هذا المنتج كتخفيض السعر على سبيل المثال تتاح لنا القدرة على المنافسة أكثر في سوق العمل حيث لم يعد الهدف هو كتابة برنامج وفقط بل الهدف هو كيفية تصميم وتحليل هذا المنتج لإيصاله بأحسن وأجود طريقة ممكنة.

وعند التفكير بالتحسين لابد أن يخطر ببالنا **الهندسة** والتي هي **طريقة الحصول على أحسن وأفضل حل** وبذلك ينتقل مفهوم الهندسة إلى **تصنيع** المنتج البرمجي بسلاسة ومن هنا يأتي باختصار مفهوم علم:

هندسة البرمجيات وهو **تطبيق المفاهيم الهندسية في تطوير المنتجات البرمجية**.

The application of a systemic, disciplined, quantifiable approach to the development, operation, and maintenance of software and the study of these approaches. [IEE93]

ولذلك تم إيجاد طرق وأدوات خاصة لتسهيل **تطوير** المنتج البرمجي وعلم هندسة البرمجيات هو مجموعة الطرق والنظريات والادوات التي تهدف للوصول إلى المنتج البرمجي بأحسن وأفضل طريقة ممكنة وضمن الإطار الزمني **Time Frame**.

ولذلك انتقل هذا العلم إلى مستوى الـ**Business** ولم تعد تمارس كهواية حيث أصبحت لهذه المنتجات البرمجية جانب خطير وهو **Safety Critical** وأصبحت هذه البرامج تؤثر على حياة البشر (كالطيار الالي)

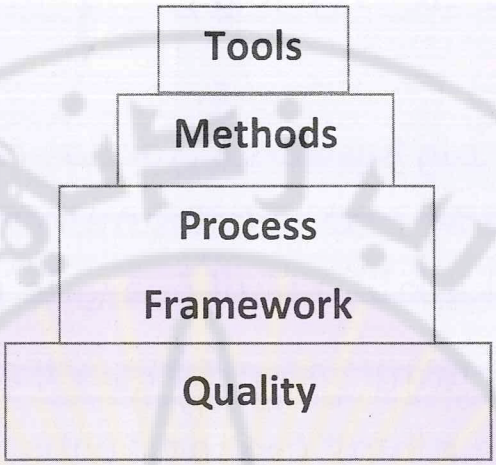
ومن هذا المنطلق ونتيجة التوسع الكبير في البرمجيات في أدق تفاصيل حياتنا أصبحنا نحتاج لطرق أساسية ومحددة المعالم والنتائج للوصول بمنتج كفؤ ومضبوط نستطيع الاعتماد عليه وهذه الطرق هي ما سنتعلمه في مقررات **هندسة البرمجيات**.

التصنيع والتطوير والتحسين :

هل نقول أننا **نصنع** المنتجات البرمجية؟

التصنيع هو الحصول على منتج فيزيائي ذو حجم وكتلة ومساحة ونستطيع لمسه ومشاهدته في مراحل صنعه أمّا المنتج البرمجي فهو كائن غير فيزيائي لانستطيع التفاعل معه لذلك نطلق على صناعة البرمجيات مصطلح **تطوير البرمجيات** **Software Development** وعلينا الانتباه وعدم الخلط بين المعنى في اللغة العربية لكلمة " **تطوير** " الشئ ونقله من مرحلة إلى مرحلة وبين معناها في " **تطوير البرمجيات** " والتي نقصد بها صنع المنتج من الصفر **From Scratch** وعندما نقصد تحسين المنتج البرمجي يمكننا أن نقول **Software Improvement**.

ويمكن فهم مصطلح هندسة البرمجيات عن طريق ***Layers Technology*** أو مايسمى بمفهوم هندسة البرمجيات كـ Layers وهي:



1- **Quality:** وهي التي تشمل جميع الطبقات والتي تضبط جودة مراحل تطوير المنتج البرمجي.
2- **Process Framework:** وهي ليست طريقة وإنما هيكلية من الإجرائيات لنصل إلى المنتج البرمجي كما أننا عند تطبيقها سنصل حتماً إلى النتيجة المطلوبة.
3- **الـ Methods:** تشمل الطرق المتبعة لتنفيذ نشاطات التطوير البرمجي ولنتمكن من تنفيذها نحتاج إلى tools معين كما أن استخدام tools غير مناسبة تؤثر على جودة المنتج.
4- **Tools**

وبالتالي *Layers Technology* أو مايسمى بمفهوم هندسة البرمجيات كـ Layers هي: **مجموعة من الطبقات بحيث تؤثر كل طبقة على جميع الطبقات التي تعلوها**. فطبقة الجودة يجب أن تشمل نشاطات التطوير وطرقها وأدواتها ونشاطات التطوير تحتاج الى مجموعة من الطرق لتحقيقها والطرق تحتاج الى مجموعة من الأدوات إلى تنفيذها.

Process Models (Process Framework):

مجموعة من **الطرق والإجرائيات** التي تضبط عمل تطوير المنتج البرمجي حتى يصل إلى منتج برمجي سليم وهي الخطوات ذاتها التي كنا نقوم بها أثناء قيامنا بالمشاريع البرمجية في كليتنا لكن هذه الخطوات هل ترتقي لتعميمها لإنتاج أي منتج برمجي؟ هل من الممكن أن تكون نموذج برمجي **Process Model** ؟

_ ماهي الشروط الواجب توافرها في الـ Process حتى ترقى لأن تصبح Process Model ويمكن اتباعها كنموذج لأي تطوير برمجي ؟

قابلية تقسيم العمل الخاص بتطوير العمل البرمجي إلى مراحل بحيث كل مرحلة تدعى **Checkpoint** أو **Milestone نقطة رجوع** بحيث إن صادفنا مشكلة نستطيع الرجوع إلى المرحلة الأخيرة التي تأكدنا من أنها صحيحة.

واختبار الصحة **Quality Check** يتم من خلال الـOutput الخاص بالمشروع البرمجي, ويسمى المنتج عند الـMilestone بمنتج مرحلي **Work Product** ويكون في المرحلة الأخيرة هو الـProduct الذي نسعى لبرمجته...

وكل منتج مرحلي يحتاج لعدة مهمات مرحلية **Work Tasks** أي يكون المنتج المرحلي قد نفّذ المهمات المرحلية التي وضعناها...

و منه أي نموذج يحقق الشروط الأربعة:

Work Tasks , Work Product , Quality Check , Milestone

يمكن أن يرقى ليكون نموذج Model يتم تعميمه مع كل النماذج المشهورة لتطوير المنتجات البرمجية كـ نموذج الشلال Waterfall Model ونموذج اللولب Spiral Model...

ونستنتج أنه لا يوجد Process model واحدة موحدة لكل منتج وبالتالي كل منتج له Process model خاصة به حسب طبيعة المنتج وسنقوم بدراسة عدة Process Models وهل تنطبق عليهم الشروط السابقة وما فائدة كل Process Model.

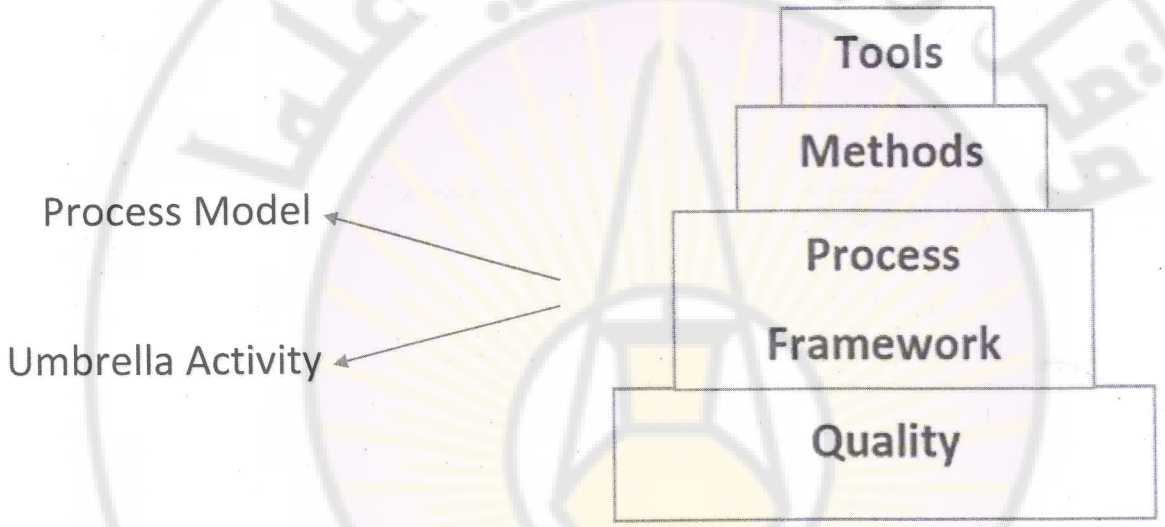
-النهاية-

تذكرة :

الهدف من Software Engineering:

هو تطوير المنتج البرمجي أي الوصول لهذا الـ product ضمن المعايير المطلوبة (الزمن- الجودة-السعر المقبول....) عن طريق تطبيق المفاهيم الهندسية الخاصة في تطوير المنتج البرمجي.

-يتم تمثيل الـSoftware Engineering ضمن المخطط التالي :

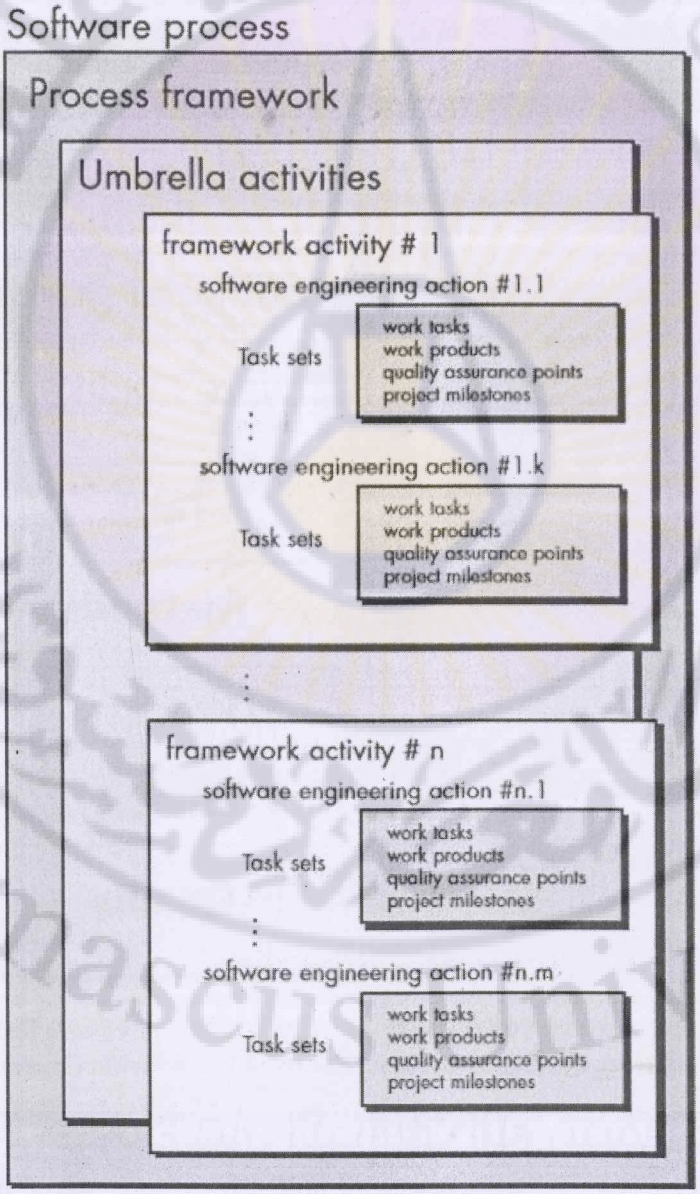


و الـ Process Framework:هي الأجرائيات التي تضمن عند تطبيقها الوصول للمنتج البرمجي.

وتتألف من:

1- Process Model (Framework Activities): وهي النموذج الخاص (المهمات الخاصة) في عملية تطوير المنتج البرمجي والذي تحدثنا عنه في المحاضرة السابقة.

2- Umbrella Activities: مجموعة مهمات مخصصة لإحاطة (حماية) الـ Framework Activities.



Software Process Framework - Figure 2.1 – Page 32 (SE – Practitioner Approach 7th Edition)

Umbrella Activities:

وهي عبارة عن مجموعة من المهام التي تضمن سير العمل الخاص بتطوير المنتج البرمجي بشكل سليم وبدون أخطاء وضمن الوقت المحدد وبخطورة أدنى.

ويمكننا تصنيف بعض الـ Activities الخاصة ب الـ Umbrella Activities (والتي ليس لها علاقة بعملية تطوير المنتج) كالآتي:

- Software project management الإدارة
- Formal technical reviews مراجعة التقارير من قبل مصدر خارجي
- Software quality assurance التأكيد على الجودة
- Software configuration management ادارة الإعدادت البرمجية
- Work product preparation and production. التحضير البدائي والنهائي
- Reusability management. إدارة إعادة الاستخدام
- Measurement. إدارة وقياس التقدم في التطوير البرمجي
- Risk management. إدارة الخطورة

_ ماهي الحماية التي يمكن أن تؤمنها الـUmbrella Activities ؟!!!

بدايةً يجب أن ننوه أن Umbrella Activities ليس لها علاقة بعملية تطوير المنتج البرمجي

إذاً هل تختلف الـUmbrella Activity (Quality Assurance) عن الـ Quality check ؟

نعم لأن عملية الـquality check هي عملية التأكد من سلامة المنتج البرمجي(المرحلي) قبل تسليمه وذلك من قبل فريق التطوير نفسه أما الـUmbrella Activity من خلال Quality Assurance فـهم أشخاص مختلفين تماما" مهمتهم مراجعة المنتج وهم مستقلين عن عملية التطوير ويمكن التطوير من دونهم.

Milestones:

الان سوف نتحدث عن بعض وجهات النظر في الـProcess Framework, التي تتألف من Process Model و Umbrella Activities ونحن نعلم أن الـProcess Model تتألف من عدة Milestones ينتج عنها Work Product منتجات مرحلية يحتاج لمجموعة من الـ Work tasks لإنتاجه.

فما هي الـMilestones التي نعرفها؟

Planning, Analysis, Implantation, Testing, Design, Optimization... etc.

نلاحظ أن هذه الـMilestones تتواجد تقريباً في جميع الـModels لكن بمهام مختلفة إذاً يمكننا أن نضع توصيفاً أكثر تجريداً More Abstract لها:

- **Software Specification (Planning + Analysis + Design)**
- **Software Development (Analysis + Design + Implementation)**
- **Software Validation (Testing)**
- **Software Evolution (Improvement)**

(Introduction to SWE slide 9)

ويوجد توصيفات أخرى (وجهة نظر أخرى) للـ (milestones):

- **Communication and planning** التواصل والتخطيط
- **Software modelling** النمذجة
- **Code generation** توليد الكود إنطلاقاً من نماذج المرحلة السابقة
- **Testing and quality assurance** التأكد من جهازية المنتج

(Introduction to SWE slides 10 + 11)

الفائدة من اتباع أسس ومبادئ هندسة البرمجيات

Software Engineering Benefits:

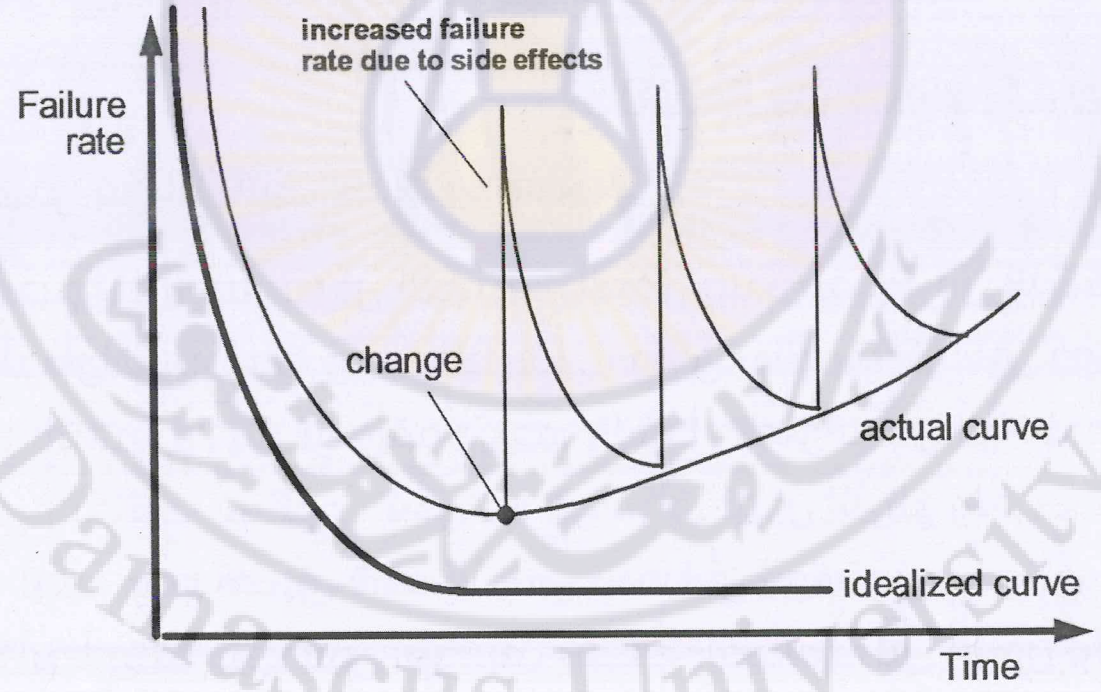
-التخفيض من التكاليف **Reducing Costs** :

من الصحيح نظرياً أننا نخفض التكاليف (العمل بعدد محدود من الأشخاص, توفير الخطوات البرمجية وعدم اتباعها...) ولكن بذلك سنؤثر على جودة المنتج وبالتالي لن يتم تسويقه وبيعه بالشكل المتوقع أي أننا عندما نقوم بتخفيض التكاليف سنقوم بشكل غير مباشر بتخفيض الأرباح ولكن على المدى البعيد ومع تطبيق هذه الأسس نلاحظ أن الوقت سيزيد والجودة ستزداد أيضاً بينما ستنخفض التكاليف (تكلفة الصيانة) على الزبون ولن يضطر لطلب منتج جديد بالكامل في كل مرة يحتاج منتجه إلى التحسين أو إضافة مزايا جديدة.

-التقليل من الأخطاء Reducing Failure :

على الرغم من أن تطبيق هذه الأسس سيزيد من تعقيد نظام المنتج البرمجي إلا أن هذه الأنظمة يجب أن تتمتع بإمكانات متعددة لتواكب التطور الرهيب الحاصل في عالم البرمجيات وبالتالي عند تطبيق هذه الأسس سيتم الحد من الأخطاء.

من السهل نسبياً كتابة برنامج دون استخدام أسس ومبادئ هندسة البرمجيات , كما أن العديد من الشركات لا تستخدم هذه الأسس في منتجاتها وبالتالي منتجات هذه الشركات ستكون أكثر غلاءً وأقل وثوقية مما يجب أن تكون على أرض الواقع.



الصفات (المتطلبات) التي يجب أن يتمتع بها المنتج البرمجي:

جميع المتطلبات التي سنذكرها هي متطلبات غير وظيفية Non-Functional Requirements أي أننا ومن البديهي يجب علينا أن نحققها في كل عملية تطوير لمنتج برمجي ما و هذه المتطلبات هي:

- **Maintainability** (قابلية الصيانة):

أي منتج برمجي يجب أن يكتب بطريقة تمكننا من تعديله فيما بعد وذلك حسب المتطلبات المتغيرة للزبائن , وعلى الرغم من أن قابلية الصيانة لأي منتج برمجي هي أمر مهم وحساس جداً إلا أنه متطلب **حتمي لا** مفر منه خصيصا" في بيئة العمل المتغيرة.

- **Dependability and security** (الاعتمادية و الأمان):

الاعتمادية تشمل العديد من الصفات وهي **reliability, security and safety** فاعتمادنا على منتج معين يعني أن لا يتسبب بأضرار مادية واقتصادية وبشرية في حالة فشله, ويمنع المستخدمين من إلحاق الضرر به.

- **Efficiency** (الكفاءة):

يتم قياسها **بعدد** الأخطاء الموجودة أي أن المنتج البرمجي يجب أن يحوي **أقل** عدد من الأخطاء كما يجب أن يضمن **استجابة النظام ومعالجة الوقت واستخدام الذاكرة**.

- **Acceptability** (القبول):

أي يجب على المنتجات البرمجية أن تكون مقبولة من جميع أنماط المستخدمين المصممة لأجلهم وهذا يعني أنها يجب أن تكون قابلة للفهم و الاستخدام ومتوافقة مع العديدمن الأنظمة.

أنواع المنتجات البرمجية:

في البداية هنالك نمطين من المنتجات البرمجية هما:

- **Generic products:**

أي أن الشركة المنتجة هي التي تنتج المنتج ومتطلباته وتحدد آلية عمله ثم يأتي المستخدم ليطلب هذا المنتج مثل **office.**

- **Customized products:**

أي أن المستخدم هو من يحدد متطلبات المنتج وآلية عمله.

و بناءً على ذلك يكون لدينا عدة أنواع من المنتجات البرمجية:

1) **Stand-alone Applications (desktop applications):**

أي أن هذا التطبيق موجود عند المستخدم ولا يعمل إلا عند المستخدم نفسه ويحوي العمليات الأساسية والضرورية كمان أنه لا يحتاج إلى اتصال بالشبكة (**just one user**).

2) **Interactive transaction-based Applications (web applications):**

يكون لها واجهة على الويب(**Web Interface**) نقوم باستخدامها عن طريق المتصفح (**multi users**).

3) **Embedded control systems:**

هي عبارة عن أنظمة تحكم ضمن البيئة وليست ضمن الـ **virtual machine** كالحاسب أي أنها يجب أن تعمل ضمن ظروف الزمن الحقيقي(Action and Re-action) مثل **coffee machine, auto pilot.**

4) **Batch processing systems:**

هي أنظمة متخصصة في ترتيب أولويات المعالج وتكون جزء من أنظمة أخرى.

5) **Entertainment systems:**

هي أنظمة متخصصة للتسلية كـ **Video Games**.

6) Systems for modeling and simulation:

في بعض الأمور التي تمس حياة الأشخاص أو الظروف الصعب تحقيقها من الصعب القيام بعملية تطوير منتج برمجي لها مثل تدريب الطيارين أو انتاج أنظمة موزعة ولمعالجة هذه المشكلة تم تطوير برامج المحاكاة لتساعدنا في تجريب الأمور التي تمس حياة البشر (Airplane Simulator) أو محاكاة وجود مئات الحاسبات والشبكات المتصلة مع بعضها (Cisco Packet Tracer).

7) Data collection systems:

هي أنظمة مهمتها **فقط** تحصيل البيانات ومعالجتها مثل برامج الفحص الجوي و **Data Base Management System (DBMS).**

8) Systems of Systems:

هي أنظمة (حزم) تحوي بداخلها مجموعة من الـ applications مثل **office** الذي يحوي عند تصيبه **(word-power point-access...).**

Web-Based Applications:

الويب هو منصة لتشغيل التطبيقات والكثير من الشركات والمنظمات تتجه نحو تطوير هذا النوع من المنتجات عوضاً عن المنتجات والبرامج المحلية.

فالـ**Web Application** يقدّم ما يسمى بالـ**Web Service** التي نستطيع الوصول إليها واستخدامها من أي مكان ومن أي نظام تشغيل فقط نحتاج لـ**Browser** واتصال بالأنترنت.

بعض المواصفات للـWeb Application:

- **Software reuse:**

الكثير من هذه المنتجات تتطلب سرعة في التوصيل **Fast Delivery Time** أي الزبون يحتاج للتطبيق بأسرع وقت ممكن ولذلك لدينا الـ**reusability** أي إعادة استخدام ما تم إنتاجه في تطبيقات سابقة.

- **Incremental and agile development:**

سنتحدث عنها في محاضرات قادمة.

- **Service-oriented systems:**

يمكن استخدام خدمات في منتجنا تقدمها **Web Services** أخرى.

- **Rich interfaces:**

امتلاك واجهات جميلة وغنية وسهلة الاستخدام وتتمتع بميزات تغني تجربة المستخدم **User Experience**.

أخلاقيات مهندسي البرمجيات:

- ❖ يجب أن يتبع سلوك إنسان شريف و أخلاقي لكي يتم احترامه كـ إنسان محترف بمجال عمله.
- ❖ يجب أن يتبع سلوك أخلاقي الذي يتضمن مجموعة من المبادئ الصحيحة أخلاقياً أكثر من تمسكه بالقانون.
- ❖ الخصوصية: يجب عليه أن يحترم خصوصية الأشخاص والشركات التي يتعامل معها.

- الكفاءة: أي يجب عليه ألا يبتعد عن مستوى اختصاصه وألا يقبل الأعمال التي يعرف أنه غير قادر على أنجازها بالشكل المطلوب.
- حقوق الملكية الفكرية: أي يجب عليه أن يكون عالماً بالقوانيين الموضوعة لحماية الملكية الفكرية مثل براءة الاختراع و حقوق النشر كما يجب أن يضمن حماية حقوق الملكية الفكرية للمستخدمين.
- سوء الاستخدام: أي يجب عليه ألا يستخدم قدراته ومهاراته في مجال سيء لأن الاستخدام السيء يتراوح من تافه نسبياً (اللعب على الحاسب) إلى خطير للغاية (نشر الفيروسات).

النهاية

نتابع معكم في هذه المحاضرة مع بعض الأمثلة والنماذج لبعض الـ **processes** لكن قبل ذلك يوجد مفهوم يجب أن نضيفه إلى النموذج العام **Generic Process Model** الذي تحدثنا عنه في المحاضرات السابقة وهو مفهوم الـ **Activities Flow** أو تدفق (اتجاه) النشاطات (الإجرائيات) ويدعى أيضاً (**Work Flow, Process Flow**) وهو :

The manner in which the process elements are interrelated to one another. SE-Practitioner Approach 7th Edition p39.

أي هو الوسيلة أو الطريقة التي تتفاعل بها عناصر العملية **Process Elements** مع بعضها البعض, أو بمعنى آخر هو الاتجاه الذي يحدد الانتقال من **Milestone** إلى **Milestone** آخر.

سندرس على مدار محاضرتين مجموعة من الـ**Process Models** والتي يمكن أن تندرج تحت **نمطين وهما:**

- **Agile Processes:**

أي أن هذه الـ**Processes** "رشيقة" أي تدعم التعديل والتغيير المستمر في المتطلبات حسب رغبات الزبون.

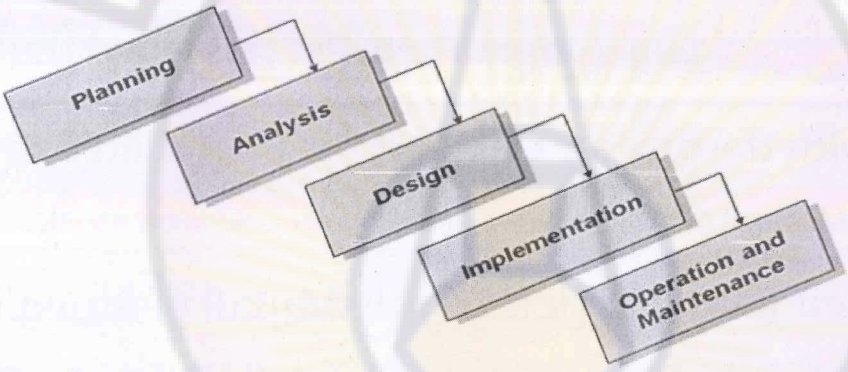
Plan-driven Processes:

أي أن جميع الاطوار في هذه **Processes** مخططة مسبقاً وبالتالي لا نقوم بأي عملية تطوير غير محسوبة ومخططة لها ويطلق عليها أيضا" **Traditional Process Models ومنها:**

❖ Waterfall Process Model (نموذج الشلال):

لدينا هنا نموذجين مقترحين:

النموذج A:



نلاحظ وجود الـ **Milestones** التالية:

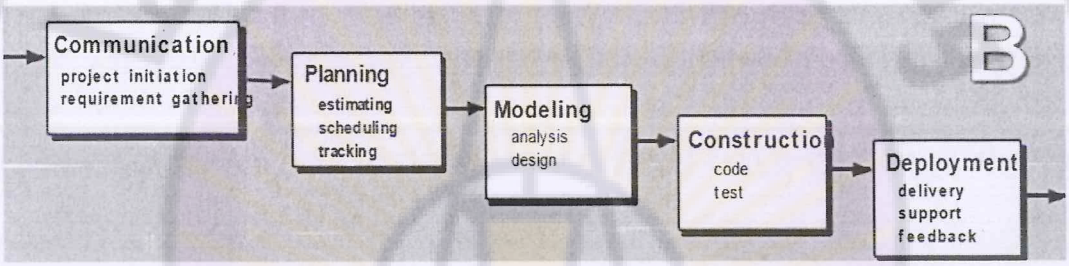
- **Planning**: تحديد الزمن المتوقع والكلفة المتوقعة والتقنيات المستخدمة.
- **Analysis**: وضع الخطوط العريضة (المبدئية) للحل البرمجي ورسم المخططات التحليلية.
- **Design**: وضع المخططات التصميمية (النهائية).
- **Implementation**: عملية تحويل المخططات التصميمية إلى كود.

- Operation and Maintenance: المرحلة مابعد تسليم المنتج البرمجي تشمل (أسئلة المستخدم- صيانة المنتج-.....).

ويكون الـProcess Flow:

بشكل خطي لأنه وبعد عملية الـPlanning تأتي عملية الـAnalysis وبعدها عملية الـDesign ثم الـImplementation وأخيراً مرحلة الـOperation and Maintenance.

النموذج B:



نلاحظ وجود الـmilestones والـwork tasks التالية:

- Communication:
  1) Project Initiation تحليل المشروع.
  2) Requirement Gathering تجميع المتطلبات.
- Planning:
  1) Estimating تقدير المتطلبات (الزمن-الكلفة-....).
  2) Scheduling وضع جداول للأنتاج.
  3) Tracking المتابعة حسب خطة العمل.
- Modeling:
  1) Analysis.
  2) Design.

- **Construction**:
  1) **Code** كتابة الكود للمنتج.
  2) **Test** تجريب هذا الكود والمنتج.
- **Deployment**:
  1) **Delivery**.
  2) **Support**.
  3) **Feedback**.

والـ **Process Flow** هنا يكون خطي أيضاً لأنه وبعد كل مرحلة ينتقل إلى المرحلة التي تليها مباشرة :

Planning→Analysis→Design→Implement→Maintenance

بعد كل مرحلة من هذه المراحل سينتج لدينا منتج مرحلي (**work product**) فبعد مرحلة الـ **Planning** سيكون لدينا مايسمى (**Investigation Report**) وهو تقرير كامل متكامل عن تحقيق الشروط,وبعد الـ**Analysis** سيكون لدينا (**Specifications**) المواصفات الأساسية للمنتج, وبعد الـ**Design** سيكون لدينا (**Design Report**) مجموعة مخططات رئيسية للمنتج وبعد الـ**Implementation** سيكون هو الـ**Application** وبعد الـ**Maintenance** سيكون تقرير صيانة (**Maintenance Report**) عدد المستخدمين-هل المنتج جيد أم ناقص...

<u>نلاحظ من النموذجين التالي:</u>

- كلا النموذجين <u>يحققان شروط **Process model**</u> باحتوائه على عدد من الأطوار **milestones** و المنتجات المرحلية **work products** ومهمات العمل الجزئية **work tasks** و الانتقالات بين الأطوار تكون خطية

process flow مع التسليم بوجود نقاط اختبار جودة بنهاية كل طور من الأطوار.

- النموذج B يختلف عن النموذج A حيث تمت إضافة Phase كامل (Milestone) وهو الـCommunication كما تم دمجAnalysis & Design ضمن Phase واحدة هي Modeling.
- في كلا النموذجين يتم الانتقال من Phase1 إلى Phase2 باتجاه واحد ولا يتم إلا بعد الانتهاء من الـwork tasks وإنتاج Work product للـPhase1 وبعد هذا الانتقال لا نستطيع العودة إلى الـphase1, فهذه النماذج تحمل صفة الشلال الذي يسقط من مكان مرتفع إلى مكان منخفض (من هنا جاءت التسمية).

ومنه كلا النموذجين هو Waterfall Model لكن من هو الأفضل؟

نلاحظ تخصيص طور كامل لمرحلة التواصل مع الزبون في النموذج B لضمان فهم واضح وكامل لمتطلبات الزبون واعطائه الفرصة الكاملة للتعبير والتغيير ما دمنا ضمن هذا الطور.

إيجابيات وسلبيات استخدام الـWaterfall Process Model:

Advantages and Disadvantages of Waterfall:

- **السلبيات Disadvantages :**
  1) غير قابل لتقبل متطلبات الزبائن المتغيرة.
  2) عدم القدرة على الانتقال من مرحلة إلى أخرى إلا عند الانتهاء من المرحلة السابقة (اتجاه واحد).
- **الإيجابيات Advantages:**

1) المحافظة على النقاط الرئيسية في عملية التطوير.
2) التسلسل الخطي للـ**Flow Activities**.
3) تحكم عالي.
4) بنية وخطة واضحة **Plan-driven**.
5) يتم تسليم تقييم المنتج في نهاية عملية التطوير والاختبار.
6) زمن التنفيذ سريع نسبياً.

استخدام الـ Waterfall Model:

When to use Waterfall Model:

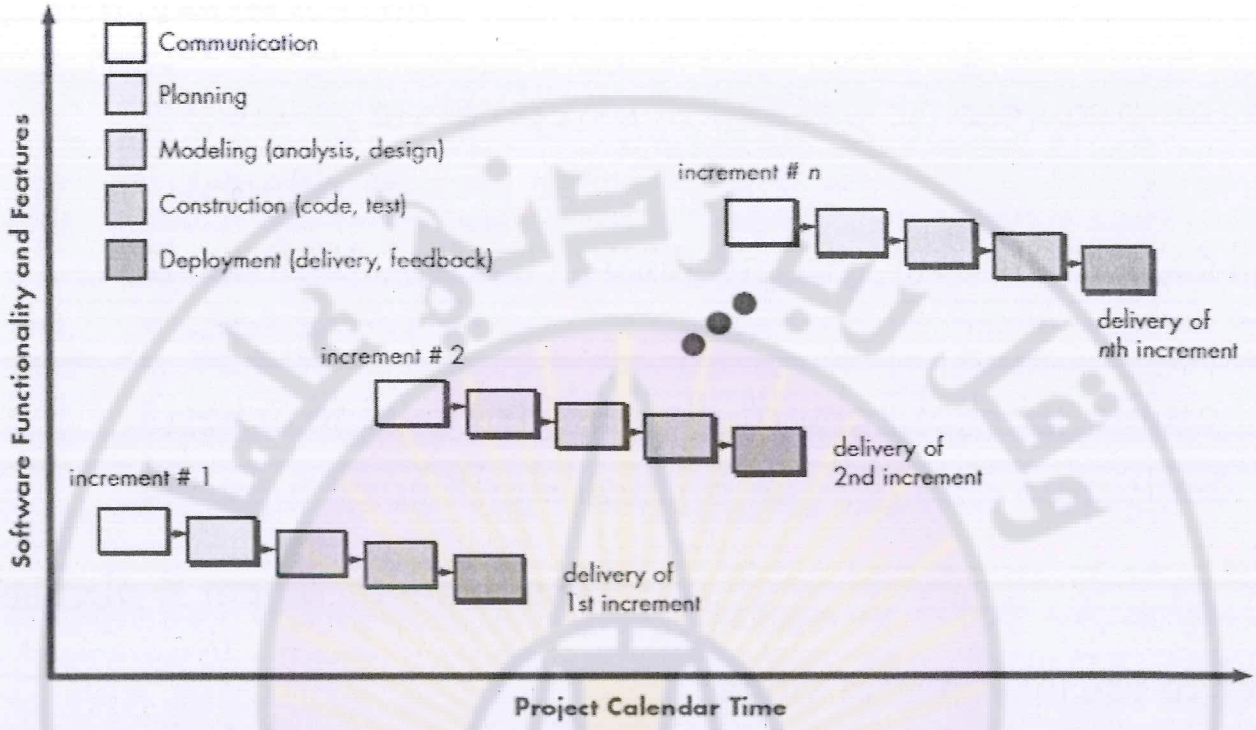
1) عندما تكون المشاريع صغيرة.
2) المتطلبات مفهومة و واضحة و ثابتة.
3) التقنيات المستخدمة مفهومة وليست ديناميكية.
4) التغييرات محدودة إلى حد ما أثناء عملية التطوير.

يوجد تمثيل آخر لنموذج الـWaterfall يدعى **V-model** يمكنكم الاطلاع عليه في الصفحات 39 - 40 من SE: Practitioner Approach 7th edition. وهو نموذج مخصص لتوضيح مراحل وأنواع الاختبار المختلفة وأماكن تصحيح الخطأ إن وجد.

❖ Incremental Process Model: النموذج التزايدي

انطلاقاً من أكبر مشكلة واجهناها عند استخدام نموذج الـ**Waterfall** وهي عدم قابلية التعديل والصيانة المتأخرة تم إنشاء ما يسمى **Incremental Process model** وفيه يتم تقسيم متطلبات المشروع **requirements** إلى **عدة تزايدات** واعتبار كل تزايد مشروع أو منتج

مستقل وبعد الانتهاء منه نقوم بالانتقال لتحضير وتنفيذ التزايد التالي وهكذا...



*يمكن استخدام Waterfall model في انتاج كل increment.

الذي ينتج لدينا يسمى **increment** وهو ليس منتج كامل وإنما جزء منه وبالتالي يمكننا تسليم هذا الجزء للمستخدم بخدمات محدودة ليجربه ويقوم بطلب التعديلات (في حال وجودها) وعلى التوازي يمكننا التفكير بعملية تطوير الجزء الثاني من هذا المنتج...

الإيجابيات Advantages:

1) تقليل كلفة متطلبات الزبائن المتغيرة.
2) من السهل إعطاء المستخدم معلومات عن الجزء الذي تم تطويره وبالتالي المستخدم يمكنه أن يعطي رأيه بالعمل المنجز حتى الآن ويرى كيفية التنفيذ.

3) يمكن للمستخدم أن يستفيد من الجزء الذي تم تطويره ويقوم باستخدامه.

السلبيات Disadvantages :

1) ليست كل مرحلة (جزء) تم إنجازها تعبر عن مدى تقدم عملية تطويرالمنتج.
2) بنية النظام يمكنها أن تنهار في حال الإضافات الجديدة الغير مناسبة.
3) إضاعة الوقت والمال في حال تم تطوير أجزاء غير مناسبة.
4) صعوبة تجميع الأجزاء المطورة في حال كانت غير مترابطة.

استخدام الـ Incremental Model :

When to use Incremental Model:

1) في حالة التغيرات المتكررة وليس الإضافات.
2) في حال المشاريع متوسطة الحجم.
3) عندما يكون المستخدم متشوق لرؤية عملية تطوير المنتج أي عندما يكون مطلوب تسليم جزء من المنتج بسرعة.

❖ Prototyping Model: نموذج النمذجة الأولية

يتم وضع نموذج أولي يحوي الخدمات الأساسية ولايتم التركيز إلا على المتطلبات الوظيفية **Functional Requirements** وتهمل الـ **Non-Functional Requirements** وهو تجريبي والفائدة منه إعطاء المستخدم نموذج أولي (مبدئي) عن المشروع حتى يتم فهم المتطلبات بشكل أفضل بالنسبة للزبون وللمطورين.

ويطلق على هذا النموذج الأولي Core Product.

الإيجابيات Advantages:

1) أقرب (واقعياً) إلى متطلبات المستخدم وبالتالي يمكننا من تجنب الجهد المبذول في عملية التطوير.
2) يحسن جودة التصميم و الاستخدام النظام وقابلية الصيانة لاحقاً.
3) يقلل من جهد تطوير المنتج لاحقاً.

السلبيات Disadvantages :

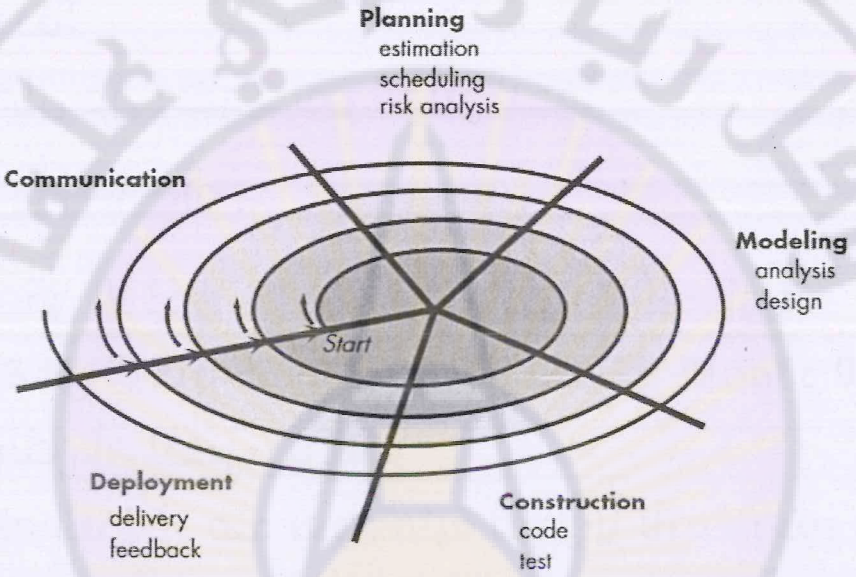
1) يمكن ألا تكون ضمن معايير الجودة المطلوبة.
2) تهمل الاختبارات والتوثيق أثناء تحضير النموذج الاولي.
3) يمكن أن تتغير بسرعة.
4) يتم التخلص منه بعد عملية التطوير لأنه لايعتبر جيداً كـمنتج برمجي, أي الوقت المبذل عليه يعتبر مكلفاً.

استخدام الـ Prototype Model :

1) عندما تكون المتطلبات غير واضحة أو غير محددة.
2) عندما يكون مطلوب نسخة سريعة عن المشروع ككل.
3) لشرح خيارات التصميم.
4) للمساعدة في عملية التحقق من متطلبات المنتج قبل البدء بعملية تطويره.
5) للقيام بعملية الاختبار في وقت مسبق.

❖ The Spiral Model: نموذج اللولب

يعتبر هذا النموذج من النماذج المتزايدة **Incremental** وتعتمد على مبدأ الـ**Slices** ولكن بطريقة تزايدية وهذا النموذج **Risk-Based Process Model** أي أننا نعتمد على هذا النموذج في تطوير البرمجيات ذات الخطورة العالية.



ما هو أول شيء نبدأ بدراسته (تطويره) عند استخدام نموذج الـSpiral ؟

معالجة (تطوير) المتطلبات المعرضة للفشل بشكل أكبر لأننا سنعالجها أكثر من مرة عند كل **Iteration** .

الإيجابيات Advantages:

1) تحليل كمية كبيرة من الخطر ولذلك سيتم تجاهل الخطر بشكل ملحوظ.

2) جيدة ومثالية في حالة المشاريع العالية الخطورة وذات المتطلبات المتغيرة.
3) إمكانية إضافة وظائف إضافية في وقت لاحق.
4) جزء من المنتج يتم إنتاجه في وقت مبكر من حياة المنتج.

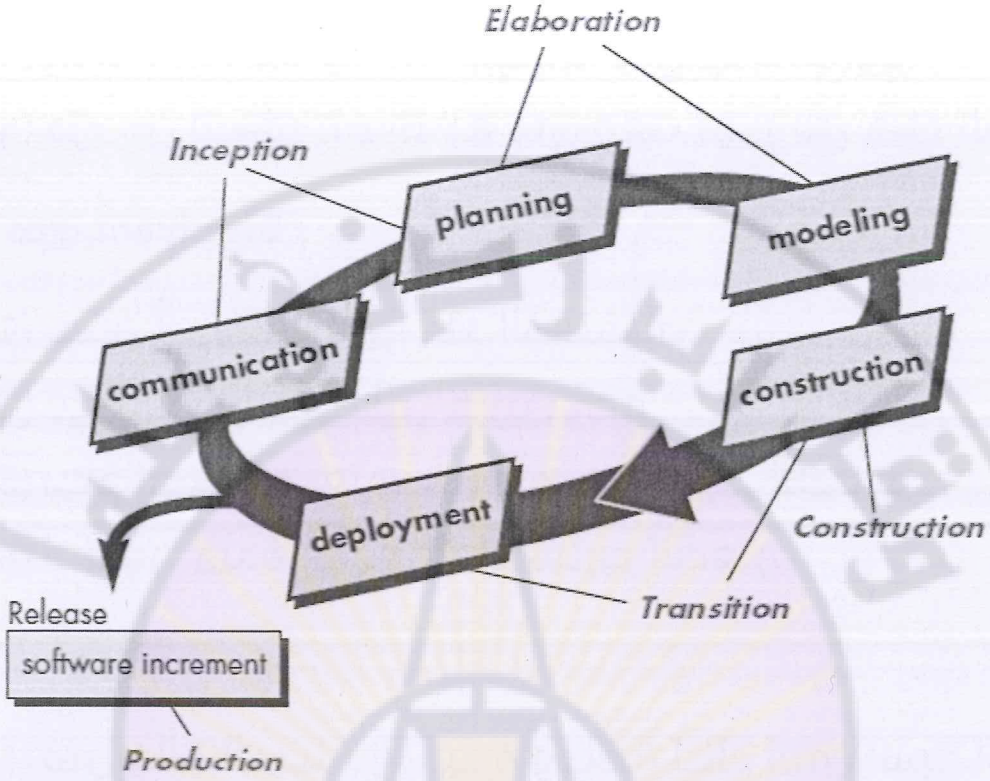
السلبيات Disadvantages :

1) يمكن أن تكلف كثيراً عند الاستخدام.
2) تحليل المخاطر يتطلب خبرة خاصة.
3) نجاح المشروع يعتمد على مرحلة تحليل المخاطر.
4) لا تكون جيدة في حالة المشاريع الصغيرة.

استخدام الـ Spiral Model :

1) في حالة المشاريع الكبيرة وخاصة لتلك التي نسبة الفشل فيها عالية.
2) عندما تكون عملية التحقق من المخاطر والكلفة مهمة.
3) في حالة المشاريع المتوسطة إلى العالية الخطورة.
4) الالتزام بالمشاريع طويلة الأمد غير جديرة بالثقة بسبب التغيرات المتوقعة للأولويات الاقتصادية.
5) المستخدمون غير متأكدون من حاجاتهم.
6) عندما تكون المتطلبات معقدة.
7) في حال وجود خط انتاج جديد.
8) التغيرات العلمية المتوقعة (كالبحوث والاستكشافات).

❖ النموذج الموحد :The Unified Process Model



تم توحيد جميع مزايا الـProcess Model السابقة وجمعها في Process Model واحدة هي الـ Unified Model التي تعتمد على الـUML وهدفها هو إنتاج منتج بجودة عالية.

يعتمد الـUP على الـIncremental product فهو تزايدي incremental ودوري iterative.

أساس الدورية في الـUnified Model هو الـworkflows والتي هي على التمثيل:

(Planning-Analysis-Design-Testing-Support)

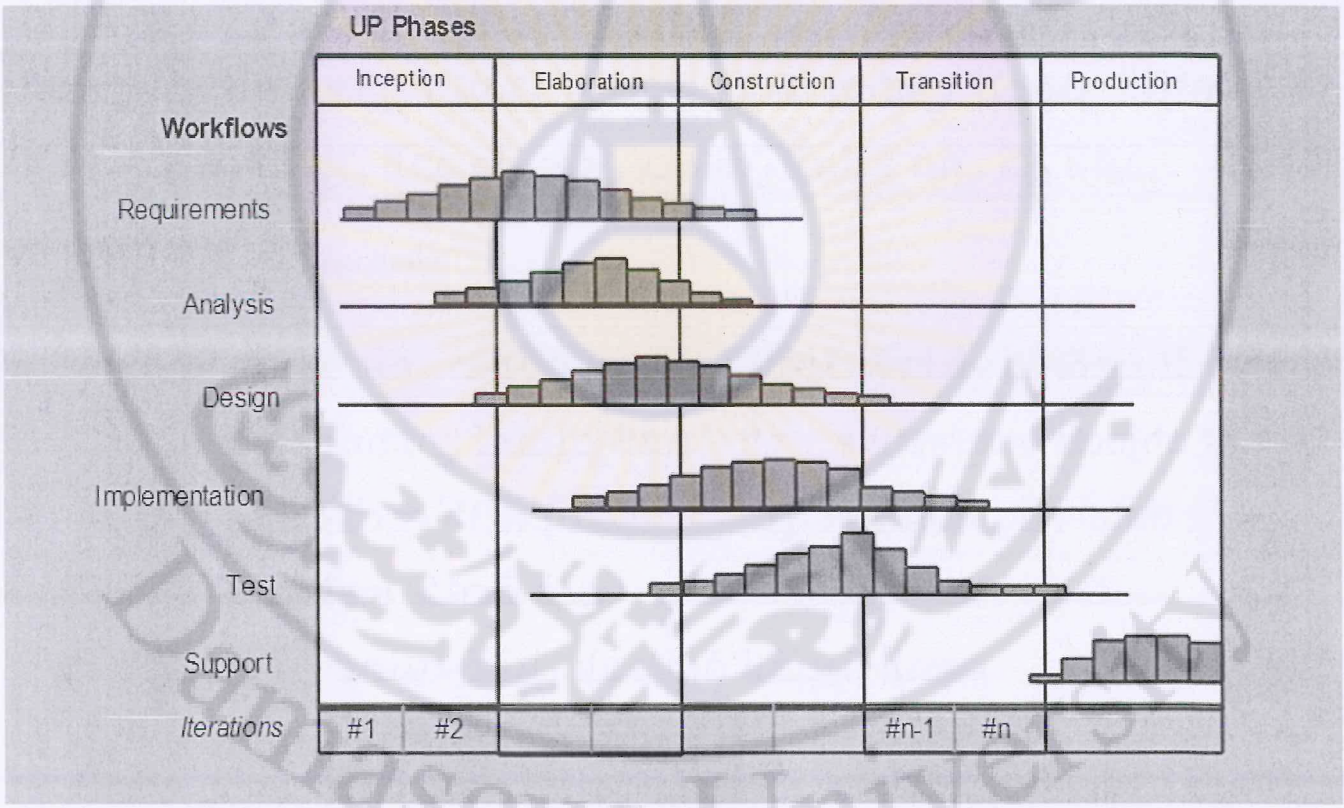
وحتى ننتهي من **milestone** واحدة يمكننا أن نقوم بالدوران على الـ **workflows** عدة مرات في كل طور من أطوار نموذج الاجرائيات الموحد **milestones** والتي هي:

Inception, Elaboration, Construction and Transition

الطور البدائي, الطور التوضيحي, الطور البنائي و الطور التوظيفي.

بحيث تحدد عدد الدورات بحسب حجم المشروع.

والسؤال المهم هنا ما العلاقة الرابطة بين UP work flows وبين UP (Milestones) phases ؟



نلاحظ من الجدول أنه على الرغم من أن كل طور يتضمن عدة دورات من **workflows** ليتم إنجازه ولكن في الحقيقة بعض الاعمال المتضمنة في

workflows لا يتم استدعاءها أو إنجازها حسب الطور الموجودة ضمنه.
فعلى سبيل المثال يحتاج طور inception لدورتين كاملتين من workflows ليتم إنجازه ولكن مراحل العمل المنجزة في كل دورة هي فقط requirements والقليل من الـanalysis والـdesign والباقي يهمل...

ومثلاً عند الـelaboration نقوم بالـrequirements ونكثر من analysis والـdesign ونبدأ بالـ implementation و الـtesting...

أي تأثير الـWorkflow نسبي على الـMilestones.

وبالتالي نحن هنا بحاجة الى عدة دورات قبل الحصول على احد الإصدارات التي يمكن للزبون الحصول عليها بعكس النموذج التزايدي الذي يضمن للزبون الحصول على جزء من المنتج في كل تزايد.

الإيجابيات Advantages:

1) يعتمد على طريقة تصميم وتنفيذ وتحليل مفهوم الـ object oriented أي تقسيم العمل إلى set of objects interactive with each other بمعنى أن كل object يحوي data و methods خاصة فيه.
2) يدعم عملية تحليل الخطورة الكبيرة الحجم.
3) يضمن النتائج بجودة عالية.
4) استخدام فعال للموارد المتاحة.
5) القدرة على اكتشاف القضايا في وقت باكر من المشروع وبالتالي يمكن تسليم أجزاء من المشروع للمستخدم بانتظام أي يبقى أصحاب المصحلة على تواصل دائم.

6) يمكن أن تتغير لاستيعاب الحالات المختلفة.

السلبيات Disadvantages :

1) تكلفة عالية و زمن كبير لأنها عملية ضخمة.
2) لايمكن أن ينفذ في حالة المشاريع الصغير والغير **object oriented**.
3) يمكن أن تكون عملية معقدة جداً.
4) عدم القدرة على التحكم بعملية التطوير.
5) تحتاج إلى خبرة جيدة للتعامل معها.

استخدام الـ Unified Model:

1) في حالة المشاريع الكبيرة.
2) مناسبة في حالة المشاريع التي تدعم OOP.
3) نستطيع استخدامه في جميع أنواع المشاريع.

(embedded-web development-desktop application...)

- النهاية -

السلام عليكم, تحدثنا في المحاضرة السابقة عن العديد من الـ process models منها:

Waterfall – Incremental – Prototype – Spiral – Unified

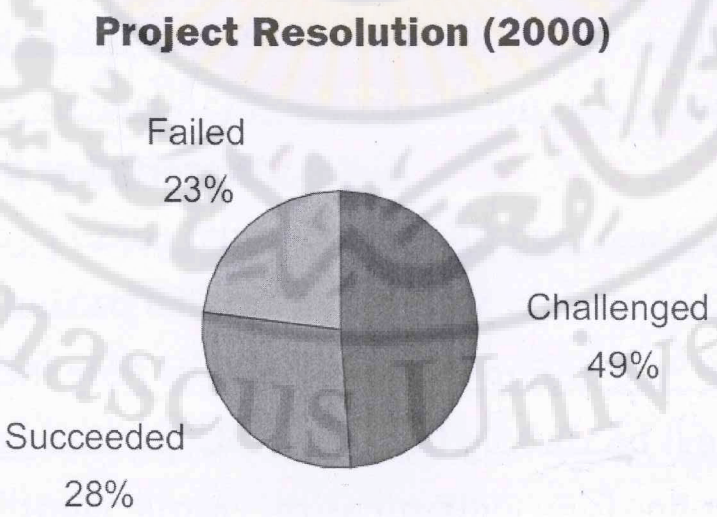
وكلاً منها له استخدامه ولا يوجد جواب محدد ومطلق حول استخدام كل نوع منها فكل نموذج يتمتع بإيجابيات وسلبيات ولكن يمكن القول بأنه يوجد حالات معينة ينصح باستخدام نموذج بدلاً من الآخر ومنها:

- إذا كانت المتطلبات غير واضحة ومبهمة تماماً يمكن استخدام نموذج الـ **Prototype**
  لوضع تصور أولي عن المنتج المراد تنفيذه ولمساعدة الزبون في تحديد المتطلبات التي يريدها.
- إذا كان المشروع كبير يمكن استخدام **Unified** (كما يمكن استخدام **Spiral**).
- إذا كان الفريق متوسط الحجم أو فريق التطوير مبتدئ يمكن استخدام **Waterfall or Incremental**.
- إذا كان المشروع يتطلب تطوير المنتج بأسرع وقت ممكن وأراد المستخدم المنتج كاملاً نستخدم الـ**Waterfall** وإذا أراد جزءاً منه نستخدم الـ**Incremental** ولكن اذا كانت المتطلبات مبهمة وغير مكتملة مع الحاجة للتطوير بسرعة نستخدم الـ**Incremental** ولا نستخدم الـ**Prototype** لأنه بعد الانتهاء من النموذج الأولي للمنتج المطلوب يهمل ولا يرقى ليكون منتج نهائي وعلينا البدء بتطوير المنتج من الصفر بعد تحديد المتطلبات وذلك يعتبر مكلفاً بالنسبة للوقت المحدد .
- إذا كان المشروع ذو خطورة عالية يمكن استخدام **Spiral**.

انتهت المحاضرة السابقة عند الـUnified Process والتي تعتبر أكثر الـmodels رسمية وتستخدم في حالة المشاريع الكبيرة والمتوسطة وهي تتألف من عدد من الأطوار وكل طور نحتاج إلى عدد من الدورات على الـ workflow لإنجاز طور واحد من أطوار الـUP(والتي من الممكن أن تكون غير مستخدمة) كما أن دورة واحدة على الأطوار لا تضمن إنتاج منتج نهائي, أي من الممكن أن نحتاج لدورات أُخرى على الأطوار.

مقدمة:

كل هذه التحسينات والأفكار التي ضيفت على الـPlan-Driven Process Model من Incremental Development و Fast Delivery لم تستطع مواكبة متطلب الـ Rapid Development and Delivery (التطوير والإنتاج السريع) فلم تستطع الاستجابة للأسواق الجديدة و الحالات الاقتصادية المتغيرة وظهور العديد من المنتجات والخدمات المنافسة, فحسب تقرير لـStandish Group (تهتم في دراسة عوامل فشل المشاريع البرمجية وتسمي تقاريرها CHAOS reports) لعام 2000 كانت نسب نجاح وفشل المشاريع البرمجية كالتالي:



حيث Challenged هي نسبة المشاريع التي تخطت التكلفة المالية للمشروع Over-Budget والإطار الزمني المتوقع Over-Time estimation وبالقليل من المتطلبات والمزايا المطلوبة Fewer Features and Functions وكان 82% من هذه المشاريع العامل في فشلها هو Waterfall-style scope management أو بمعنى اخر Plan-Driven Process Model...

حيث كان الفكر السائد هو أن الطرق التي تؤدي إلى المنتجات البرمجية الناجحة هي التخطيط الحذر Careful Project Planning وضمان الجودة الرسمية Formalized Quality Assurance واستخدام الوثائق الرسمية حتى عند الانتقال من Framework Activity إلى أخرى.

كل هذه الأفكار بلورها مجتمع هندسة البرمجيات الذي كان مسؤول عن تطوير برمجيات ضخمة مثل أنظمة المركبات الفضائية والأنظمة الحكومية (السائدة في تلك الفترة)

وكانت هذه البرمجيات تحتاج لزمن تطوير كبير (10 سنوات أو أكثر) والتغيير فيها مكلف فكانت عملية التخطيط والتصميم واستخدام الوثائق بشكل كبير أمراً لا بد منه...

أما مع بدايات 1990 وبدء ظهور البرمجيات الصغيرة والمتوسطة وظهور الحاجة للتطوير السريع (المنافسة) كان لابد من طرق جديدة تواكب الطلب فبدأت فلسفة وطراق الـAgile بالظهور والتبلور حتى عام 2001 حيث وضع التعريف الرسمي للـAgile development.

الـ Agile Development :

Agile تعني الرشاقة والرشاقة في علم هندسة البرمجيات تعني السرعة في التطوير ويتم ذلك بتخفيف بعض النشاطات التي كانت تمارس في الـ Plan-Driven Approach والتي كانت سبب في عملية بطء التطوير البرمجي وزيادة

البيروقراطية في العمل البرمجي لكن كما نعلم أن السرعة تتناسب عكساً مع عملية الجودة.

ومن هنا يبدأ مفهوم الـAgile وهو ضبط عملية التوازن بين السرعة وضمان الجودة.

The optimum proportion between speed and quality.

وبالتالي يجب علينا تحديد المراحل والنشاطات **الأقل أهمية** في عملية تطوير المنتج البرمجي بحيث إذا قمنا بإهمالها او التخفيف منها لا يكون تأثيرها كبير.

وهنا يجب التنبيه إلى الاختلاف الكبير بين مفهومي **التخفيف** و**التخلص أو الإزالة** فلا نقوم بالتخلص مثلاً من الـTesting وإنما يمكننا تخفيفها إلى حد ما أو عدم اعتبارها نشاط قائم بحد ذاتها.

المراحل والمهام التي يمكننا إزالتها:

← **Analysis Phase:** لأنه في مرحلة الـDesign يمكن أن نقوم بعملية الـ Analysis أيضاً.

← **Documentations:** التقارير والوثائق التي يتم تناقلها بين فرق التطوير بين المراحل والنشاطات والتي هي صلة التواصل الرسمية التي يمكن أن يعتمد عليها فريق التطوير.

ومنه نستنتج أن هدف الـ Agile:

هو تحقيق السرعة عن طريق تخفيف بعض الأعمال والنشاطات الروتينية أثناء تصميم المنتج البرمجي.

خصائص وشروط الـ Agile Development:

- الاعتماد على فريق خبير يعتمد على وسائل تواصل مباشر ذات ثقة وخبرة عالية فالـAgile تعتمد وتهتم بالأشخاص أكثر من التخطيط والتعامل الرسمي.

- الحصول على منتج قابل للاستخدام بفترة زمنية قصيرة.
- المراحل تكاد تكون متداخلة أي أنه لا يوجد حدود فاصلة بين المرحلة والأخرى.
- إمكانية التعديل في أي مرحلة من العمل حتى لو كان في الـIncrement الواحدة.
- الزبون (أو من يمثله) هو عضو أساسي من أعضاء فريق التطوير,فالـAgile تهدف لإزالة الحواجز الرسمية بين الزبون والمطورين لتحقيق أكبر قدر من التعاون والسرعة في تحديد المتطلبات والتغييرات.

ومنه الـAgile:

هي عملية تطوير المنتج البرمجي بسرعة بحيث نستطيع تطوير أو إنتاج عدة مراحل versions/increments بسرعة عالية وتتطلب تنفيذ مجموعة من الفلسفات والأدوات تتمحور حول **تحقيق تواصل فعال بشكل غير رسمي بين أعضاء فريق التطوير وتفضيل الوصول لمنتج على الوثائق والأوراق والاعتماد على جعل الزبون عضو من فريق التطوير عوضاً عن الرسمية والمعاملات والاستجابة السريعة للتغيرات بدلاً من اتباع الخطط.**

In 2001, Kent Beck and 16 other noted software developers, writers, and consultants [Bec01a] (referred to as the "Agile Alliance") signed the "**Manifesto for Agile Software Development.**" It stated:

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

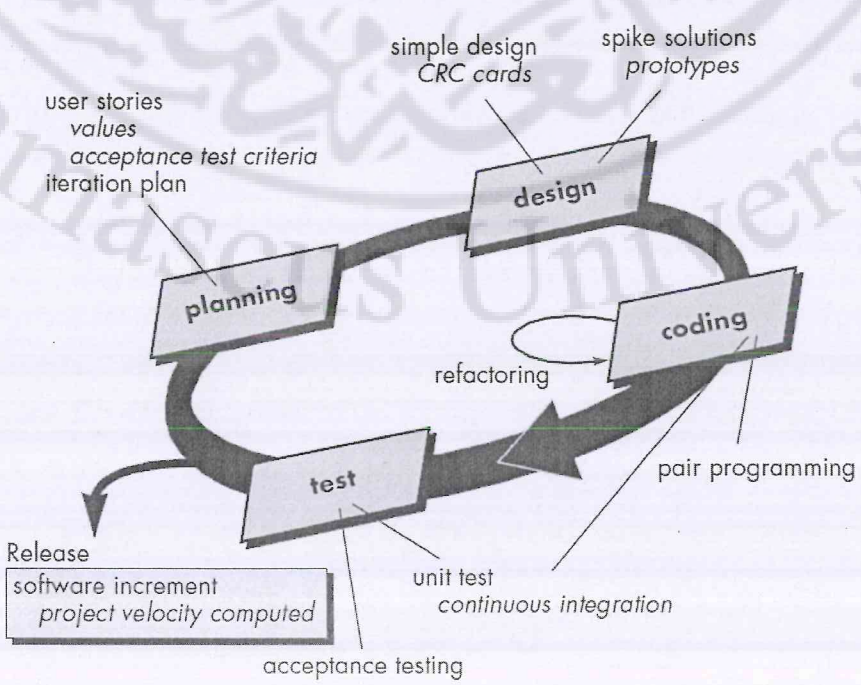
- Individuals and interactions over processes and tools.
- Working software over comprehensive documentation.
- Customer collaboration over contract negotiation.
- Responding to change over following a plan.

:Agile Approach and Plan-Driven Approach

Property	Agile	Plan-Driven
Project Size	Small to Medium	Small to Very Large
Flow	Incremental, Iterative	Linear, iterative, incremental
Customer Commitment	always	The start of project or increment
Develop new Product	Can be used.	Can be used.
Maintain Existing product	Can't be used, there is no documentation.	Can be used.
Speed	Highest	Slow to High
Respond to changes	Best response	From no response (Waterfall) to good response.
Documentation	No documentation	Good documentation
Team size	small	Small to large
Very detailed Phases	Interleaved	Detailed

بعض النماذج والطرائق في الـAgile Development :

:(XP)Extreme Programming -1



النشاطات التي يمر بها الـXP:

Planning:

في الـXP لايوجد حد فاصل واضح بين الـAnalysis و الـDesign (الـActivities التي تعرفنا عليها في النماذج العامّة) فالـAnalysis غير موجود بل متداخل مع الـPlanning, ووجود الـFlow الدائري يعطي انطباع أن عملية الـXP عملية تزايدية Incremental أي تعتمد على المنتجات الـIncremental عند الانتهاء من كل دورة والـrequirements سيتم توزيعها على increments .

ويطلق على المتطلبات في عملية الـXP اسم User Story مجموعة من القصص او السيناريوهات Scenarios للـوظائف التي يحتاجها الزبون من المنتج والتي يستطيع من خلالها فريق التطوير من فهم السياق العام والوظيفة الأساسية للمنتج المراد تطويره.

ويقوم فريق التطوير بتقسيم كل story الي من مجموعة من النشاطات activities أو المهام Tasks ثم التخطيط للـIncrement القادم بالتعاون مع الزبون وتحديد الـStories ذات الأولوية الأعلى والبدء بها ولايمكن تحديد تاريخ تسليم لهذا الـIncrement فالـagility تعتمد على جمع المعلومات الجديدة مع التقدم في المشروع Explore on the road .

وبعد الانتهاء من الـIncrement الأول يتم تحديد سرعة المشروع Project Velocity:

> **Project velocity is the number of customer stories implemented during the first release.**

والتي يمكن الاستفادة منها في جدولة مواعيد الإصدارات القادمة

> **Helps estimate delivery dates and schedule for subsequent releases (Increments).**

Design:

يقوم فريق التطوير بتصور كامل عن الحل لكل story عن طريق مايسمى CRC cards التي تساعد في عملية التفكير بالمنتج بمفهوم غرضي التوجه

Object-Oriented مع مراعاة مبدأ KIS (Keep It Simple) أي التصميمات البسيطة مفضلة على التصميمات المعقدة.

نلاحظ ان عملية الـXP لا تحوي Quality Assurance أو Quality Check وهذا يتم ضمنيا عن طريق الـtesting واستخدام الـTest unit حيث تعتبر وسيلة الضمان الوحيدة لسلامة المنتج وجودته ولذلك الـ Agile Development يعتمد على Heavy Intensive Testing الاختبارات المركزة...

الـTest unit تتألف من Test cases قد تم وضعها اثناء تصميم الـCRC (مرحلة الـDesign) لكل story وهي عبارة من مجموعة من الـoutputs المتوقعة من تنفيذ كود الـCRC.

عندما يكون الحل مبهم للـUser Story أي يحتاج لتقنية جديدة على الفريق مثلاً(على عكس مانقوم به في الـCRC حين تصميمها فنحن نعلم المشكلة ولدينا تصور كامل للحل) سيكون لدينا حالة شاذة ونقوم بشئ يطلق عليه Spike Solution وهو عبارة عن Prototype نقوم بتطويره بسرعة ويحقق النقاط الرئيسية الأولية للحل نستطيع من خلاله تقليل خطورة الفشل في توصيف المشكلة أثناء تنفيذ البرمجة الفعلية للـstory والتحقق من استيعاب الفريق للـstory على أفضل وجه ممكن...

Coding:

يتم استخدام فريق صغير نسبياً لتكويد عدة CRC معا (شخصين مثلاً) و توزيع فرق أخرى على CRC أخرى وذلك لتحقيق السرعة في الـAgile development ويمكن تطبيق مايسمى Pair Programming في الفريق الواحد فشخص يقوم بالبرمجة والاخر يقوم بمتابعته ومراجعة الكود البرمجي أثناء كتابته و يتم التبديل بينهما من حين إلى آخر...

وعند الحاجة لتعديل أو تحسين الحل لتصميم معين في مراحل متقدمة نقوم بـ Refactoring (إعادة تموضع):

Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves the internal structure. It is a disciplined way to clean up code [and modify/simplify the internal design] that minimizes the chances of introducing bugs. In essence, when you refactor you are improving the design of the code after it has been written.

فـعملية **Refactoring** لاتقوم بتغيير الويفة أو تسلسل العمليات التي يقوم بها الكود البرمجي بل تقوم بتحسين البنية الداخلية للكود وتنيفه لتقليل حالات هور الأخطاء البرمجية عند التنفيذ,فـهدف الـ**Refactoring** هو تحسين تصميم الكود البرمجي الذي تمت كتابته مسبقاً.

الهدف الرئيسي لـ**Refactoring** في عملية الـ**XP** هو التحكم بالتعديلات التي نقوم بها مع تقدّم عملية إنشاء المنتج عن طريق التغييرات البسيطة التي نضيفها على التصميم (التي بدورها تحسن بشكل كبير منه), فالـ**XP** تعتمد على القليل من المنتجات المرحلية (عدا الـ**CRC** و الـ**Spike solutions**) والتصميم يعتبرأداة عابرة في عملية التطوير المنتج ويجب أن يكون تحت التعديل باستمرار...

Testing:

بعد الانتهاء من تطوير الكود البرمجي لـ**story** الواحدة نقوم باستخدام الـ **Test Unit** أو الـ**test cases** التي وضعناها اثناء تصميم الـ**CRC** للتاكد من جودة وجهازية الحل

"Fixing small problems every few hours takes less time than fixing huge problems just before the deadline".

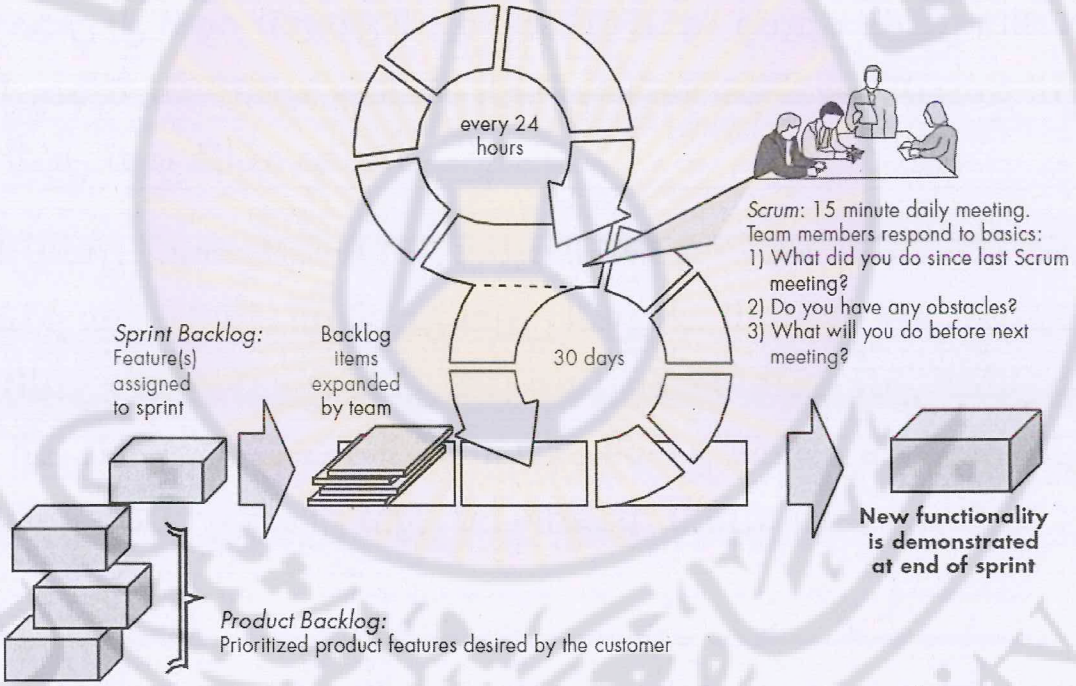
وعند تقديم الـ**increment** للزبون يوجد اختبارات تدعى **Acceptance Tests** أو **Customer Tests** وهي احتبارات معرفة من قبل الزبون يقوم بتعريفها على المنتج بشكل عام وماهو ظاهر وقابل لمراجعته من قبله وعند نجاح الـ**increment** بها

يكون الـ**increment** قد قبله الزبون **Accepted**, يمكن أن نطلق على الـ**increment** قبل قبوله بـ**Beta-Increment**.

بعد إصدار هذا الـ**Increment** نعيد الكرّة ونختار **Stories** جديدة ونخطط للـ**Increment** الجديد وهكذا حتى الوصول للـ**Increment** النهائي الذي يمثل المنتج المطلوب.

SCRUM-2:

أتت الـ**Scrum** لتحل مشكلة أساسية في الـ**XP** فكما علمنا أن الـزبون يشكل جزء من فريق تطوير المنتج مما سيؤدي لوجود توتر **Stress** و تشويش **Interfering** من قبله أثناء العمل إلى المنتج مما يسبب حالة عدم ارتياح في عملية التطوير للفريق البرمجي...



تعتمد الـ**Scrum** على شخص يدعى **Scrum Master** وهو:

1- يكون عقدة التواصل بين الزبون والفريق المطوّر.

2- يضمن عملية التفاعل بين الـ**Sub-teams** بالفريق الواحد.

3- يتأكد من سير عملية تطوير المتطلبات بشكل سليم.

يعتبر الـ**Scrum Master** مسيّر **Facilitator** ولا يعتبر مدير **Manager**.

يقوم الـScrum Master بجمع المتطلبات من الزبون والتي يطلق عليها في هذا النموذج Product Backlog والتي تعتبر قامة To-Do على فريق التطوير القيام بها والتي يمكن أن تكون:

Features, Requirements, User stories and Descriptions

عملية تطوير عنصر واحد من عناصر الـBacklog يطلق عليها اسم Sprint (ينجز ضمن إطار زمني 30 يوم مثلاً Scrum Cycle) وخلال هذا الإطار الزمني يتم تحقيق اجتماع يومي (Scrum Meeting) لمدة 15 دقيقة بين فريق التطوير والـ Scrum Master حتى تنفيذ الـ Increment المطلوب والذي يطلق عليه Shippable Product Increment قابل للشحن و الاستخدام .

ثم يقوم الفريق باختيار عنصر اخر من الـBacklog والبدء بـSprint جديدة وإنتاج Increment جديد وهكذا...

The name is derived from an activity that occurs during a rugby match where a group of players forms around the ball and the teammates work together (sometimes violently!) to move the ball downfield.

تحدّث الدكتور في بداية المحاضرة عن استخدام الـAgile development في عملية الـMaintaining الصيانة للمنتجات البرمجية حيث لاينصح (لكن يمكن) استخدام الـAgile Methods في عملية الصيانة فإذا كانت هذه المنتجات الأخيرة قد تم تطويرها باستخدام الـAgile سنحتاج لعمليات هندسة عكسية **Reverse Engineering** حتى نستطيع استنباط الوظائف والمتطلبات التي تقوم بها هذه المنتجات لايجاد العطل وإمكانية التغيير فيه, أما اذا كانت قد طورت باستخدام الطرق الكلاسيكية فستقوم بهدم البنية **Structure** الخاصة بهذه المنتجات من **Documentations, Models...Etc.** وستتحول لمنتجات غير قابلة للصيانة فيما بعد.

Requirements Engineering

بداية ماهو تعريف المتطلبات **Requirements** ؟

هي مجموعة الوظائف والخدمات التي يقدمها النظام.

من يقوم بتحديد هذه المتطلبات ؟

الزبون **Customer**, مدير الشركة التي تحتاج النظام البرمجي **Managers** أو المستخدمون **End Users** (الموظفون في الشركة مثلاً الذي سيستخدمون البرنامج فعلياً), سنطلق عليهم **Stakeholders**...

ومهمتنا كـمهندسي برمجيات هي أيجاد الطرق والوسائل لجمع الوظائف والمعلومات التي من المتوقع أن ينفذها المنتج البرمجي وهذه العملية يطلق عليها Requirements Engineering وهي:

The broad spectrum of tasks and techniques that lead to an understanding of requirements is called *Requirements Engineering*. *SE practitioner approach P120*

تأتي أهمية المتطلبات من أن أي سوء فهم أو تواصل في هذه المرحلة سيؤدي إلى نتائج سلبية ومن الممكن أن تؤدي إلى فشل المنتج البرمجي الذي بشكل عام يعتبر مكلف ماديّاً...

"The seeds of major software disasters are usually sown in the first three months of commencing the software project". ***Caper Jones***

سنكمل في هذه المحاضرة مع أنواع المتطلبات ثم مع الطريقة السليمة لربط المتطلبات وتنسيقها في وثائق تكون مرجعية للإنطلاق بالمرحلة القادمة في عملية تطوير المنتج البرمجي...

أنواع المتطلبات Requirements Types:

بدايةً إن عملية **تجميع وفهم وتثبيت** المتطلبات تتم في مرحلة الـ**Analysis** ولكن يمكن أن يكون البدء في تجميعها **Requirements Gathering** أو **Information Gathering** قد بدء من مرحلة الـ**Communication** لكن في مرحلة الـ**Analysis** يتم أخذ المعلومات المهمة والمفيدة في المشروع وتثبيتها وتوثيقها بحيث لا نستخدم المعلومات غير المهمة والتي يمكن أن تؤدي إلى فشل النظام لاحقاً وقد ينتقل أيضاً مجموعة من نشاطات هذه العملية إلى مرحلة الـ **Planning**.

وهنا ظهر ما يسمى **Feasibility Study** أي دراسة إمكانية نجاح المشروع وحسب نتيجة هذه الدراسة يتم أخذ القرار بإتمام المشروع من عدمه وذلك حتى لا يتم إضاعة الوقت.

ويمكن أن تندرج المتطلبات تحت الأصناف التالية:

- **User Requirements:**

تمثل الوظائف التي تم فهمها من الزبون أو المستخدم ونقوم بإعادة كتابتها وصياغتها بطريقة يفهمها المستخدم ولانستخدم فيها مصطلحات تقنية أو غير مفهومة بالنسبة لهم ونستفيد منها في التأكد من أننا قد فهمنا الوظائف التي طلبها المستخدم بشكل سليم.

> **User requirements are statements, in a natural language plus diagrams, of what services the system is expected to provide to system users and the constraints under which it must operate.** ***SE by Sommerville P83 | our course, slide 3***

- **System Requirement:**

وهي مستنتجة من الـUser requirement حسب فهم المحلل Analyst للمتطلبات مع تفصيل أكثر ووضع الخطوط العريضة للحل لتكون مرجع يمكن الرجوع لها عند الحاجة .

> **System requirements are more detailed descriptions of the software system's functions, services, and operational constraints. The system requirements document (sometimes called a functional specification) should define exactly what is to be implemented. It may be part of the contract between the system buyer and the software developers.**
>
> ***SE by Sommerville P83 | our course, slide 3***

مثال مطروح في السلايدات يمكن استخدامه لفهم متطلبات المستخدمين ومتطلبات النظام بشكل أفضل:

نلاحظ هنا أن الـuser requirement تكتب بشكل تجريدي بحيث أي شخص يقرأها يستطيع أن يفهمها بينما الـsystem requirement تأتي لتفصيل وشرح ما كتب في الـuser requirement وموجهة للأشخاص التقنيين بشكل أكبر...

User requirements definition

1. The Mentcare system shall generate monthly management reports showing the cost of drugs prescribed by each clinic during that month.

System requirements specification

1.1 On the last working day of each month, a summary of the drugs prescribed, their cost and the prescribing clinics shall be generated.
1.2 The system shall generate the report for printing after 17.30 on the last working day of the month.
1.3 A report shall be created for each clinic and shall list the individual drug names, the total number of prescriptions, the number of doses prescribed and the total cost of the prescribed drugs.
1.4 If drugs are available in different dose units (e.g. 10mg, 20mg, etc) separate reports shall be created for each dose unit.
1.5 Access to drug cost reports shall be restricted to authorized users as listed on a management access control list.

Domain Requirement:

هي المتطلبات والقيود المشتقة من مجال عمل هذا النظام وليست مشتقة من متطلبات المستخدمين فعلى سبيل المثال في نظام برمجي طبي معين يوجد متطلبات (من وزارة الصحة مثلاً) على النظام تحققيها:

Insulin Pump System that delivers insulin on demand include the following domain requirement:

- **The System safety should be assured according to standard IEC 60601-1: Medical Electrical Equipment – Part 1 General requirement for Basic Safety and Essential Performance.**

http://www.SoftwareEngineering-9.com/Web/Requirements/DomainReq.html

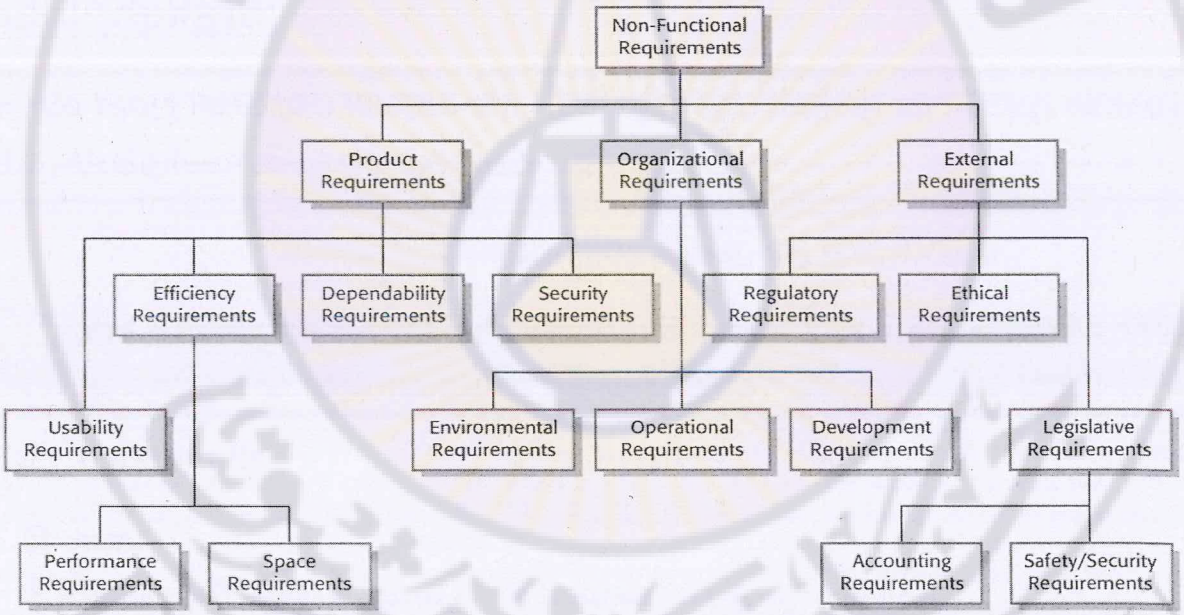
Functional Requirements:

هي مجموعة المتطلبات والوظائف والـ Actions التي يقدمها النظام وهي تعتمد على طبيعة هذا النظام والمستخدمين المتوقعين لاستخدامه وعدم الدقة فيها يمكن أن يخلق بعض المشاكل كما في اختلاف تفسير المتطلبات بين المستخدم والمطورين.

هذه المتطلبات يجب أن تكون **كاملة Complete** أي يجب أن تصف كل متطلبات النظام وأن تكون **متناسقة Consistent** أي أنها يجب أن تكون بعيدة عن التضارب مع بعضها.

Non-functional Requirements:

هي المتطلبات التي لا تتصل مباشرةً بالخدمات أو الوظائف التي يقدمها النظام وعلى الأغلب لن يطلبها المستخدم حرفياً أي يجب أن تتحقق ضمن المشروع بشكل بديهي مثل التقيّد بالوقت المحدد والأمان واستخدام معايير محددة, وغالباً ما يتم تطبيقها على **كامل** النظام وليس على جزء أو خدمة أو ميزة منه, وفي حال عدم تطبيق أحد هذه المتطلبات يمكن أن يجعل النظام عديم الفائدة. فعلى سبيل المثال الفشل في نظام قيادة الطيار الآلي لتحقيقه للمتطلبات الاعتمادية **Reliability Requirements** سيجعل منه نظام غير آمن ولن يتم استخدامه.



ويمكن أن تصنف الـ **Non-Functional Requirements** إلى 3 أنواع وهي:

❖ Product Requirements:

هي الصفات التي تحدد أو تقيد تصرف المنتج البرمجي ويجب أن يتمتع بها الـ **Final Product** مثل: **Execution Speed, Reliability, Security Requirements , Usability Requirements**

❖ Organizational Requirements:

هي المتطلبات الناتجة عن اتباع السياسات والإجراءات التنظيمية لكلا المؤسستين (الزبون وفريق التطوير) مثل :

Developing Environment, Programming Language, Process Standards ...etc.

❖ External Requirements:

هي المتطلبات المشتقة من عوامل خارجية عن النظام البرمجي وعمليته التطويرية مثل:

- Interoperability Requirements المتطلبات التوافقية مع أنظمة برمجية أخرى يتم استخدامها (كـنظام بنكي).
- Legislative Requirements المتطلبات التشريعية التي تضمن تنفيذ النظام البرمجي ضمن القوانين.
- Ethical Requirements متطلبات أخلاقية تضمن أن يتم قبول النظام من قبل مستخدميه.

Stakeholders:

كي يتم تجميع المتطلبات السابقة على أكمل وجه يجب التواصل مع أشخاص محددين والذين نطلق عليهم اسم Stakeholders ويمكن تعريفهم بـ :

الأشخاص المعنيين والمستفيدين من نجاح النظام.

Anyone who benefits in a direct or indirect way from the system which is being developed.

ويمكن تصنيفهم إلى:

- ☑ Users
- ☑ Developers يمكن أن يشمل الـ designer – analyst - supporter
- ☑ System Administrators
- ☑ Testers
- ☑ Support Staff
- ☑ Maintainers
- ☑ System managers
- ☑ System owners

Requirements Engineering Process:

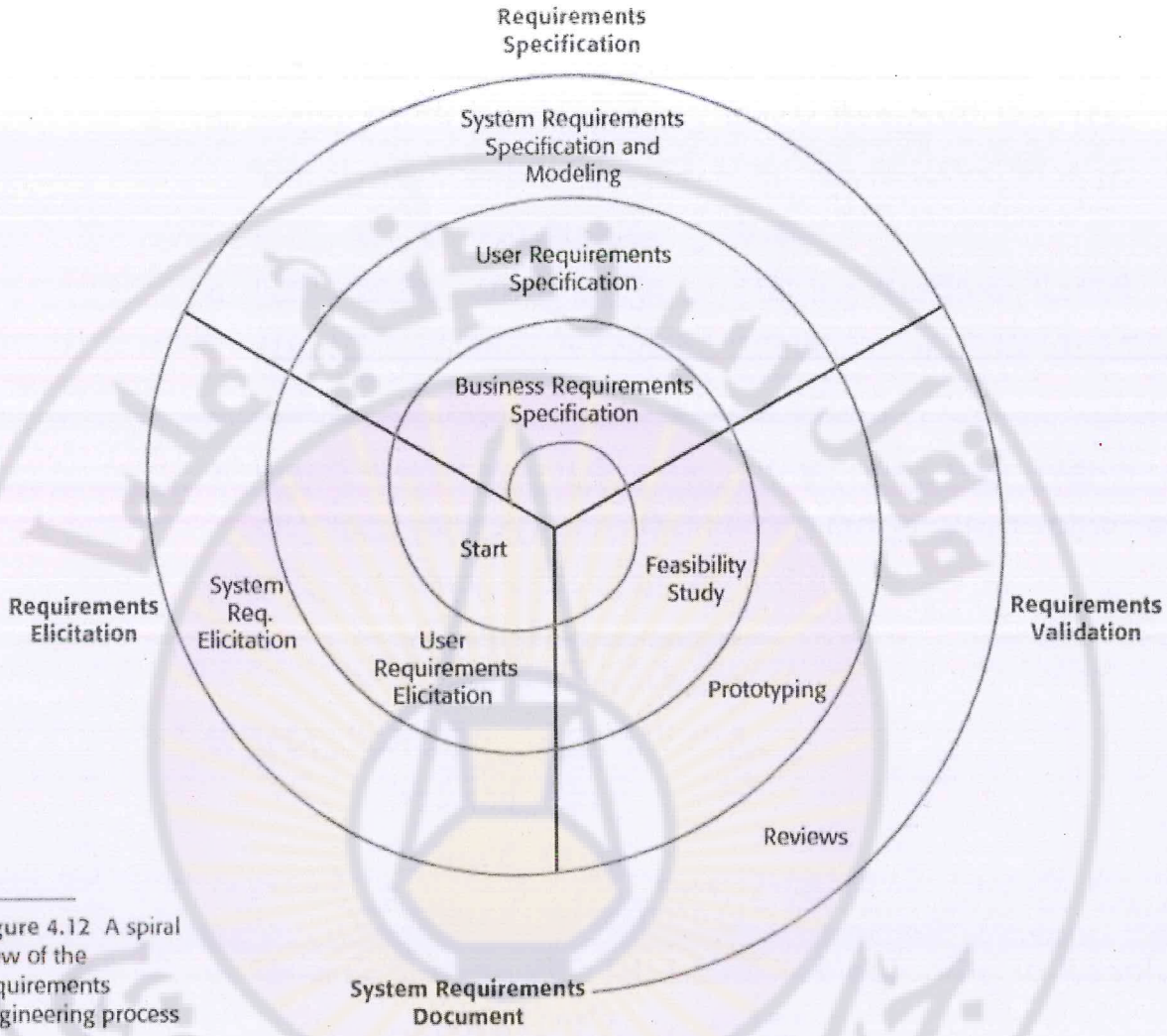


Figure 4.12 A spiral view of the requirements engineering process

كما قلنا سابقاً إن عملية تجميع وفهم المتطلبات هي عملية صعبة وأن أي خطأ في فهم المتطلبات سيؤدي إلى فشل النظام ولذلك يوجد عدد من العمليات العامّة التي يجب القيام بها لضمان عملية تجميع وفهم المتطلبات الأساسية للنظام وهي:

- **Requirements Elicitation:**

هي المنحى المبدئي/البنائي وتتألف من الـ **work tasks** التالية:

1. Requirements discovery:

هي المرحلة التي يتم فيها عملية تجميع المتطلبات وبما أن أساس هذه العملية هو **التجميع Gathering** لذلك هنالك عدة أنواع للتجميع منها:

← Interviews المقابلات:

عبارة عن مجموعة من المقابلات التي تتم لأشخاص محددين بغاية تجميع المعلومات الخاصة عن موضوع معين.

- **خطوات الـInterview :**

a. التحضير للمقابلة بشكل أساسي وتحديد المعلومات التي يجب الحصول عليها وبالتالي يجب تحديد من هم الأشخاص الذين يجب أن أحصل على هذه المعلومات من خلالهم.

b. إعطاء الشخص ملخص كامل وسريع عن نوع الأسئلة مع الحفاظ على أخلاقيات العمل و خصوصية هذا الشخص بحيث إذا أراد إنهاء المقابلة في أي وقت يريده لا نمنعه ذلك.

c. تصميم الأسئلة بحيث أحصل على أكبر قدر ممكن من المعلومات خلال أقصر زمن, وأن يبقى هذا الشخص ضمن سياق المقابلة كما يمكن ملاحظة أن هناك نوعين من الأسئلة:

✓ **Open-ended questions**

✓ **Closed-ended questions**

d. توثيق المقابلة عن طريق عدة طرق:

✓ تسجيل صوت

✓ تسجيل فيديو

✓ تسجيل ملاحظات

e. تقييم المقابلة من خلال إعادة كتابة المعلومات التي تم الحصول عليها ثم مراجعتها والتأكد من صحتها.

- **مساوئ الـInterview :**

i. الشخص الذي نجري معه المقابلة يكن أن يستخدم مصطلحات ومفردات من الممكن أن تكون غير مفهومة لنا.

ii. الكلفة الكبيرة للوقت المستغرق في المقابلات.

← Surveys الاستبيانات:

مجموعة من الأسئلة يتم توزيعها على مجموعة من الأشخاص ثم يتم تحليل النتائج والحصول على المعلومات من خلال تحليل هذه النتائج وبالتالي يتم التخلص من مشكلة إضاعة الوقت في **Interviews**.

- مساوئ الـSurveys :

عدم الحصول على نتائج صحيحة 100% وذلك بسبب عدم شعور الشخص الذي يجري الاستبيان بمسؤوليته عن المعلومة التي يعطيها.

← Observation المراقبة:

المراقبة شئ مزعج للبعض وبالتالي عندما يشعر أي شخص أنه تحت المراقبة سوف يقوم بأفعال لم يعتد على القيام بها وهذا يؤدي إلى نتائج غير صحيحة وتستخدم عادة في الـprototype لأنها تستخدم عندما يكون المستخدم ليس لديه قدرة على التعبير.

2. **Requirements classification and organization:**

تصنيف المعلومات التي تم جمعها وتنظيمها بشكل متماسك في إطار معين يخدم عملية تطوير النظام.

3. **Requirements prioritization and negotiation:**

ترتيب الأولويات بين المهمات التي يجب تنفيذها بحيث يضمن عدم حدوث تضارب أو تقاطع بينها ويتم تحديد هذه الأولويات بين المستخدم و المطورين.

4. **Requirements specification:**

يتم توثيق المتطلبات التي تم استنباطها من الـ User Requirements بشكل مبدئي initial وهذه الوثائق ليست النهائية فهي تختلف عن وثائق وتوصيف المرحلة القادمة التي ستكون أكثر تفصيلاً.

- **Requirements Analysis (Specification):**

يتم وضع توثيق رسمي للمتطلبات يطلق عليه System Requirements Specification SRS حيث من الممكن أن يحوي مجموعة من النماذج التصويرية Graphical Models أو النماذج الرياضية Mathematical Models أو حالات الاستخدام Usage Scenarios أو حتى نموذج أولي...

- **Requirements validation:**

يتم في هذه المرحلة مراجعة الـ Specification الخاص بالمتطلبات من قبل فريق من الـReviewers (يتألف من مجموعة من المهندسين, الزبائن أوالـstakeholders) ليضمنوا من أن جميع المتطلبات قد تم توصيفها وحل جميع التناقدات والغموض فيها.

- **Requirements management:**

مجموعة من النشاطات التي تساعد فريق التطوير في التحكم و متابعة عملية **Requirement Engineering** في أي مرحلة من المراحل السابقة بطريقة صحيحة.

وهنا تكون قد انتهت محاضرتنا لنكمل في المحاضرة القادمة مراحل Requirements Engineering Process بشكل أكثر تفصيلاً (حسب السلايدات)...

-النهاية-

Business Requirements أو User Requirements

Information Finding أو Information Gathering أو Requirements Discovery

Grouping أو Clusters أو Requirements Classification

Arrangement أو Requirements prioritization

what the system should do أو Function Requirements

what the system should have أو Non-Functional Requirements

وكنّا قد بدأنا في المحاضرة السابقة بالـRequirements Engineering والتي هي مجموعة من الطرق والنشاطات في عملية جمع متطلبات المنتج البرمجي وتعرفنا على المراحل التي تمر بها عملية " هندسة المتطلبات " وقد توقفنا عند مرحلة Requirements Analysis والآن سندخل بها بشكل تفصيلي لكن قبل ذلك طرح الدكتور سؤال هام كنا قد ذكرناه في المحاضرة السابقة وهو:

في أي مرحلة تتم عملية الـRequirements Engineering؟

تشمل المراحل التي لها علاقة بالـ Requirements بشكل أساسي ولكن بنسب مختلفة وهذه المراحل هي :

Communication – Planning – Analysis

والآن نكمل بشرح باقي المراحل الأساسية للـ Requirements Engineering :

Elicitation – Analysis (specification) – Validation – Management

2- Requirements Analysis (Specification):

هي المرحلة الخاصة بتوثيق الخصائص والـ **Requirements** للمشروع بشكل موجه لكلاً من الزبون **Customer** والـمطوّر **Developer** .

كما أنها تسمح للمحللين **Analysts** بتوضيح المتطلبات الأساسية التي يعتمد عليها المشروع وتعطي إمكانية بناء نماذج تصف تصور المستخدم للمشروع والوظائف التي يقوم بها والمشاكل التي يواجهها والعلاقة بينهما, وهذه النماذج تعطي صورة عن سلوك النظام والمعطيات التي يتعامل معها وكل ذلك من خلال عملية (**Documentation** - التوثيق).

عند القيام بعملية التوثيق يجب الوضع بعين الاعتبارعدة نقاط مهمة وهي:

a. متطلبات المستخدم يجب أن تكون مفهومة للمستخدمين النهائيين والزبائن الذين من الممكن ألا يكون لديهم أي معرفة تقنية من قبل.
b. **System Requirements** يتم كتابتها بشكل أكثر تفصيلاً كما يمكن أن تحوي على معلومات تقنية أكثر من الـ **User requirements**.
c. المتطلبات يمكن أن تكون جزء من العقد لمطوري النظام لذلك من المهم أن تكون كاملة قدر الإمكان.

وبالتالي يجب اتباع طرائق توثيق معيارية ودقيقة ومضمونة لأن أي خطأ يحدث في هذه المرحلة سيؤدي إلى فشل المشروع.

إذاً ماهي الطرق اللازمة لتوثيق هذه المعلومات والمتطلبات ؟

من أهم الطرق التي لدينا هي الطرق الكتابية أو السردية الـ Textual والتي يمكن أن تصنف تحت:

☑ **Natural Language:**

تعني توثيق ما تم جمعه في تقرير منسق ومكتوب بلغة طبيعية **Natural Language** حيث يمكن أن نعطي تعابير وتوضيحات كثيرة وكل متطلب يمكن صياغته في جملة واحدة مرقمة.

إلا أن هذه الطريقة غير محبذة لأنها في بعض الأحيان قد تكون غير واضحة لكلاً من المستخدمين و المطورين, كما أن المتطلبات الوظيفية والغير وظيفية يمكن أن تندمج مع بعضها في هذا التقرير.

☑ **Structured Natural Language:**

ظهرت لمعالجة المشاكل التي تظهر عند استخدام الـ **Natural Language**,حيث تم اقتراح وضع بنية أو قالب **Template** موحد وكل جزء من هذا النموذج يعبر عن جزء من المتطلبات على أن يتم الالتزام بهذا القالب في كتابة التقارير.

وفي حال وجود نقص أو خطأ يمكن معالجة ذلك إمّا بإضافة ملحق **Appendix** أو بتغيير القالب وهذا يعتبر من مساوئ الـ**Structured Natural Language** .

☑ **Design Description Languages:**

مع أن استخدامها تراجع إلى حدا ما إلا أنها كانت قادرة على كتابة المتطلبات بلغة خاصة مثل اللغات البرمجية أي لها مصطلحات وتعابير خاصة بها فقط وعلى الرغم من أنها قادرة على التوصيف بشكل أكبر إلا أنها تحتاج إلى أشخاص محددين للتعامل معها, كما يجب مواكبة تطور مراحل هذه اللغة التي من الممكن أن تكون غير معممة على جميع الشركات وبالتالي تم إهمالها.

ولدينا أيضاً من الطرق الأخرى :

☑ **Graphical Notations: الطريقة الرسومية**

التعبير عن المتطلبات باستخدام الرسم أفضل من استخدام الكتابة حيث يؤمن الرسم ميزتين أساسيتين وهما:

- **Compression of The Data ضغط البيانات:**

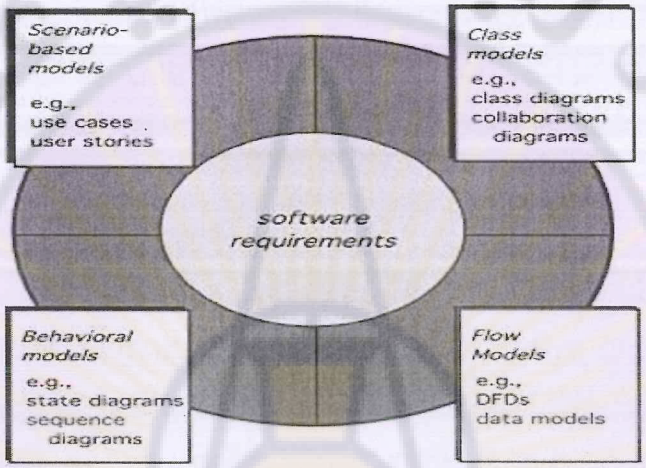
أي عندما نقوم بشرح فكرة باستخدام الكتابة يمكن أن تستغرق ثلاث أو أربع صفحات بينما إذا عبرنا عنها باستخدام الرسم يمكن أن يأخذ حجم الرسمة أو الصورة فقط ويوصل الفكرة نفسها وبالتالي يتم تخفيف الضغط عن المستخدم والمطور.

- الرسم يوصل المعلومة بشكل دقيق بشرط أن توفر معلومات شاملة وأن تكون الرموز مفهومة المعنى.

وبالتالي يجب استخدام رموز محددة ومعيارية يمكن أن يفهمها كل من يقرأها في جميع أنجاء العالم ولذلك تم الاتفاق على مايسمى

UML (Unified Modeling Language)

وهي لغة نمذجة موحدة ومعيارية تستخدم للتعبير عن الرموز والأشكال والمخططات بحيث يفهمها الجميع.



ويمكن النظر إلى المتطلبات (من ناحية النمذجة والرسم) من أربع وجهات نظر Perspectives :

Scenario-Based Models: النماذج المعتمدة على السيناريوهات مثل الـ **Use cases** و الـ **User stories**.

Behavioral Models: النماذج السلوكية أي التي تعتمد على سلوك الكائنات **Entities** في النموذج لدينا.

Flow Models: النماذج التي تعتمد على تدفق البيانات.

Class Models: النموذج التركيبي.

وجهات النظر هذه **Perspectives** سنتعرف عليها بالتفصيل وستكون عناوين للمحاضرات القادمة بإذن الله.

☑ Mathematical Specifications:

وتسمى أيضاً الـ Formal Language حيث يتم وضع توصيف للمتطلبات بطريقة تصفها بشكل دقيق فلا يمكن أن يتواجد توصيف غيره عن طريق استخدام الرياضيات المنطقية Logical Math أو التعابير الرياضية Expressions أو النظريات Theories.
فاذا كانت لدينا القدرة في استخدام التعابير الرياضية للتعبيرعن جميع المتطلبات يمكن استخدام Auto Code-Generation توليد الكود البرمجي بشكل آلي وهذا موضوع بحثي قيد التطوير حالياً و يطلق عليه Model Driven Development.

بعد أن تعرفنا على بعض الطرق في توثيق المتطلبات كيف سيكون شكل المستند Document الذي سينتج لدينا ؟

سنعرض الآن اقتراح لهذا المستند :

Preface: صفحات بداية (يختلف عن المقدمة) يمكن أن يحوي على نسخة المستند الحالي Version 1 أو Version 2

تسمى Revision History.

Introduction: مقدمة تتحدث عن النظام المطلوب والشركة التي تطلبه والـ Business Outcome المتوقع منه.

Glossary: مجموعة المصطلحات والكلمات المستخدمة في المستند.

User Requirements Specification: الجزء الخاص بمتطلبات المستخدم User Requirements

System Architecture: نظرة عامة لبنية النظام.

System Requirements Specification: الجزء الخاص بمتطلبات النظام System Requirements.

System Models: النماذج والمخططات التي ستوثق كل ماذكر سابقا".

Appendices: الملحقات.

Index: الفهرس.

في سورية يتم التوجه نحو المستندات الـLess Models More Text أما الشركات فتتوجه نحو More Models Less Text.

3- Requirements Validation:

يتم في هذه المرحلة التأكد من أن جميع التفاصيل والمتطلبات التي يريدها المستخدم قد تم توثيقها حيث على المتطلبات تحقيق مايلي:

Validity: المتطلبات مطابقة لما يريد الزبون.

Consistency: التكامل.

Completeness: الكمالية.

Realism: واقعية الحلول والمتطلبات والقدرة على انجازه.

Verifiability: إمكانية الوصول لكود برمجي يحقق المتطلبات.

ويمكن تنفيذ هذه المرحلة من خلال عدّة طرق :

- المراجعات Reviews :

حيث يقوم بالتحقق من الشروط السابقة مجموعة من الأشخاص من قسم ضمان الجودة Quality Assurance يعتبرون مستوى عالي من المطورين من حيث الاختصاص والخبرة وسنطلق عليهم Reviewers.

- النماذج الأولية Prototypes:

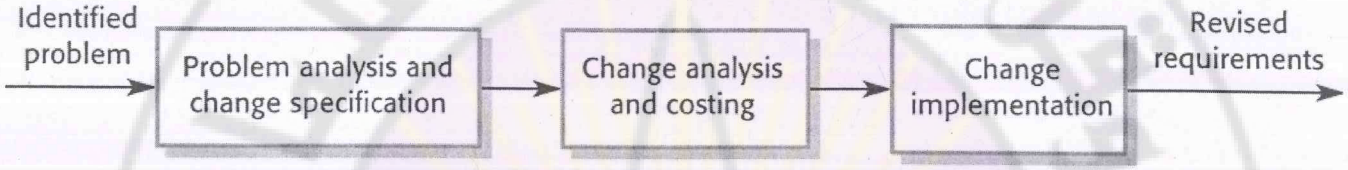
يتم وضع نموذج أولي للمتطلبات وعرضه على المستخدم للتعبير عن رضاه من المتطلبات.

- Test-Case Generation:

يتم استخراج حالات اختبار Test Cases يمكن تطبيقها على الكود البرمجي لاحقاً, أي التحقق من قدرة المستند على إعطاء حالات اختبار Testability وتتم عملية الاستخراج من قبل فريق الـQuality Assurance أيضاً.

4- Requirements Management:

هي إدارة عملية جمع وتصفية **Refinement** ونقاش المتطلبات ومن أهم مراحل الـ**Management** هي القدرة على معالجة التعديلات والتغييرات **Changes** التي تظهر مع تقدّم عملية التطوير وتحديد أهمية هذا التغيير وإمكانية تلبيته ويتم ذلك بـ:



Problem Analysis and Change Specification: يتم تحليل هذا التغيير والتحقق من أنه متطلب أساسي قابل للتحقيق.

Change Analysis And Costing: ندرس عملية إضافة هذا التغيير وكم سيكلف من الوقت ومن التكاليف الخاصة بالتعديل من ثم مناقشتها مع الزبون

Change Implementation: تضمين هذا التغيير في المستند الجديد.

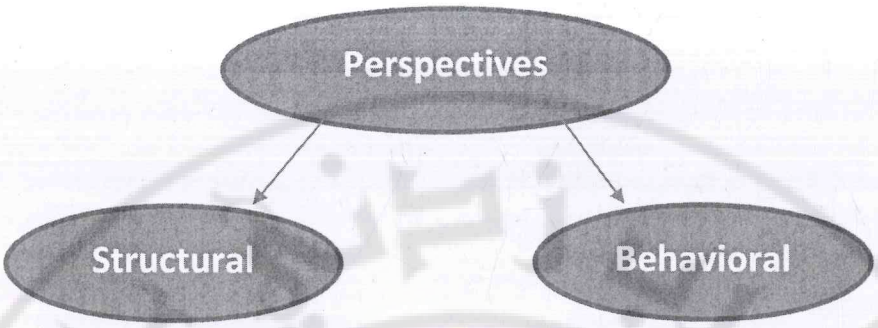
سنتكلم في هذه المحاضرة عن الـ System Modelling أي نمذجة النظام وكلمة نمذجة تعني رسم المخططات, وبدايةً سنستخدم هذه المخططات في نمذجة المتطلبات Requirements التي تعرفنا عليها في المحاضرات السابقة وذلك لهدفين هما **توثيقها** و **لتكون صلة الوصل بين الـ Analyst والـ Developer من جهة وبين الـ Analyst والـ Customer من جهة أخرى.**

ولذلك من المهم جداً توثيق هذه المتطلبات باستخدام المخططات بشكل دقيق و مفهوم بحيث تكون مرجع لكل من الزبون والمطوّرين في حال حدوث أمر ما, أي من الممكن أن تكون جزء من العقد.

إن عملية بناء المتطلبات بما فيها من بناء المخططات والوثاق تتم في مرحلة الـAnalysis ولقد تم تسميتها **بالتحليل** لأنه في هذه المرحلة يقوم المحللون Analysts بالتعبير عن هذه المتطلبات ليس كـكتلة واحدة و إنما من عدة وجهات نظر, حيث يمكن أن يكون للـ Customers وثيقة ونماذج مخصصة لهم ووثيقة للـDevelopers مخصصة لهم, ويمكن أن يتم جميع المخططات والنماذج في وثيقة واحدة وكل طرف يهتم بما يخصه وهو المنحى المنتشر...

سنلاحظ في هذه المحاضرة أن كل فكرة يمكن أن تتواجد **بأكثر من وجهة نظر** وقد تختلف من مرجع إلى آخر ويعود سبب الاختلاف إلى عدم تواجد صيغة موحدة لهذا العلم وأن كل الأفكار قد تم تطويرها من قبل شركات مختلفة فكل شركة تنظر للفكرة بالشكل الذي يناسبها ونحن علينا أن نستخدم ما نراه مناسباً وأن نطور ما أسماه الدكتور Common Behavior أي وجهة نظر مشتركة بين وجهات النظر هذه...

ولكن هنالك بعض وجهات النظر العامّة Perspectives والتي تصلح في أي مكان و منها:



❖ **Structural Perspective: منظور بنيوي**

وهو المنظور الذي يهتم بتحديد الكائنات **Entities** و العناصر **Components** التي يتألف منها النظام وبنية البيانات التي يعالجها, وسنستخدم في هذا المنظور **Class Diagrams**.

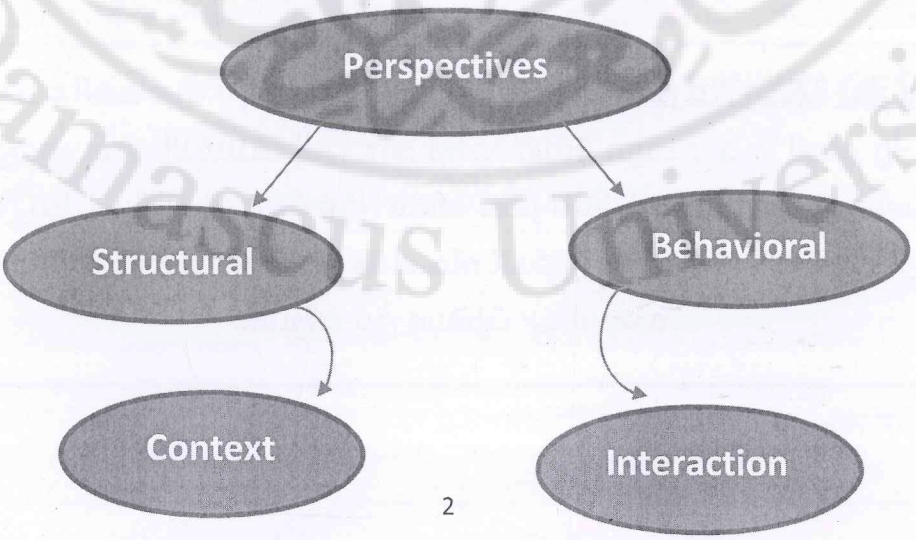
❖ **Behavioural Perspective: منظور سلوكي**

يعبّر عن عملية التفاعل **Interaction** بين النظام و الكائنات والأحداث **Events**.

ويتم بتحديد الكائنات المجردة **Abstract entities** التي تصف النظام وحركة الاتصال **Flow of The Data** داخل هذه الـ **Entities** .

ومن المخططات المستخدمة في هذا المنظور **State Diagram, Sequence Diagram**.

ويمكن إضافة وجهتي نظر بشكل أكثر تفصيلي على وجهات النظر العامّة السابقة وهي:



❖ **Interaction Perspective: منظور التفاعل**

يتم فيه نمذجة التفاعلات بين النظام والبية المحيطة به **Environment** أو بين عناصر النظام نفسه.

ومن المخططات المستخدمة هذا المنظور هو **Use Case Diagram**.

لكن ما الفرق بين الـ Interaction والـ Behavioural ؟

بعض وجهات النظر تعتبر الـ**Interaction** و الـ**Behavioral** وجهة نظر واحدة, وبعضها يميّز بينها كالآتي:

Behavioural: السلوكي

Set of interactions happens internally between set of objects or Entities to achieve certain functions.

Interaction: التفاعل

External interactions happens between the environment and the Software.

❖ **Context Perspective: منظور السياق**

هي الـ **External Projection** أو وجهة نظر المراقب الخارجي أي يحدد **فعلياً** أين تقع منظومة الـ **Software** الذي نطوّره ضمن البيئة الأكبر المحيطة به.

وهذا يعني أنه لا يوجد جزء مستقل وإنما دائماً سيكون جزء من منظومة أكبر وأعلى.

من بداية عملية وضع توصيف النظام **System Specification** يجب تحديد الـ **System Boundary** لمعرفة ماهي حدود الـ **Software** للعمل المنجز أي معرفة الـ **Environment** المحيطة به, ويتم ذلك بالعمل مع الـ**Stakeholders** أو باستخراجها من المتطلبات لتحديد الوظائف التي يجب أن يتم

تضمينها للنظام, فمثلاً يمكن أن يتقرر تضمين بعض المهام ليتم تنفيذها أو أن يتم تنفيذها يدوياً (موظف) أو أن تتم هذه المهمات بمساعدة نظام خارجي...

الـ Environment يمكن أن تتكون من Systems أو Humans **أي** كل ما هو خارج حدود العمل يعتبر من الـEnvironment.

DFD Diagrams:

من أحد النماذج أو المخططات في الـ Context Perspective هي مخططات Data Flow Diagrams (DFD) والتي من الوهلة الأولى لاسم هذه المخططات يمكن أن توحي بأن ليس لها علاقة بالسياق Context وإنما تتعلق بتدفق البيانات والمعلومات في النظام الذي يتم تطويره وهذا صحيح و**لكن** يمكن استخدام هذه المخططات في وصف البيئة المحيطة بالنظام وسياقه ونطلق عليها حينها Context DFD وستتألف من عدة مستويات level0,level1,level2.

وسنطلق على المستوى الذي يعبّر عن السياق بـ Context DFD أما المستويات الأخرى فتضيف قليلاً من التفاصيل على المخطط في كل مستوى...

الدكتور قد ميّز بين الـContext Level والـ Level 0 ومن الممكن أن نجد بعض المراجع تعتبر الـContext Level هي نفسها الـLevel 0. (اختلاف وجهات نظر)

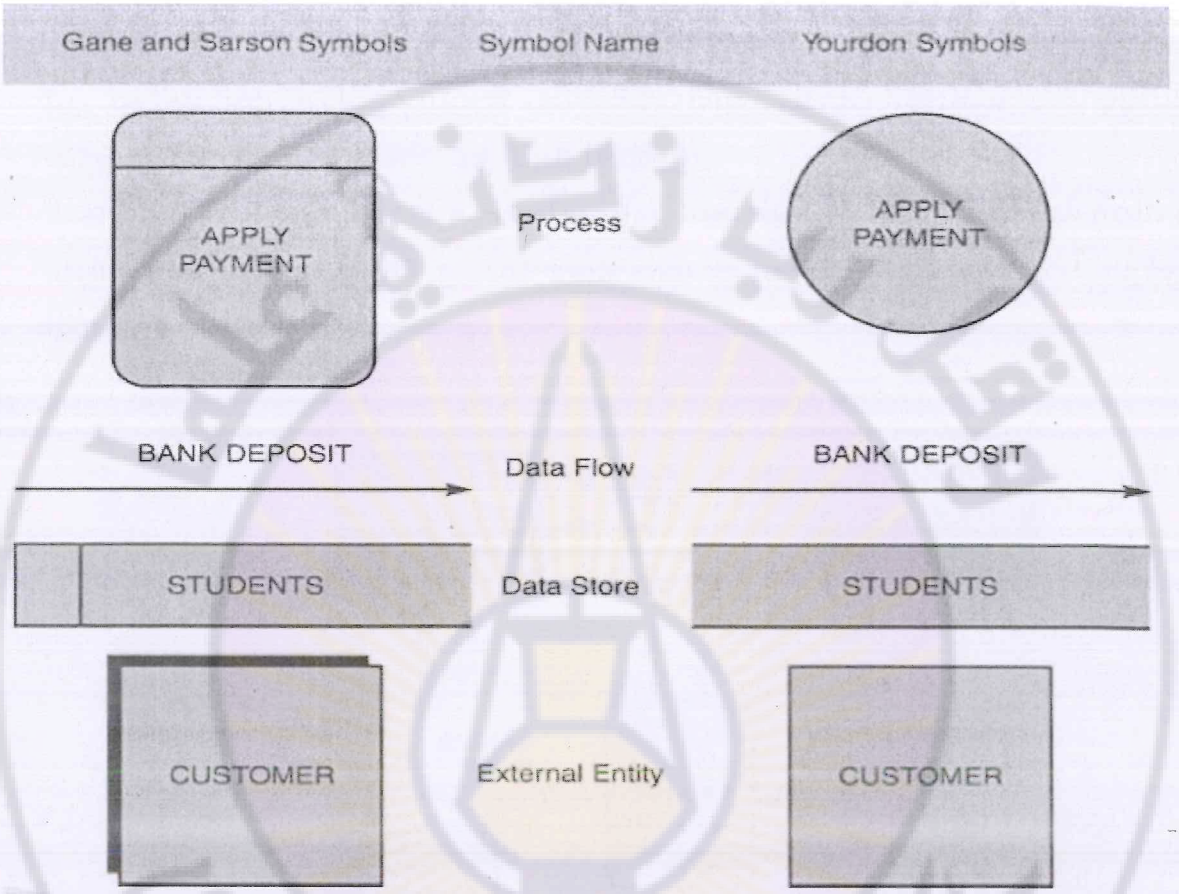
The DFD is presented in a hierarchical fashion. That is, the first data flow model (sometimes called a level 0 DFD or context diagram) represents the system as a whole. Subsequent data flow diagrams refine the context diagram, providing increasing detail with each subsequent level.

SE-Practitioner Approach 7th edition – Page 187

وسنتعرف على الرموز المستخدمة في هذه المخططات والتي هي:

Data Flow (Relation) – Process – External Entity

وهذه الرموز لها نوعين أو لنقل وجهتي نظر وهي **Yourdon Symbols** و **Gane and Sarson Symbols**:



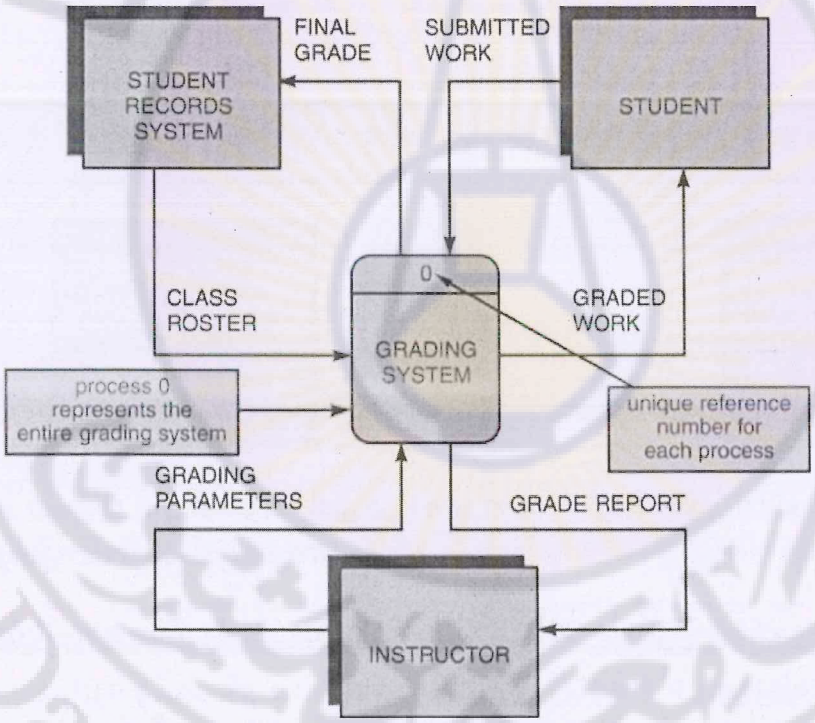
*الـ**Data Store** تستخدم في مستويات أدق من **Context Level** أو المستوى 0 وتستخدم للتوضيح في داخل الـ**Processes** وفي مخططات أخرى للـ**DFD**.

عندما نمذجة الـ **Context** باستخدام مخططات الـ **DFD** سيكون هناك **Process واحدة** وهي النظام الذي نقوم بتطويره وعدد غير محدود من **الكائنات Entities** و **Relations** أيضاً عدد غير محدود لكن ماهي انواع الـ **Relations** التي ممكن أن تتواجد في مخططات الـ**DFD**:

Data Flow That Connects	*Okay to Use?*	
	YES	NO
A process to another process	☑	☐
A process to an external entity	☑	☐
A process to a data store	☑	☐
An entity to another entity	☐	☑
An entity to a data store	☐	☑
A data store to another data store	☐	☑

- من غير المنطقي أن أقوم بربط الـ Entities ودراسة العلاقات المتبادلة بينها وبالتالي تكون علاقة Entity to Entity **غير قابلة للتحقيق.**
- الـ Process هو الوسيط الحصري لعملية تبادل المعطيات الداخلية والخارجية ومنه تكون علاقة Entity to Data store وعلاقة Data store to another data store أيضاً **غير قابلة للتحقيق.**

مثال:



كما قلنا سابقاً أن الـ Context يحوي **Process واحدة** وستكون هنا Grading System وسيكون لدينا عدد من الـ Entities وهي المدرس Instructor ونظام ثانوي لسجلات الطلاب Students Records System و الطالب Student.

يوجد مثال آخر في السلايد 19 من سلايدات System Modelling يوضح Levels of DFD Context Diagram .

Use Case Diagrams:

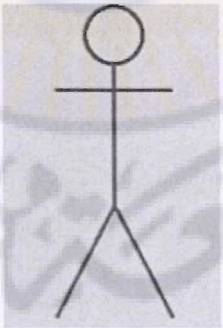
حالات الاستخدام Use Cases تعني كيفية تبادل الـInteraction بين النظام والبيئة والكائنات المحيطة به وبشكل عام هي مجموعة الخدمات والوظائف (المتطلبات) التي يقدمها النظام إلى Entity ما وتعد موجهة بشكل رئيسي إلى الـ User, أي يجب أن نبتعد عن التعقيد فيها.

فهي تعتبر قاعدة أساسية للتواصل مع الزبون وتأكيد المتطلبات ويمكننا من خلالها اشتقاق المخططات التحليلية والتصميمية للنظام في مراحل نمذجة متقدمة.

لايوجد في الـ Use Case Diagram داعي لتمثيل الوظاف والـObjects الداخلية في النظام فقط نمثل الـ Business Processes أو الوظائف المستقلة التي يمكن ان نحصل عليها نتيجة طلب ما, أو الوظائف التي تحقق متطلبات المستخدمين.

رموز مخططات حالات الاستخدام : Use Case Diagram Symbols

a. **Actor:** هو كينونة Entity تُعتبر المتفاعل (الممثل) الوحيد في البيئة المحيطة النظام ويمكن أن يمثل شخص أو منظمة أو أي أجهزة أو أنظمة خارجية ويتم رسمه على شكل Stick Figure :



وله عدة أنواع ويكون الرئيسية منها:

- Primary business actor:

هو الكينونة التي تطلب خدمة من النظام.

- **Primary system actor:**

هو الكينونة التي تتواصل بشكل مباشر مع النظام.

على سبيل المثال في الـ **Queuing System** لبعض الشركات يمكن أن يتواجد موظف خاص لإعطاء ورقة الدور والتسجيل في الرتل ويكون هو الـ**Primary System Actor** أما الزبون الذي بمجيئه وطلبه دور قد فعّل الحدث يكون الـ**Primary Business Actor**.

بعض وجهات النظر تقوم بنمذجة فقط الـ **Primary System Actors** أما الـ **Primary Business Actors** يتوضحون بحقل الـ **Trigger** في التمثيل الجدولي أو النصي لحالات الاستخدام.

هل من الممكن أن يكون الـ **Business Actor** و الـ **System Actor** هو نفس الشخص؟

بالطبع ممكن ومثال على ذلك عندما يريد شخص أن يسحب مبلغ من الصراف الآلي **ATM** عن طريق بطاقته الأئتمانية فهو عندما يقوم بإدخال معلومات البطاقة يكون **System Actor** وعندما يحصل على المبلغ (أي استفاد) يكون **Business Actor** .

أما الحالات الأخرى لـ **Actors** فتكون ثانوية ويطلق عليه **Secondary Actor** أو **Server Actor**:

- **External Server Actor:**

هو من يتواصل مع النظام ولكن بمرتبة أقل من الـ **Primary** حيث يستجيب لخدمة تم طلبها من قبل **Primary Actor** ويقوم بالرد و**مثال على ذلك** مدير منتدى عندما يتلقى طلب انضمام إلى هذا المنتدى و تكون مهمته ويرد قبول أو رفض الطلب.

- **External Receiver Actor:**

هو أيضاً يتواصل مع النظام ولكن لا يقوم بأي عملية رد أي أنه يستقبل التعليمات أو المعلومات من النظام فقط و**مثال على ذلك** أمين مستودع يتلقى طلبات من النظام ويقوم بشحنها إلى وجهاتها.

الفرق أن الـ **Server Actor** يقوم بعملية تفاعل مع النظام أي أنه يقوم باستقبال طلب ما من النظام و بعد ذلك يرد هذا الطلب بينما الـ **Receiver Actor** يقوم فقط باستقبال طلب من النظام.

b. **Use case:** شكل أهليلي نضع في وسطه اسم هذه الحالة, يعبر عن خدمة أو وظيفة يقدمها النظام.

c. **Relations:**

وهي التي تربط طالب الخدمة **Actor** مع الخدمة **Use case** التي يطلبها أو بين خدمة و أخرى ولها عدة أنواع:

- **Association**:

هي العلاقة التي تربط بين **Actor** و **Use Case**.

وعادةً مايكون المهم في المخطط هو فقط الـ **Primary Actors** أما الـ **Secondary Actors** فيتم إسقاطهم خاصة عندما تكون التفاصيل كثيرة وكبيرة في المخطط لأنها ستعقد أكثر من أن تفيد.

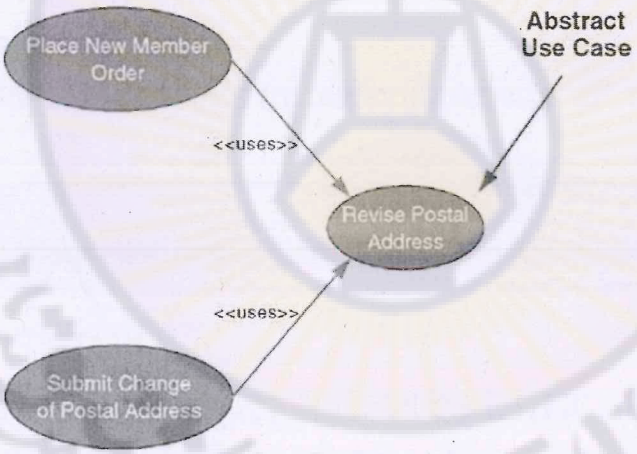
وفي نسخة الـ **UML الثانية** سنلاحظ في كثير من الأدوات أنه قد تم إسقاط إشارة السهم من الخط الواصل بين الـ**Actor** و الـ **Use case** وقد كانت تستخدم إشارة السهم للتميز بين المفعّل لـ **Use case** أي طالب الخدمة وبين المستفيد منها وقد لاحظوا أن 90% من طالبي الخدمة هم المستفيدين منها لذلك تم اسقاطها.

- **Abstraction**:

تستخدم لصياغة وظيفة أو حالة استخدام في النظام والتي يمكن أن يتكرر استخدامها في حالات استخدام أخرى.

مثال: حالتي استخدام للتسجيل طلب في نظام ما, لدينا الحالتين **Place New Member Order** أي تسجيل طلب والحالة الثانية **Submit Change of Postal Address** وهي حالة تغيير عنوان بريدي.

سنلاحظ أن كلا الحالتين تشتركان في عملية **تأكيد العنوان البريدي**, أي يمكن فصل هذه الحالة عنهما ولنسميها **Revise Postal Address** وتكون هذه الحالة حالة تجريدية **Abstract Use Case** اي لايمكن تفعيلها لوحدها (لا يستطيع المستخدم تفعيل حالة **تأكيد العنوان البريدي**) وإنما يتم تضمينها **Include** في حالات أخرى, وتكون **إجبارية الحدوث** أي في حالة **تسجيل العضوية** من المؤكد أنه سيتم تفعيل حالة **تأكيد العنوان البريدي.**

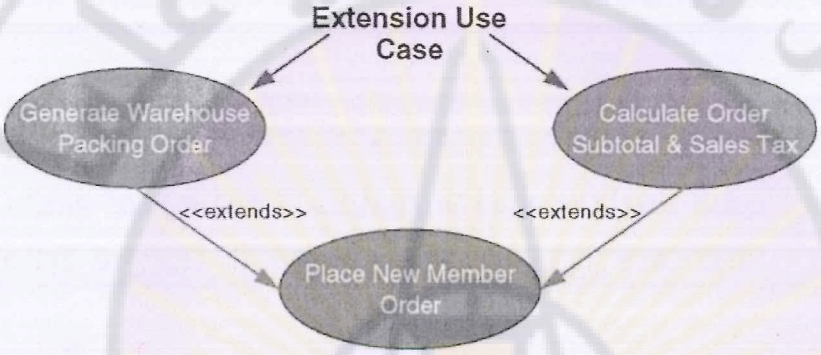


UML notation: a directed arrow labeled *<<uses>>* or *<<includes>>* and it is read as the example: Place new Member Order *Uses* Revise Postal Address.

- **Extension**:

تستخدم لصياغة وظيفة أو حالة تستكمل حالة أساسية أخرى, أي حالة **اختيارية** يمكن أن يتم تفعيلها من حالات أخرى عند تحقق شرط معين من قبل الـ **Actor**.

مثال: عملية تسجيل طلب **Place new member order** يمكن أن يختار الـ **actor** ان يتم تسجيل الضراب المدفوعة مع الفاتورة **Calculate Order Subtotal** أو توليد فاتورة من المستودع مع الفاتورة العادية **Generate Warehouse Packaging Order** والحالات الأخيرة هي **Extension Use Cases**.



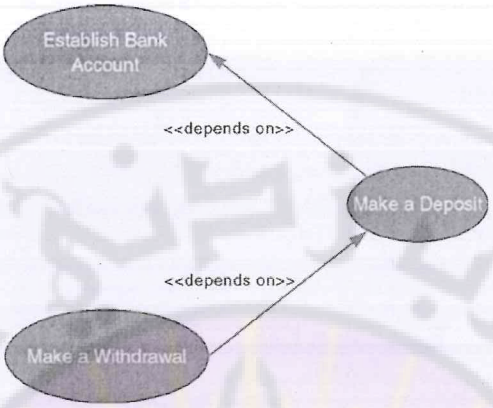
UML notation: a directed arrow labeled *<<extends>>* from the extension UC to the main UC and it is read as: Generate Warehouse Packing Order *extends* Place new member Order.

ملاحظة: اتجاه الأسهم في الـ **UML** لا يستدل به عن الحالة الأساسية من الحالة المشتقة, ويستخدم فقط في القراءة.

- **Depends On**:

علاقة تنشأ بين حالتي استخدام عندما يشترط حدوث الحالة الأولى حدوث الحالة الثانية قبلها.

مثل عملية سحب مبلغ من البنك تعتمد أولاً على عملية إيداع مبلغ في البنك وعملية الإيداع تعتمد على عملية فتح حساب وهكذا...

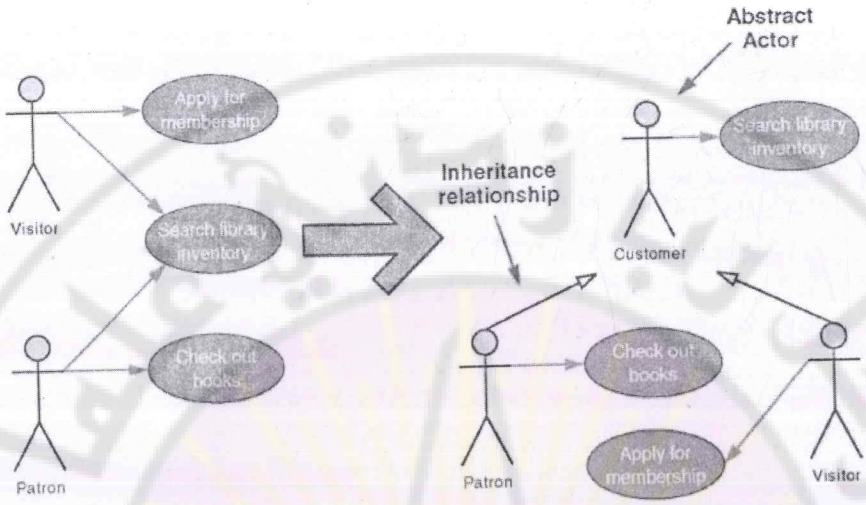


بعض وجهات النظر تعتبر علاقة الاعتمادية **Depends On** زيادة تعقيد للمخطط ويستغنون عنها بتوضيح الاعتمادية في حقل الـ **Precondition** في جدول **Use Case**.

UML notation: a directed arrow labeled *<<depends on>>* from the UC that depends on another UC.

➢ **Inheritance:**

علاقة نستخدمها عند وجود تصرفات مشتركة بين أكثر من **Actor**, فنقوم بجعل **Actor** واحد ويسمى **Abstract Actor** بتفعيل هذه الحالات المشتركة ثم نقوم بجعل الـ **Actors** الآخرين يرثون منه هذه التصرفات (التفعيلات) التي يقوم بها.



UML notation: a directed arrow beginning at one actor and pointing to the abstract actor whose interactions the first actor inherits.

ملاحظة:

يجب أن يكون هناك دائما حدود للنظام بحيث يكون الـ **Actors** ينتمي للـ **environment** والـ **Use Cases** تنتمي للـ **System Boundary** أي يجب عدم وضع الـ **Actor** ضمن الـ **System Boundary** ولكن كلاً منهم في جهة لوحده.

Primary actor أهم من **Secondary actor** أي عند الرسم الأولوية لـ **Primary** وفي حال كان الرسم يحتاج إلى تفصيل أكثر نضع الـ **Secondary**.

في اخر سلايدين من **System Modelling** يوجد مثالين لمخططين **Use Cases** مع نمذجة نصية (جدولية) لـ**Use Case Diagram**.

-النهاية-

نتابع في هذه المحاضرة مع الـ System Modelling فقد توقفنا في المحاضرة الماضية عند الـ Use Case Diagram والذي يُستخدم في منظور الـ Interaction عند نمذجة متطلبات النظام, وقد ابتدأ الدكتور في هذه المحاضرة بالتذكرة ببعض النقاط الهامة من المحاضرات السابقة:

- تم ذكر الـModelling في عدّة Process Models على أنها طور Milestone من أطوار تطوير المنتج البرمجي فهل يعتبر الـ Modelling هو نفسه طور التحليل أو التصميم Analysis or Design ؟؟ لا, بل يعتبر الـModelling جزء أو أداة في طور التحليل.
- كما تم ذكر مصطلح الـ Modelling في الـ Requirement Specification والـ Specification هو تقرير كامل ومفصل عن المشروع يوضح جميع المتطلبات التي تم جمعها من الـ Stakeholders وهذا التقرير سيعبر عن وجهتي نظر, الأولى من وجهة نظر المستخدم User أي ستكون هذه المتطلبات مكتوبة بطريقة سهلة و أكثر تجريداً بحيث يستطيع أن يفهمها المستخدم ووجهة النظر الثانية ستكون من وجهة نظر المطوّر Developer حيث سيكتب بطريقة أكثر تقنية ليحدد ما يقوم به النظام بالضبط كما أنه سيكون نقطة مرجعية في عملية التأكيد على المتطلبات Validation.
- من أهم الطرق والـ Notations في كتابة تقارير الـ Requirements Specification هي الـ Text والـ Models .

- الـ Models هي وسيلة و أداة للتعبير عن متطلبات و مواصفات النظام **بطريقة بيانية** عن طريق المخططات Diagrams.

- من غير الممكن التعبير عن كامل متطلبات النظام بمخطط واحد أو حتى بتقرير واحد, حيث يجب تقسيم المتطلبات بحيث كل منها يعبر عن وظيفة معينة ومن ثم تجميع هذه المتطلبات من عدّة وجهات نظر ومن عدّة أجزاء ثم نعبّر عن هذه الأجزاء بالمخططات

المناسبة ليتكون لدينا نظرة كاملة ومتكاملة Integrated View عن النظام ويتم بعد ذلك الانتقال إلى مرحلة التصميم وكتابة الكود.

ونكمل الآن مع نوع جديد من المخططات :

مخطط الصفوف Class Diagram:

من أهم المخططات التي تعبر عن Structural Components لأي نظام برمجي هي الـ Class Diagrams , وهو تمثيل تجريدي لأجزاء النظام فهو يعتبر من المخططات المستخدمة لتوضيح وجهة النظر البنيوية Structural.

لكن ما هو الصف Class ؟

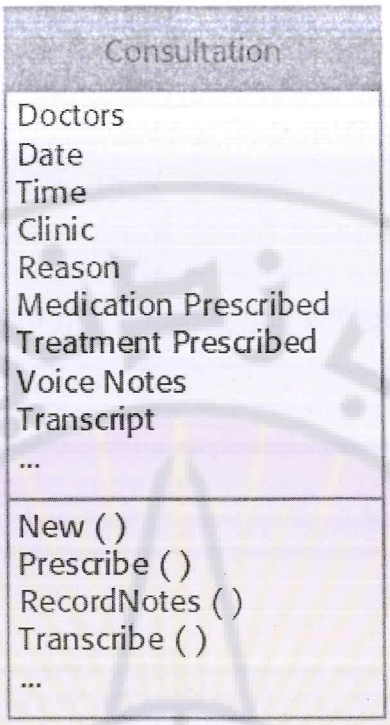
الصف يعبّر عن مجموعة من الكائنات Entities or Objects التي تشترك بمجموعة من الصفات والسلوكيات فالكائن هو شيء يختزن بداخله صفات محددة وسلوك محدد والصف يعتبر كحاوي Container لهذه السلوكيات والصفات ومهمته الأساسية هي تغليف Encapsulation هذه الأشياء المشتركة وجمعها.

ولكن ما الذي يتم تحديده أولاً, الصفوف أم الكائنات؟

يتم أولاً تحديد الكائنات أو الـObjects ومن ثم يتم تحديد الصفوف التي يمكن تشكليها من الكائنات ويتم الحصول على الكائنات عن طريق قراءة تفصيلية و فهم دقيق لمتطلبات النظام والمتطلبات يتم الحصول عليها من Use Case Diagrams.

والمخطط Diagram هو رسم العلاقات والروابط بين الـ Classes (Classes with Relations) .

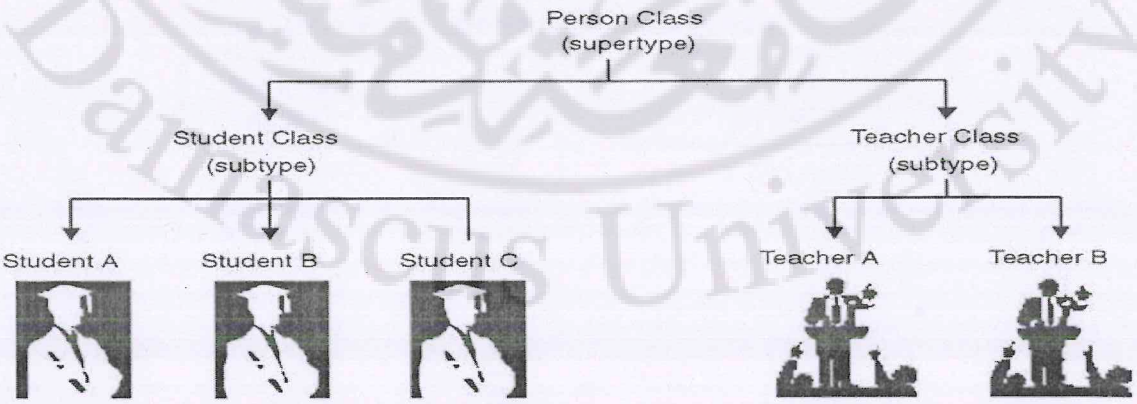
ويتم تمثيل الـ Class في الـ UML notation بمستطيل يحوي اسم الصف ويمكن وضع الصفات أو الـAttributes و الطرائق (السلوكيات) Methods أيضاً.



من المهم تحديد العلاقات التي تربط بين الـ **Classes** ولها عدة أنواع:

- **الوراثة Inheritance:**

هي عملية اشتقاق مجموعة من الصفوف تسمى **Subtypes** من صف **Supertype** لترث صفاته **Attributes** وسلوكياته **Behaviours** ويضيف كل **Subtype** صفاته وسلوكياته الخاصة به وتكون العلاقة بين صفوف الآباء **Supertypes** والأبناء **Subtypes** علاقة وراثة.



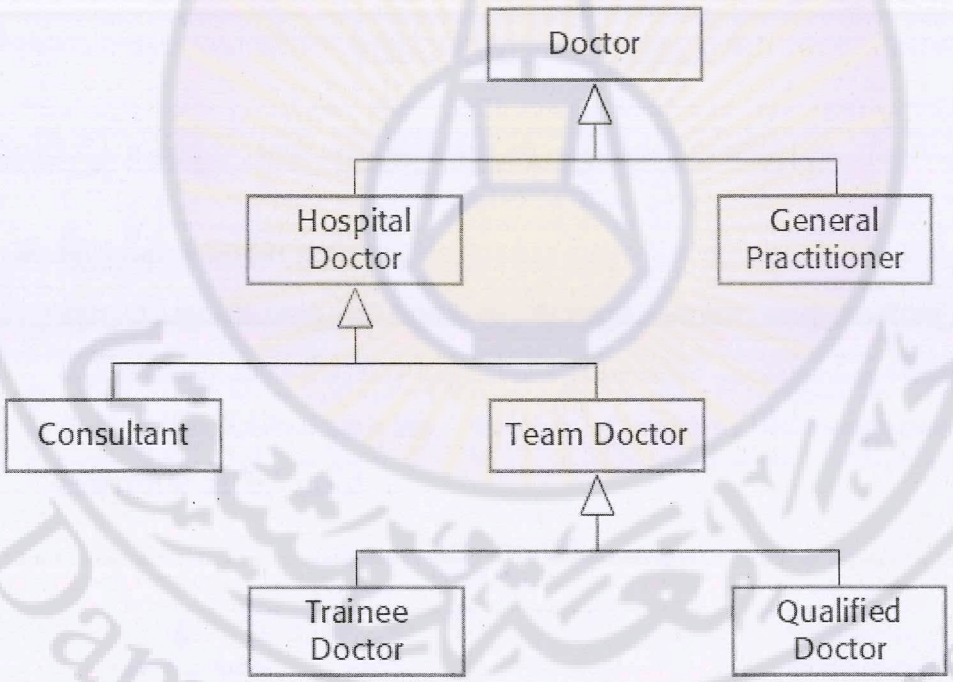
التعميم Generalization:

هي عملية جمع الصفات **Attributes** و السلوكيات **Behaviours** المشتركة في عدّة صفوف مختلفة في صف واحد يسمى **Supertype** ثم نشتق منه الصفوف المختلفة التي شكلناها منه وتسمى **Subtypes**.

(تشبه الـ **Inheritance** لكن بوجهة نظر مختلفة)

Polymorphism – the concept that different objects can respond to the same message in different ways.

Override – a technique whereby a subclass (subtype) uses an attribute or behavior of its own instead of an attribute or behavior inherited from the class (supertype).

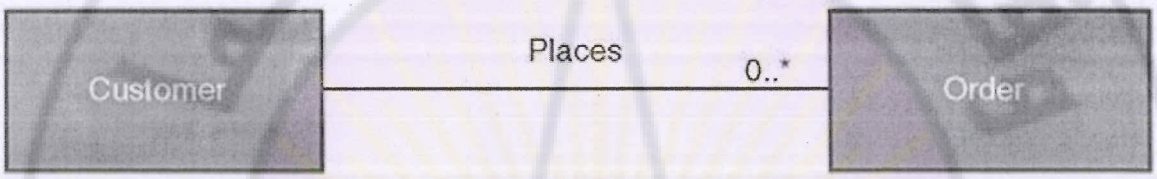


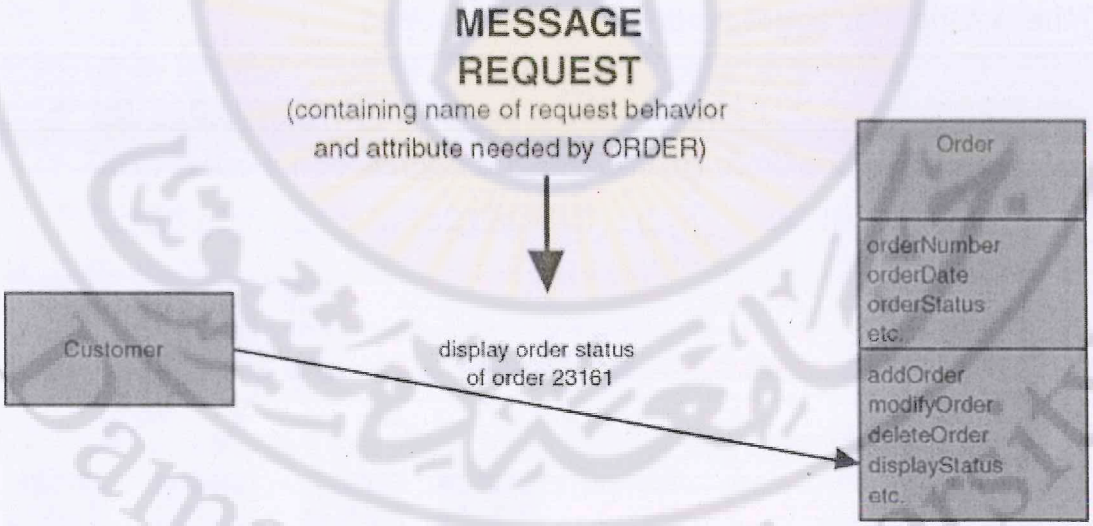
Association:

هي علاقة الربط بين صف وصف آخر وعلاقة الربط تكون باستدعاء صف لطرائق وصفات الصف الآخر ويتم تمثيله بخط ويمكن قراءتها بعدة اتجاهات و يمكن أن تكون موجهة.

- **Multiplicity**: هي التي تحدد تماما عدد الـ **Object** من **Class** معين التي تتفاعل مع عدد معين من الـ **Object** من الـ **Class** الآخر.

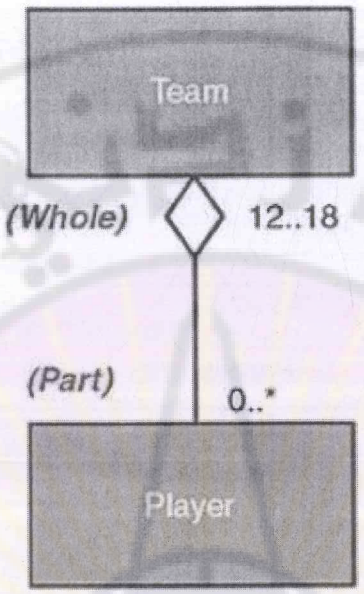
Multiplicity – the minimum and maximum number of occurrences of one object/class for a single occurrence of the related object/class.





Aggregation:

هي علاقة تربط بين صفين حيث يكون صف أول يحوي جزء أو عدة أجزاء (كائنات) من صف ثانٍ ويتم تمثيلها بخط يحوي من طرف الصف الأول **Diamond Shape**.



In UML 2.0 the notation for aggregation has been dropped.

Composition:

تشبه علاقة الـ**Aggregation** لكن الصف الأول يكون مسؤول عن إنشاء الصفوف الجزئية وتدميرها ويتم تمثيلها بشكل ديناري **Diamond Shape** مطمس.