معـــايير في تقييـــم لغـــات البرمجـــه

بماذا يجب أن نفكر عند إختيار لغه برمجه .

عروة عيسى . ٢٠٠٤/١٢/٥ نسخه أوليـــّـــه ، للحصول على النسخة النهائية راجع : **www.orwah.net**

الفهرس ..

مقدمه ٤
لمن هذا الكتاب ٥
ملاحظات لابد منها ٥

دعم OOP *البرمجه غرضيه التوجه . أو البرمجه الشيئيه*

مقدمه عن غرضيه التوجه ٦
التغليف Encapsulation ٦
الوراثه من أنماط موجوده ٧
تعدديه الأشكال (Polymorphism) ٨
مناقشه ٩

لغات البرمجه الغرضيّه الصرفه مقابل اللغات الهجينه ١٠
الوراثه المتعدده Multiple inheritance ١١
نموذج الأصناف الساكن مقابل نموذج مرجعيه الغرض ١٢
الإداره الآليه للذاكره ومجمع النفايات garbage collect ١٤
الاستدعاء المتأخر (وتعدديه الأشكال) Late Binding and Polymorphis ١٥
معلومات الأصناف في زمن التشغيل RTTI ١٥
القوالب والبرمجه الجينيه (Generic Programming) ١٦
التجريد Abstraction ١٧
بعض المزايا الأخرى ١٧
القياسيه ١٨
المقروئيه Readability **وتنظيم اللغه** ١٩
- سرعه التكويد ٢١
أمن اللغه Language Safe ٢٣
وصول منخفض المستوى low-level access ٢٦

السرعه Speed ٢٨

سرعه تطوير التطبيق ((الإنتاجيه)) ٢٨
سهوله اللغه ٢٨
قابليه إعاده الإستخدام ReUsibility ٣٠
التوسعيه Extendibility ٣٠
التخاطب مع نظم وتقنيات مختلفه ٣٠
دعم التقنيات المختلفه ٣١
توفر مكتبه أدوات واسعه Wide Component Library ٣٢
خلاصه ٣٢
السرعه في التنفيذ Executing Speed ٣٤

حجم التطبيق Application Size ٣٧
دعم المجاري المتعدده Multi Threading ٣٨
معالجه الإستثناءات exception handling ٣٩
دعم GUI **و** graphics ٣٩
الإنتشار والتسويق والدعم الفني Marketing and Support ٤١

المحموليه Portability ٤٤
المحموليه على صعيد نظم التشغيل ٤٤
اللغات التي تعمل على منصات خاصه ٤٤
نسخ من اللغه مخصصه للعمل على نظم مختلفه ٤٦
محررات مختلفه تدعم نظم مختلفه ٤٧
المحموليه على صعيد إصدارات نظام التشغيل المختلفه ٤٧
المحموليه على صعيد نظام تشغيل واحد_أجهزه مختلفه ٤٩
مكتبات زمن تشغيل (Run Time) ٤٩
الأدوات التي إستخدمناها في برنامجنا ٥٠

التكامل مع نظام التشغيل OS integrity ٥١
دعم الويب Web Supporting ٥٢
مفتوحه المصدر Open Source ٥٣

فريق الإعداد ٥٥

بسم الله الرحمن الرحيم

لغات البرمجه هي أدوات خاصه صممت لتحقيق أهداف مختلفه ، وكل أداه منها تملك ميزات تناسب مجموعه حالات أكثر من حالات أخرى ، وهذه الميزات نفسها يمكن إعتبارها نقاط قوه أحيانا ونقاط ضعف أحيانا أخرى ، حسب المشروع والغايه التي نريد تحقيقها . وأمور متعدده أخرى تتعلق بمستخدم الأداه وظروف الإستخدام.
هذه اللغات موجهه لحاجات مختلفه، ووجدت لتحل مشكلات مختلفه بطرق مختلفه، وتستخدم في بيئات برمجيه متباينه .
المشكله في لغات البرمجه أن فلسلفه كل لغه متعلقه بالأهداف التي بنيت اللغه على أساسها ، ومزايا لغه برمجه ما هي ناتج دمج هذه الأهداف من ناحيه والضريبه المترتبه على أولويه هذه الأهداف من إهمال دعم بعض المزايا المتعارضه معها من ناحيه أخرى ، خذ مثلاً بعض اللغات الشائعه :

- لغـــه C++ : من أهدافها القوه والتحكم، تتميز بالكثير من التعقيد ، ولها سرعه مميزه في التنفيذ .
- الدلفي: والهدف هو التركيز على برمجه النوافذ ، وزياده الإنتاجيه، مميزه في برامج E_Business
- الجافا : تسعى لقابليه النقل بدون كبير اهتمام بالسرعه ، مطرزه بالعديد من مزايا القوه ، مميزه في التطبيقات الموزعه والويب .
- VB : تهدف إلى السهوله وسرعه التطوير . وتبسيط برمجه النوافذ ، مميزه في تطبيقات أوفيس .

يمكننا ببساطه ان نقرر أن نجاح كل من هذه اللغات لا يعود إلى المميزات التي ذكرناها فقط ، فما هي إلا غيض من فيض ، وتوجد عشرات الأسباب الأخرى المهمه لتقرير أهميّـــه لغه برمجه ما أو نجاحها في الإنتشار والتسويق .

ماسأقدمه في هذه العجاله هو عرض مبسط للمزايا الأساسيه التي تحكم أهميه ونجاح لغات البرمجه من ناحيه، وما هي المعايير التي ينبغي وضعها في الحسبان عند إختيار لغه برمجه ما من أجل مشروع معين. وتبقى العديد من الجوانب بحاجه لمزيد من التوسع والتدقيق ويبقى المشروع قابلا للتعديل وإضافه معايير وبنود أخرى إلى هذه القائمه المبسطه والمختصره .
وليعذرني القاريء إذا وجد بعض النقص هنا ، وبعض الركاكه وعدم الوضوح هناك . فالمبتغى هو رضوان الله أولا ولفت النظر لبعض الجوانب المهمه ثانيا ، ويترك للقاريء حريه الرأي والقرار بعد الإضطلاع عليها .

. والله وليّ التوفيق .

لمن هذا الكتاب

هذا الكتاب موجه أساسا للأخوه التقنيين المهتمين بالإضطلاع على الميزات والفروقات بين لغات البرمجه الشائعه ويبقى مفيدا لكل للأفراد الذين يريدون البدء بإختيار لغه برمجه تناسبهم وتناسب نوعيه البرامج التي يرغبون بإنتاجها أو للشركات المعنيه بإنتاج تطبيقات برمجيه وتهتم بمعرفه لغات البرمجه الأنسب لكل مشروع .

ملاحظات لابد منها

نحن لا نقدم في هذا الكتيب نتائج جاهزه وخلاصات ثابته عن المشاريع المناسبه لكل لغه أو مقارنات ثابته بينها، ولانهتم بتقديم ذلك أصلا ، وذلك من منطلق الإقتناع أن هذا قد يكون رأي شخصي وقد أختلف به مختصون بالتقنيات وعلوم الحاسب حول العالم ، ولامجال لنحشر أنفسنا في هذه المعمعات المبنيه على أساس التعصب بالرأي لصالح لغه برمجه دون أخرى .

إنما هدفنا ذكر مزايا تقنيه وفنيه مؤثره على لغات البرمجه وشرحها ومناقشتها ، ثم قد نورد أمثله على دعم بعض لغات البرمجه لهذه المزايا للإستئناس ليس أكثر ، ويبقى المعيار المطروح هو الهدف والغايه أما المثال بلغات البرمجه فهو قابل للعلاج والمناقشه وإبداء الرأي الشخصي ، وما ورد من تحديد لإسماء لغات برمجه هو رأيي الشخصي بها وطبعا يخصني وحدي ويقبل الخطأ والتصحيح ..

ترتيب المعايير في هذا البحث ليس بحسب الأهميه ، إنما وضعت بحسب التسلسل الزمني في كتابتها والغايه هي تبيان أن أهميه كل معيار قد تختلف حسب نوع المشروع الذي نرغب بتطويره .

هناك فرق كبير بين لغه البرمجه كنحو Syntax وكمعايير أساسيه وبين بيئه تطوير اللغه DE (منفذ اللغه) . حيث قد يوجد للغه واحده أكثر من محرر وقد تختلف هذه المحررات فيما بينها في دعم بعض مزايا ، وسنتناول المحررات الأكثر شهره في هذا الكتاب أو المزايا المشتركه بينها .

يوجد تشابه كبير في بعض الحالات بين بعض اللغات المشتركه ، مثل لغات NET. مثلا ، أو نسخ معدله من اللغه تشترك معها بالمزايا الأم ، وفي هذه الحاله لن نشير إلى الأسماء المنفصله لكل لغه مالم يوجد إختلاف يستحق الذكر ، لذلك فليعذرني القاريء إذا وجد إجحاف في ذكر بعض اللغات المهمه بالإسم ، وذلك إما لتشاركها مع غيرها ، أو للتقصير في الحديث عنها .

دعم البرمجه غرضيّـــه التوجّه (OOP)

البرمجه غرضيه التوجه Object Oriented Programming :

البرمجه الغرضيه هي تقنيه ثوريه غيرت مسار البرمجه منذ عده عقود من الزمن .

لاتعتبر البرمجه الغرضيه التوجه تقنيه برمجه جديده ، بل تعود جذور هذه التقنيه إلى Simula-67 المطوره بواسطه العالم النرويجي Kristen Nygaard و Ole-johan Dahl في بدايات ١٩٦٠ ، وربما يمكننا أن نعتبر أن التمثيل الكامل لها هو Smalltalk-80 (١٩٨٠) والتي تعتبر أول لغه برمجه غرضيه (شيئيه) صرفه (Pure OOP) .

أصبحت البرمجه الغرضيه التوجه شائعه الإستخدام في النصف الثاني من عقد الثمانينات ، وأصبحت مدعومه من معظم اللغات الحديثه ، وربما أمكننا القول أن دعم اللغات للـــ OOP أصبح من المعايير المهمه التي تحدد أهميه لغه برمجه ما .

مزايا الـــبرمجه الغرضيه واضحه في العديد من لغات البرمجه مثل :

C++, Object-c , Object and turbo Pascal , CLOS (an OOP of Lisp) , Eiffel , Ada(on last incarnations), most recent Java ,Python . … etc

مقدمـــه عن غرضيـــه التوجـــه

وددت ان أستطرد قليلا في شرح مبسط عن غرضيه التوجه للتركيز على أهميه هذه التقنيه وضروره فهم مامعنى دعم لغه برمجه ما للـــ OOP . تعتمد هذه التقنيه في مضمونها على ثلاث مفاصل :

- التغليف encapsulation .
- الوراثه inheritance .
- تعدديه الأشكال polymorphism .

التغليف Encapsulation :

تعتمد فكره غرضيه التوجه على إخفاء البيانات .

وتستخدم الأصناف لتحقيق ذلك ،حيث يتم إخفاء البيانات داخل الأصناف الخاصه بها ، أو نقول بعباره أخرى يتم تغليف البيانات داخل الأصناف .

عاده يتم توضيح هذه الفكره بإستخدام ما يسمى الصناديق السوداء (black boxes) ، حيث لا تضطر أن تعرف كيف تتم الأمور بالداخل وما هي المحتويات الداخليه ، وكل ما يهمك هو كيف تتعامل مع واجهه الصندوق الأسود وتعطيه معطياتك وتأخذ النتائج بغض النظر عن ما يتم في الداخل . إن ما يهمك فعليا من الصندوق هو آليه التعامل معه (مع واجهته) ولا تعطي إهتماما كبيرا عن تفاصيل داخل الصندوق ،

خذ على سبيل المثال جهاز التلفزيون ، إذا أعتبرنا التلفزيون صندوقا أسود ، فإن كل ما يهمنا من هذا الصندوق هو كيفيه تشغيله وإطفاءه وتغيير المحطات وبعض الأمور الثانويه الأخرى ، دون إهتمام بفهم الدارات الداخليه المكونه له ، ودون الدخول في تفاصيل معالجته للإشاره وتحويلها إلى صوره وصوت .

إذن نخزن البيانات داخل الأصناف وعندها يمكننا أن نكتفي بمعرفه كيفيه إستخدامها من الخارج . إن كيفيه الإستخدام تدعى واجهه الصنف (class interface) وهي التي تسمح للأجزاء الأخرى من البرنامج بإستخدام الأغراض المعرفه من هذا الصنف ، وبالتالي عندما تستخدم غرض ما فإن معظم شفرته تكون مخفيه ، ونادرا ما تعرف ما هي البيانات الداخليه له حتى أنه قد لا توجد طريقه لدخول البيانات الخاصه به بشكل مباشر مالم تستخدم المناهج المتاحه على الواجهه والتي تسمح لك بتغيير وقراءه البيانات ، وذلك يعتبر من أهم الفروق بين البرمجه غرضيه التوجه و البرمجه الكلاسيكيه والتي تكون البيانات فيها عامه لكل الأصناف غير تابعه لصنف محدد كما أنك تستطيع تغيرها مباشره وبالتالي تقع في مطب ظهور أخطاء نتيجه عدم إمكانيه إختبار القيمه المدخله قبل إدخالها وذلك بسبب كون البيانات ظاهره ويمكن الوصول المباشر إليها ،،...

- **كما أن للتغليف ميزه سحريه للمبرمج نفسه**

لإنها تسمح له بتغيير التركيب الداخلي للصنف في التحديثات المستقبليه مع بقاء واجهه الصنف نفسها ، وبالتالي ستطبق التغييرات تلقائيا على بقيه الأغراض التي إستخدمت هذا الصنف بإقل عناء ممكن ، دون الحاجه لتغير شفرتنا في مناطق مختلفه من البرنامج . وذلك مثل حاله توابع API لها أسماء ثابته ، مهما تغيرت آليه عملها الداخليه طالما بقيت الأسماء كما هي ستبقى برامجنا تستدعيها بنفس الطريقه ، وتغيير في بنيه هذا التوابع لن يستوجب تغيير في البرامج التي تستدعيها

الوراثه من أنماط موجوده :

غالبا ما نحتاج بناء نموذج مختلف قليلا من صنف موجود ، بدلا من بناء صنف جديد من البدايه ، ربما نحتاج إضافه مناهج جديده أو خصائص أو تعديل أخرى موجوده .

والفكره من ذلك هي أننا نريد المتابعه من نقطه توقف الغير ، وليس البدء من الأول لذلك سأقترح أننا نستطيع فعل ذلك بطريقتين :

- نسخ الشفره من هناك ولصقها هنا . وبذلك ستضاعف شفرتك مرتين ، ناهيك عن الأخطاء ، والغرق في تفصيلات تبعدك عن مشروعك الأساسي ، والخروج عن مبدأ مركزيه الشفره .

- لماذا لا تقوم بدلا من ذلك بإستخدام إحدى أروع ميزات البرمجه الغرضيه : ألا وهي الوراثـــــــه (inheritance) .

وبالتالي نستطيع ببضعه سطور شفره فقط أن نبرمج كائن جديد (زر مثلا) يحوي جميع ميزات الزر العادي وزياده بعض الخصائص والدوال الخاصه بنا .

ملاحظه :

إن البرمجه غرضيه التوجه تفضل القدره على التوسع و إعاده الإستعمال (reusability) ، على أي شيء آخر ، حيث أنك تستطيع كتابه شفرات تستخدم أصناف من بنيه وراثيه معقده دون أي معرفه بالأصناف المحدده التي تشكل جزء من هذه البنيه ، وبكلمه أخرى تبقى هذه البنيه الوراثيه وبرامجك التي تستخدم هذه البنيه قابله للتوسع والتغيير ، حتى بوجود آلاف السطور من الشفره التي تستخدمها . ولكن بشرط واحد أساسي : أن يكون الصنف السلف لهذه الشجره الوراثيه مصمم بعنايه فائقه .

وبالتالي تضمن أنك لن تتوقف في مرحله ما عن تطوير المشروع لغرقك في كميه هائله من الشفره والتي أصبحت غير مفهومه وغير صالحه لكل الحالات ،

وهذه هي تركه غرضيه التوجه الأساسيه ، أنها جعلت تطوير البرامج الضخمه أمرا ممكنا وسهلا .

تعدديه الأشكال (Polymorphism)

توفر هذه الخاصيه القدره على استدعاء المناهج المحدده في صنف ما إعتمادا على نوع هذا الصنف .

وتتيح الاشاره إلى صنف على أنه صنف آخر يكون بالعاده في نفس فرع شجره الوراثه كالاشاره للصنف الابن على أنه أحد الآباء. مما يتيح اعاده الاستخدام للاصناف و جعل البرامج أكثر قابليه للتطور وأكثر تفاعليه في زمن التشغيل (Run Time) .

وبالتالي فالعمليه نفسها تتصرف بشكل مختلف في أصناف مختلفه . the same operation amy behave differently on different classes

منـــــاقشـــــه :

هذه المزايا الثلاث (الاصناف،الوراثه،تعدد الاوجه) لازمه في أي لغه برمجه موصوفه بأنها لغه برمجه شيئيه.

فالتغليف هو صلب غرضيه التوجه الأساسي ويتم تمثيله برمجيا بالقدره على بناء الأصناف . وفي حال كانت لغه البرمجه لا تدعم الأصناف فعندها لا أرى سببا من وجهه نظري لتسميتها لغه برمجه غرضيه التوجه .

لغه الفيجوال بيزك Visual Basic دعمت الأصناف في الإنتقال من الإصداره ٤ إلى الإصداره ٥ ، وربما تكون اللغه الأساسيه بين اللغات المشهوره التي تمتاز بضعف في دعم OOP خاصه في الإصدارات المبكره .
تعتبر معظم اللغات السابقه الذكر تدعم الوراثه بشكل جيد ، ماعد VisualBasic نفسها التي لم تستطع دعم الوراثه بشكل جيد في نسخه VB 6 ، وتم دعمها لاحقا في لغه VB.Net. كذلك تفتقد JavaScript إلى دعم الوراثه بشكل مشابه لVB .

ربما يمكنني القول أن VB هي الخاسر الأكبر بين اللغات الأكثر شعبيه في معيار دعم الـــ OOP .
ولكن VB.NET تداركت كل ذلك ، وبشكل عام في هذه الدراسه ماسنذكره عن C# سيكون موجودا في VB.net مالم نذكر خلاف ذلك .
كذلك لغه Perl تعاني ضعفا واضحا في هذه الناحيه ، فهي ليست لغه غرضيه التوجه تماماً ، وتم دعم الأغراض مؤخرا في دوره حياه اللغه ، كما أنها هي الأخرى لاتدعم الوراثه بشكل جيد ؟ .
لاحظ الشفره اللازمه لتعريف غرض في Perl :

```
package MyClass;
sub new}
my $class = shift;
my $self;{} =
bless $self, $class
$self->initialize();  # do initialization here
return $self;
{
```

هذه الشفره موافقه للشفره التاليه في لغه Paython :

```
class MyClass:
pass
```

لغات البرمجه الغرضيّه الصرفه مقابل اللغات الهجينه :

من الفروق الأخرى بين لغات البرمجه التي تدعم البرمجه غرضيه التوجه هو مدى دعمها لتقنيات البرمجه التركيبيه التقليديه، واللغات الصرفه هي تلك التي تدعم النموذج الشيئي فقط، فتستطيع التعامل مع الاصناف والمناهج فقط ولا تستطيع تعريف دوال وبرامج فرعيه تقليديه أو تعرف متغيرات عامه (global).

ومن بين اللغات التي نتحدث عنها يمكن اعتبار Eiffel و Smalltalk على أنها لغه شيئيه صرفه مئه بالمئه . كذلك يعتبرون الكثيرون أن Java لغه شيئيه صرفه ، وهذا هو رأيي الشخصي .

يبدو للوهله الاولى أن هذه الميزه سلبيه لا سيما وأنك تستخدم العديد من المناهج الثابته والتي لا تختلف كثيرا عن الدوال والبرامج الفرعيه في اللغات الهيكليه، وكل ما نجنيه هو زياده تعقيد الكود اللازم لتمثيل هذه الاجراءات.

في رأيي الخاص أن اللغات الشيئيه الصرفه تقدم الكثير للمبرمجين المستجدين حيث أنها تجبرهم على استخدام تقنيات البرمجه الشيئيه وتعلُّم هذا النمط من البرمجه.

وبنفس الوقت قد تنزعج من ذلك في كثير من الأحيان . حيث ستجبرك الكثير من هذه اللغات على فهم فلسفه البرمجه الغرضيه كلها ولصقها أمام عينيك قبل البدء بكتابه تطبيقات ولو كانت بسيطه . مثال لطباعه عباره Hello world في جافا نحتاج إلى كتابه الشفره التاليه :

```
public class HelloWorld {
        public static void main (String[] args) {
            System.out.println("Hello, world!");
        }
    }
```

يجب أن يعرف المبتدء العزل ، الأنماط ، المناهج ، الصنف String ، مصفوفات ، مناهج أصناف مقابل مناهج أغراض .

يجب أن يعرف كل ذلك من أجل هذه الـــ Hello World الصغيره ؟ . *وفوق ذلك يجب أن تحفظ بملف إسمه "HelloWorld.java``* .

في المقابل فإن باسكال الشيئيه ولغه C++ و C# وPaython يمثلون نموذج من اللغات الهجينه أو المدمجه (hybird languages)، فكل منهم يتيح للمبرمج أن يستخدم النموذج التقليدي بالإضافه لغرضيه التوجه .

الوراثه المتعدده Multiple inheritance:

بعض اللغات تدعم تقنيه في الوراثه تسمى الوراثه المتعدده ، أي أن نقوم بتوريث صنف ما من أكثر من أب (*CALCULATOR_WATCH* inherits from *CALCULATOR* and *WATCH*) وتعتبر لغه Eiffel داعم جيد لهذه التقنيه .الجدير بالذكر أن لغه C++ تدعم الوراثه المتعدده بشكل جيد أيضا (للأمانه العلميه قد تجد في بعض الكتب وملفات بعض مترجمات ال++C تخويفا من إستخدامها نظرا للمشاكل التي قد تحدث) ، يعتبر العديدون أن Eiffel تدعم هذه التقنيه أفضل بكثير من ++C . كذلك Payhon تدعم الوراثه المتعدده ، في حين أن كل من Java و Delphi و SmallTalk و #C لاتدعم هذه التقنيه ، ولكن (Java و Delphi و #C) يؤمنون دعم للواجهات Interfaces ممايؤمن القدره على القيام بما تقوم به الوراثه المتعدده ويسد الثغره التي خلفتها الوراثه المفرده ، ولكن بهذه الحال سنكون قد خسرنا إمكانيه إعاده الإستخدام (Re_Use) وسنضطر إلى إعاده إشتقاق الصنف كل مره نريده بها .

ويقال أن للوراثه المتعدده كميه من المشاكل ، وينظر لها أنها صوره ايجابيه من قبل بعض المبرمجين، ومسببه للمشاكل من وجهه نظر الآخرين.

ولكن الشيء الأكيد أن الوراثه المتعدده والمتكرره ستستحضر العديد من الأفكار الأخرى إلى الذهن كالأصناف الوهميه والتي ليس من السهل السيطره عليها، و مع ذلك فهي تعتبر قويه جدا .

يقول أحد خبراء Eiffel : "لاتصدق من يقول لك أن الوراثه المتعدده صعبه أو تسبب أخطاء ، هذا صحيح فقط في اللغات التي تدعمها بشكل سيء" .

على كل حال ماهو تفسير إستغناء مايكروسوفت عن دعم الوراثه المتعدده في منصه NET. ، لماذا لاتدعم #C الوراثه المتعدده ؟ ولماذا لاتدعمها لغه من الوزن الثقيل مثل Java ؟

لاأعرف ربما أكون مخطئا ولايكون العيب من هذه اللغات . بل هو مجرد قوه إضافيه تتمتع بها اللغات الداعمه للوراثه المتعدده ، طوبى لمبرمجي ++C على هذه الجوهره الفريده .

نموذج الأصناف الساكن مقابل نموذج مرجعيه الغرض :

من الفروق المهمه بين لغات البرمجه الغرضيه هو كيفيه تمثيلها للكائنات في الذاكره. فالبعض يستخدم المكدس أو الكومه و يمثل الكائنات بطرق ساكنه (Static) .وفي هذه اللغات سيكون للكائن جزء خاص من الذاكره كما هي حال لغه C++ أو Eiffel. وبالتالي فإن تعريف متغيير من صنف ما سينشيء تلقائيا منتسخا من هذا الصنف ويحجز له الذاكره مباشره .

توجد لهذا التمثيل بعض الحسنات فهو يحسن مقروئيه الشفره ، ويمكّن المترجمات من التقاط الأخطاء في زمن التصميم قبل تحولها إلى مشاكل زمن-تنفيذ كريهه ، وهو أسهل وأسرع في العمل ، ومن المهم أن نتذكر أن السهوله دائما تقلل من الوقوع في الأخطاء .

مؤخرا ظهر اتجاه لاستخدام نموذج آخر يدعى نموذج مرجعيه الغرض (Object Reference model) في هذا النموذج يتم حجز الذاكره الخاصه لكل صنف بطريقه ديناميكيه في الكومه (heap) وتكون الكائنات فعليا مؤشرات لهذا المكان بالذاكره. والفكره في ذلك أن تعريف متغير من نمط صنف ما لن يخزن الغرض داخله ، ولكنه يخزن مرجع لموقع الغرض في الذاكره ، أي عباره عن مؤشر (Pointer) يشير إلى موقع خاص بالذاكره حيث يتم تخزين جسم الغرض هناك ، وليس ضمن المتغير نفسه ، لذلك يجب حجز ذاكره للغرض قبل إستخدامه وتحريرها بعد الإنتهاء منه.

لغات مثل Java وPascal و Smalltalk تعتمد هذا الأسلوب حتى لغه #C الجديده إتجهت إلى هذا الإسلوب .

ومع أن هذا الأسلوب يزيد العمل على المبرمج لكن له مزايا متعدده تزيد من التحكم بالأغراض وكذلك تعطي قدره كبيره على إداره الذاكره . ومنها :

- **يسمح بتقنيات أفضل في نسب الأغراض :**

بما أن المتغير الذي يحوي الغرض يدل فقط على العنوان الأساسي للغرض في الذاكره ، فإننا نستطيع تعريف أكثر من متحول تدل جميعها على الغرض نفسه ، فإذا قمنا بنسب متحول جديد ما من نفس الصنف إلى آخر تمت تهيئته فإننا نستطيع التعامل مع المتحول الجديد للدخول على الغرض نفسه دون أدنى مشكله .

- **يسمح بتمرير الغرض في الإجرائيات :**

أي تستطيع تمرير الغرض بهذه الطريقه على شكل بارامتر خاص بتابع ما ، ونعامل الغرض كمتحول عادي ونمرره إلى مناهج تقوم بمعالجته والتعديل فيه من مكانه .

مثلا نفرض إجرائيه لها بارامتر وحيد من الصنف TButton ، نمرر لها زر ما فتقوم بتغيير إسمه مثلا ، *المثال بلغه دلفي :*

```
procedure ChangeCaption (B: TButton);
begin
  B.Caption := B.Caption + ' was Modified';
end;
……
// call...
ChangeCaption (Button1)
```

المتحول الذي قمنا بتمريره كزر أعطى عنوان الذاكره للإجرائيه . وهذه بدورها دخلت إليه وقامت بالتعامل معه مباشره .،

هذا يعني أن الغرض تم تمريره بالمرجع بلا إستخدام الكلمه المفتاحيه Var التي نستخدمها بالحاله العاديه للمتحولات، وبدون أي من التقييدات الأخرى التي تفرضها حاله التمرير بالمرجع (pass-by-reference) .

- تسمح بتحكم أكبر في إداره الذاكره :

بما أننا قبل إستخدام الغرض يجب أن نقوم بحجز الذاكره له (Create) ، وبعد إستخدامه يجب أن نقوم بتحرير الذاكره (Destroy) . هذا يعني أننا نملك التحكم الكامل بلحظه حجز الذاكره ولحظه تحريرها ، وبالتالي نستطيع إداره الذاكره بشكل ديناميكي ولن نخسر ذاكره تطبيقنا حتى نحتاج لإستخدام الغرض ، وبعد الإنتهاء منه يمكننا تحريره مباشره لإعاده إستثمار ذاكرته .

ملاحظه:

اذا كان نموذج تمثيل الكائنات باستخدام المراجع (object reference model) يستدعي عملا اضافيا من المبرمج فتذكر ان في لغه C++ عاده ما تستخدم المؤشرات والمراجع للتعامل مع الكائنات ، وبهذا تحصل على خاصيه تعدد الاوجه في حين أن نموذج تمثيل الكائنات بالمراجع يستخدم المؤشرات بشكل افتراضي ويقوم بعمل جيد لاخفاء التفاصيل عن المبرمج.

في لغه الجافا استثنائياً لا يوجد مؤشرات لكنها موجوده في كل مكان ولا يستطيع المبرمجون التعامل مباشره معها لذلك فانها لا تشير إلى أماكن عشوائيه في الذاكره لاسباب أمنيه.

الإداره الآليه للذاكره ومجمع النفايات garbage collector:

والمقصود آليه استعاده الذاكره التي تم تخصيصها لكائن ما بعد الانتهاء منه.

مساحات الذاكره الساكنه المحجوزه بواسطه المكدس في لغه ++C يمكن إسترجاعها بسهوله ، في حين أن إسترجاع المساحات المحجوزه بشكل ديناميكي عاده ما يكون معقد إلى حد ما .

- إنطباعي الأولي أن إستخدام النموذج المرجعي للكائنات سيجعل العمليه أكثر صعوبه وتعقيدا .

حيث أن ++C لاتملك مجمع نفايات (garbage collection) في المنفذات التجاريه الشائعه .

GC ليس ميزه قياسيه من مجموعه مزايا ++C ، ويعتبر التعامل مع حجز وتحرير الذاكره عملا صعبا ويتطلب بعض العناء والتعقيد ، وهو بيئه خصبه لتوليد أخطاء إداره الذاكره . مجمع النفايات ليس جزء من ال++C القياسيه ولكن يمكنك الحصول على العديد من خوارزميات مجمع النفايات من بعض المكتبات الخارجيه .

#C تعمل ضمن بيئه NET. وهذا يعني وجود العديد من العمليات التي لاتحتاج لتحكم وتعامل مباشر من قبل المبرمج ، وبالتالي التخلص من العناصر غير المستخدمه يتم بواسطه مجمع النفايات (GC) التي تدعمه #C .

الجافا : تملك الجافا Garbage collection ، ليس هناك الكثير ليقال في هذا الموضوع حيث أنها تستخدم خوارزميه خاصه لاسترجاع الذاكره بعد الانتهاء من استخدامها من خلال ما يسمى (virtual machin) وهذا بدون أدنى تدخل من المبرمج ما يعطي سهوله وراحه مثاليين في العمل . لكن لكل شيئ ثمن فقد تحصل على بعض الاخطاء المنطقيه بسببها.

الباسكال الشيئيه:باسكال لا تملك طريقه لتفريغ الذاكره كتلك الموجوده في الجافا. لكن عناصر الدلفي تدعم فكره العنصر الأب الذي يكون مسؤولا عن استرجاع الذاكره التي كانت مستخدمه من قبل كل الكائنات الأبناء. مما يجعل الموضوع بالدلفي اكثر بساطه.

لغات أخرى مثل Smalltalk و Eiffel تملك Garbage collection و إداره جيده جدا للذاكره

الاستدعاء المتأخر (وتعدديه الأشكال) Late Binding and Polymorphism

اذا أعاد صنف ما تعريف منهج تابع لأبيه فيجب على لغه البرمجه أن تعرف أي من المنهجين عليها أن تنفذ عند الإستدعاء . لتمكين ذلك يجب أن يدعم المترجم عمليه الاستدعاء البعيد (late binding) فجمله الاستدعاء في الشفره المصدريه لا تقوم بربط عنوان المنهج وقت الترجمه ، لكنها تنتظر إلى وقت التنفيذ لتعرف أي منهج سيتم إستدعاءه.

C++: في هذه اللغه فإن الاستدعاء المتأخر متاح فقط للمناهج الوهميه (virtual methods) مما يسبب البطيء عند التنفيذ.

Object Pascal : يمكن الحصول على هذه الميزه سواء للمناهج الوهميه أوالديناميكيه ، ويتم ذلك باعاده تعريفها بالكلمه override .

وتعتبر هذه ميزه فريده للباسكال الشيئيه.

java : وهنا جميع المناهج تستخدم الاستدعاء المتأخر، إلا إذا قمت يدويا بتمييزها على أنها مناهج نهائيه (final methods)

الفرق بين الجافا و السي في هذه الميزه فالسي تستخدم الاستدعاء المبكر افتراضيا على عكس الجافا ، انما مرده لأن السي صممت للحصول على أسرع ما يمكن من البرمجه الشيئيه ، والإستدعاء الساكن أسرع في التنفيذ ، ولكنه يعتبر برمجيا محدود أكثر من الأستدعاء المتأخر .

- والباسكال على النقيض من اللغتين الأخريين في هذه الناحيه فهي تسمح بتعريف منهج بناء أو هدام وهمي .

معلومات الأصناف في زمن التشغيل: RTTI Runtime Type Identification/Information

في اللغات الغرضيه التي تدعم فحص الأنواع ، فإن المترجم يقوم بكل مهام فحص الانواع لذلك هناك حاجه قليله لحفظ معلومات النوع او الصنف على وقت التشغيل من ناحيه ثانيه هناك بعض الحالات التي تستدعي حفظ هذه المعلومات مثل (downcast) لهذا السبب فإن لغات البرمجه التي نناقشها تدعم RTTI بشكل متفاوت.

C++:في الاصل إن لغه السي لا تدعم RTTI وقد أضيفت في وقت متاخر بشكل downcast واتاحت بعض معلومات الصنف فتستطيع ان تقارن إذا ما كان صنفين من نفس النوع .

الباسكال : الباسكال الشيئيه وبيئه العمل المرئيه تدعم وتتطلب RTTI. ليس فقط فحص النوع و downcast (باستخدام is و as) بل إن الأصناف تولد RTTI بشكل كبير للجزء المنشور (published) . في الواقع هذه الكلمه تتحكم بجزء من RTTI. كل الفكره من خلف الصفات وبيئه الدلفي وملفات النماذج كلها تعتمد على RTTL .

الجافا : وهي مشابهه للباسكال، فلدينا هنا أيضا أب واحد لجميع الاصناف يساعد على تتبع معلومات الصنف كما ان التحويل الآمن بين الأصناف هو التحويل الافتراضي ولكل صنف منهج يدعى GetClass الذي يعيد بعض المعلومات لوصف الصنف.

القوالب والبرمجه الجينيه : (Generic Programming)

البرمجه الجينيه هي تقنيه لكتابه الدوال والأصناف مع ترك بعض الأنواع البيانيه بدون تحديد. يتم تأجيل ذلك إلى وقت استخدام هذه الدوال والأصناف في الشفره المصدريه. أي أننا عندما نقوم بتعريف صف معين يكون له نوع محدد، لكن من أجل استخدام اوسع لنفس الصف يمكن أن يكون للصف نفسه أكثر من نوع و هذا من خلال استخدام الـــgeneric و هيي خاصيه تتيح أمكانيه أن يكون الصف غير معرف النوع و يمكن استخدامه نفسه تحت أي نوع ..
كل شيء يترك يحدد بإشراف المترجم، ولا يؤجل أي شيء لوقت التنفيذ وأفضل مثال على هذا الأسلوب هو الأصناف الحاويه (container classes).

C++:وهي اللغه الوحيده بين اللغات التي ناقشها التي تدعم هذه الخاصيه ويشار إلى هذه الأصناف بالكلمه Template .
الC++ القياسيه تحتوي على مكتبه ضخمه من قوالب الأصناف تدعى STL التي تدعم أسلوب مميز وقوي بالبرمجه .
C# دعمت هذه الخاصيه ، وقد أضيفت في النسخه الأخيره للNET. وهي Version 2.0 .
الباسكال: لا يوجد لديها أي قوالب والأصناف الحاويه يتم بناءها لتحوي الكائنات من النوع TObject.
الجافا:لا تحتوي قوالب أيضا يمكنك استخدام الحاويات (containers) من الصنف Object .

التجريد Abstraction :

و هو يعني القدره على تعميم الأغراض كأنواع (the ability to generalize an object as a data type that has a specific set of characteristics and is able to perform a set of actions.)

و تؤمن لغه البرمجه الدعم لهذه الخاصيه من خلال الصفوف classes حيث تعرف الصفوف الخصائص و السلوكيات(properties and methods) ثم يمكن استخدامها كنوع data type ..

و تدعم هذه الخاصيه أغلبيه لغات البرمجه SmallTalk, Java , Python, Eiffel,C# Ruby

لكن كل من السي++ و الفيجوال بيسك و البيرل لا تدعمها ..

بعض المزايا الأخرى:

سنتكلم عن بعض المميزات الأخرى التي لم نذكرها بعد لأنها ذات علاقه بلغه واحده فقط.

++C:لقد أشرت إلى الوراثه المتعدده،و الأصناف الوهميه الرئيسه،و القوالب. لكن يوجد في هذه اللغه ما يسمى تجاهل العمليات (operator overloading) و تسمح السي لمبرمجيها بتجاهل الدوال العامه (global function overloading) بل وأكثر من ذلك انها تتيح تجاهل معاملات قلب الانواع لتجعل التحويل من نوع صنف إلى آخر عمليه على مقاس برنامجك.

نموذج الكائنات المتبع في هذه اللغه يتطلب نسخ المناهج البناءه وتجاهل معامل الاسناد وهذا ما لا تحتاجه اللغات الأخرى لأنها تقوم على نموذج الأصناف المشيره.

الجافا:ولديه ميزه فريده بالبرمجه المتوازيه فالكائنات والمناهج تدعم آليه التزامن باستخدام الكلمه synchonized .

منهجين متزامنين لا يمكن أن ينفذا في آن واحد لبناء عمليه (thread) أخرى يمكنك ببساطه أن تشتق من الصنف Thread وتتجاهل المنهج run .

كبديل آخر بالامكان بناء واجهه قابله للتشغيل (وهذا ما يحدث عند برمجه ال java applets) .

من المزايا الأخرى للجافا هو فكره شفره البايت المتنقله (portable byte-code) لكنها لا ترتبط كثيرا باللغه نفسها.

الباسكال الشيئيه:من المزايا الخاصه بالباسكال هو نموذج الأصناف المشيره أو الأصناف المرجعيه أو سمها ما شئت (class references).

والاستخدتم البسيط لمؤشرات المناهج وهي الفكره وراء الأحداث ، والفكره المميزه للخصائص على أنها طريقه الوصول لبيانات الصنف فقد يكون وصولا مباشرا إلى البيانات او عبر اجراءات محدده وبناءاً عليه فإن أي تغيير في طريقه الوصول لا يتطلب تغييرا مقابلا بالاستدعاء.

القياسيـــه Standard :

بعض لغات البرمجه تكون مملوكه لشركه أو لجهه معينه ، ويحق لهذه الشركه تقرير مصيرهذه اللغه وتكون هي المسؤوله عن كل مايتعلق بهذه اللغه . القياسيه من الأمور المفيده جدا لإنتشار لغه البرمجه وإستمراريه دعمها . خاصه أنها تتيح إنتاج العديد من المنفذات للغات القياسيه مما يضيف عامل التنافس والتعدد الذي يعطي خيارات أوسع لمستخدمي هذه اللغه .

C++: هي لغه قياسيه غير مملوكه .
لقد استنفذت ANSI/ISO وقتها لكتابه المقاييس الخاصه بالسي ويبدو ان معظم المترجمات تنحى نحو تطبيق هذه المقاييس، بالرغم من وجود بعض الخصوصيه والتي تتجلى بشكل واضح في MS Visual C++ التي خرجت عن القياسيه في كثير من الجوانب . ورغم أن C++ تملك الإحترام لإنها ANSI قياسيه ولكن إمكانيه الإنتقال الفعلي من مترجم إلى آخر بعيده عن الكمال قليلا .

Eiffel : وهي لازالت مستقره منذ زمن فهي تملك مكتبه_نواه قياسيه (ELKS 95) تتحكم بها إتحاد جمعيات NICE ، وهي الجسم القياسي لـــ Eiffel .

الباسكال الشيئيه: وهذه لغه مملوكه، لذلك لا يوجد لها مقاييس . لكن بورلاند تقوم بتطوير هذه اللغه مع كل اصدار جديد منها.
الجافا : وهي أيضا لغه مملوكه ولديها أيضا علامه تجاريه للاسم ومع ذلك فإن شركه Sun أكثر من راغبه بتسجيلها لمترجمات أخرى . وقد أعطت جهات داعمه أخرى غيرها الحق بإنتاج منفذات لها .

Perl لغه غير مملوكه ، ويمكن أن تجد شفرتها المصدريه ومترجمها مجانا .

المقروئيه Readability وتنظيم اللغه :

وهي عامل مهم لوضوح لغه البرمجه وسهوله إستخدامها . تزداد أهميه هذا العامل في النظم الكبيره ،وفي النظم التي تأخذ وقت تطوير طويل ويشارك فيها كميه كبيره من المبرمجين .

وتعتبر الكثير من الجامعات أن وضوح النص النحوي يعتبر دليلا هاما على تطور وتنظيم اللغه وأدرجته بعض الجامعات في مقاييس بناء لغات برمجه عصريه ، وكلما قاربت لغه البرمجه الكلام البشري العادي والنص المكتوب كلما زادت في دعم عامل المقروئيه ، مما يساعد القادمين الجدد على اللغه والمبتدئين على الإنطلاق بسرعه أكبر .

لاحظ مثلا أن التطور من لغه الآله إلى اللغات العليا ماهو إلا تطور في نحو اللغه ومقروئيتها ، وتبسيط لمفاهيم كتابه العبارات البرمجيه ، حتى في هذه الأيام بدأنا نشاهد طرق تقنيه جديده لكتابه الشفره مثل مناهج UML و مرمزات سكربت تعتمد على الصوره وتستخدم الأشكال بشكل تفاعلي لتبسيط عمليه التكويد ، بحيث تبني برنامج متكامل بواسطه الفأره –(ماعدا البيانات) . الهدف من ذلك هو درجه أعلى من التفاعليه مع فكر المستخدم . ...

من أهم اللغات التي تبدو فيها خاصيه المقروئيه هي لغه Pascal، حيث تملك نموذج نحوي جميل يحوي كميه أكبر من الحشو لنحصل على شفره مقروءه جيدا ، ونظرا لتنظيم الباسكال ونحوها الأنيق فإنها تعتمد في الكثير من الجامعات كبدايه لابد منها لتدريس لغات البرمجه .

Eiffel كذلك تملك نحو واضح وبسيط ومقروء بشكل ممتاز ،

SmallTalk قريبه من الإثنين أيضا ، البعض يراها غريبه بشكل جميل والبعض يعتبرها ذات مقروئيه عاليه ، الذي أعجبني في SmallTalk أن شفرتها واضحه وقريبه إلى الكلام العادي ، فإذا أردنا أن نقول بالإنكليزيه :" *Jack pass the ball to Jill*" فإنها ستكتب في SmallTalk : "*Jack passtheballto: Jill*" أو "*Jack pass : theball to: Jill*" .لقد صممت لتكون سهله التعلم والإستخدام .

Poython كذلك تعتبر لغه برمجه أكادميه سهله القراءه والفهم والتعلم -حتى على مستوى التعلم كأول لغه برمجه- وتتجنب الغموض وتتميز برتابتها الشبيهه برتابه لغه الباسكل .

خذ مثلا لغه البرمجه lisp : إن lisp تحاول أن توفر كل ما تقدمه بقيه لغات البرمجه من ميزات لتكون موجهه إلى جميع الناس (وهذا برأيي لايجعل لها طابعها خاص بها)

وتعتبر من أسوأ اللغات في المقروئيه ربما بسبب اسرافها الغير طبيعي في استخدام الرموز والأقواس وبطريقه غير مفهومه أبداً بالنسبه للوافدين الجدد على هذه اللغه

خذ على سبيل المثال :

Python	y = m*x + b
LISP	(SETQ y (+ (* m x) b))

وكما ترى فالفرق واضح بين هاتين اللغتين ، وتعتبر اللغه lisp لغه صعبه جداً للقراءه والتعلم .

C++ تملك بنيه نحويه معقده ، بلا شك ليست مثل Lisp ، ولكنها تبدو كأنها لغه من عصور قديمه لايمكنك دائما فهما مهما بدت السطور مألوفه لك ، تركز C++ بشكل أساسي على الشفره المختصره وتحاول أن تكون عمليه قدر الإمكان لذلك من المألوف أن تجد رموز وعبارات مختصره في شفرتها أكثر من وجود كلمات مفهومه ، لاحظ إستخدام الأقواس {} بدل Begin و End في باسكال مثلا ، أو لاحظ إستخدام المعاملات && أو || بدل And و Or . أو حتى دوال الدخل/خرج مثل <<cin و >>cout الخ .. كما أنك ستجد عده طرق لكتابه نفس البلوك من البرنامج ، مما يربك القادمين الجدد على اللغه ويصعب عليهم فهم شفره غيرهم . ربما يقول البعض أن ذلك يعتبر مرونه عاليه ..... أقول لكل هؤلاء أن تقوم بأكثر من شيء بطريقه واحده أفضل بكثير من أن تقوم بالشيء نفسه بأكثر من طريقه . إن هذا الإستخدام المفرط للرموز والعبارات البديله والمختصره يبعد اللغه عن الإقتراب من اللغه البشريه المحكيه ويجعلها لغه ترميزيه أكثر من كونها لغه مفهومه .

مثلا النسخه الحديثه من C++ والمسماه C++/CLI والتي صدرت حديثا كتمثيل كلي لدعم NET. بعد MC++ (Managed C++) ((والتي كانت أول Visual C++ .net تصدر)) . تحوي تغييرات كثيره تماشيا مع المواصفات العصريه ولاسيما المقروئيه والتنظيم .

لاحظ مثلا هذا الكود البسيط لبرنامج HelloWorld مكتوب بـــ C++/CLI :

```
using namespace System;

int main()
{
Console::WriteLine("Hello World");

return 0;
}
```

إنه قريب جدا من كود الC++ العاديه ، ولكن أول مالفت نظري فيه إستخدام الدله WriteLine الموجوده داخل الفئه Console . لاحظ وضوح هذا الأمر وطوله مقارنه بتعليمه cout القديمه ، حتى أنه أوضح وأطول من pascal التي تكتب Writeln .

يمكن إستخدام الأمر القديم Cout ولكن لاينصح بذلك لإنه لن يستطيع التعامل مع بعض الأنواع الجديده في الدوت نيت ، لذلك يستخدم للتوافقيه الإرتجاعيه مع الشفرات القديمه بالدرجه الأساسيه . ماقصدته من هذا أن الشعور بأهميه

المقروئيه العاليه غدا أمرا بديهيا في وقتنا الحاضر ويتنامى الإهتمام مع التطور في تقنيه المعلومات ، وهذا مايفسر إتجاه مطورو ++C نحو كود أكثر مقروئيه .

Java تملك نحو مشابه للغه ++C ، ولكن أفضل منه .
#C لغه متطوره وعصريه ، أتجهت أكثر لمراعاه تنظيم الشفره و قواعد النحو أكثر من الإختصار في الشفره الذي كان سائدا في ++C . كما أنها أصبحت تلتزم أكثر بالقواعد وتكتب الشفره بطريقه واحده ، لاحظ مثلا القيود التي فرضتها على قلب الأنواع والتي كانت ++C السابقه تتساهل معها : حيث لايوجد تحويل مباشر من Int إلى Boolean في #C وبالتالي العباره التاليه خطـــــأ "if (1)" .
كما أنه لا يوجد في السي شارب ملفات رأسيه و لا استخدام للماكرو في ظل وجود preprocessor support for conditional code أي يتم التحقق من الشروط قبل الترجمه.
و هذا يجعل اللغه أسهل ، كما أنه يزيد سرعه المترجم ، ويجعل من السهوله أكثر فهم شفره الـــ #C . وكأننا نلاحظ أن #C كلغه مطوره عن ++C فهمت أهميه زياده المقروئيه والتنظيم في نحو اللغه .
لن تجد في #C مثلا إستخدام لـــ "::" أو "->" .

Perl أيضا تعتبر لغه صعبه القراءه بشكل ملحوظ ، رغم أنها يمكن أن تكتب بطريقه أنيقه ولكن الإنطباع العام أنها لغه ضعيفه المقروئيه .

- سرعـــه التكويـــد :

ولكن علينا الملاحظه أن المقروئيه والتنظيم ربما يؤثران على سرعه التكويد ، وبالتالي لغه مثل ++C ذات مقروئيه ضعيفه لإنها تركز على الكود المختصر وتستخدم عبارات مرمزه ، ولكن ذلك يزيد من سرعه التكويد ويسمح للمبرمج بالوصول إلى نتيجه الشفره نفسها بسرعه أكبر ولاسيمل عندما يكون العمل كله كتابه شفره . (طبعا لا نعني أن إنتاجيه ++C عاليه "راجع بند الإنتاجيه") ، قابلت ذات مره خبير ++C وتناقشت معه في عده مواضيع ، ولاحظت أنه لم يعر موضوع المقروئيه إهتماما وأعتبر أن إختصار الكود مفيد بالنسبه له ويجعله ينجز الأعمال بسرعه أكبر .

إن التكويد السريع يزيد سرعه تطوير بعض أنواع البرامج . كتلك المعتمده على كتابه الشفره أكثر من إعتمادها على أدوات ومكتبات اللغه ، مثلا عند كتابه الأصناف ، مما يعطي عاملا مهم لنأخذه بالحسبان أن اللغات التي تملك شفره مختصره ، ولو كانت أقل قابليه للفهم والقراءه فهي ربما تعطي سرعه إضافيه في برمجه وكتابه شفره المشروع لإول مره .

وهذا شيء رائع ويعطي للمبرمج القابليه على تجريب أفكار برمجيه بسرعه أكبر وبلا شك له عده فوائد ، ولكني لازلت منحازا إلى جانب المقروئيه إذا كنا نحقق سرعه التكويد هذه على حساب المقروئيه .

تنظيم اللغه يعني التقيد الصارم بالقوانين مما يحمي المطور من أخطاء كثيره مستقبلا كأخطاء قلب الأنماط ، ويزيد من سهوله تطوير منتج واحد من قبل فريق ضخم بسبب تحسين قدرتهم على فهم شفره بعضهم بعضا .

الكثير من اللغات مثل Python تحاول الإختصار والتسريع في التكويد كذلك ، حيث لاداعي لتعريف المتحولات والثوابت بنفس الطريقه التقليده المتبعه في اللغات الأخرى
SmallTalk تعتبر لغه مختصره كذلك وتركز على مبدأ تصغير الشفره . فهي مثلا لاتحتاج مساحات من الشفره لتعريف المتغيرات كما في اللغات الأخرى ، إنها ليست بحاجه لـــ Type من أجل تعريف كافه أنواع المعطيات أو الصفوف والكائنات.

أمن اللغـــــه Language Safe :

لا أقصد أي من مفاهيم الأمن المتعلقه بالإختراق أو حمايه البرامج .

ما أقصده هنا هو تساهل اللغه مع أخطاء المستخدم والسماح له بإنتاج تطبيقات غير مستقره . اللغه الآمنه تبعد المستخدم عن الخطر قدر المستطاع ، وتوجههُ إلى إتباع الطرق الآمنه أكثر في الحل ، وتعطيه أدويه ذات أعراض جانبيه قليله قدر المستطاع .

لاحظ مثلا الجافا .

جافا جيده جدا من هذه الناحيه ، تسمح لك بتنفيذ الأشياء من وجهه النظر الآمنه فهي تجبرك مثلا على التعامل مع الأغراض في بنيتها غرضيه التوجه الصرفه حتى في برامجك الصغيره ، مما يخدمك في المستقبل في حال توسيع برنامجك . حيث تكون قد بنيت قاعده سليمه وفق قوانين البرمجه الغرضيه التي تسهل إعاده استخدام الشفره وتوسيعها مما يجعلك تزيد حجوم برامجك بثبات وإستقراريه بعيدا عن الأخطاء التي كان من الممكن أن تقع بها لولا أن أجبرتك بالعمل على الأصول من البدايه والتي ربما كلفتك إنتاج برامج غير مستقره .. ،

كما أنها مثلا تحميك من الوقوع في أخطاء المؤشرات الحسابيه ولاتسمح لك بإستخدامها أو تعريفها مع أنها تعتمد عليها داخليا وتستفيد من ميزاتها . نعم تعتبر المؤسرات أدوات فائقه القوه ورائعه للتحكم والسرعه ، ولكن حديثا تم إعتبار المؤشرات أنماط غير آمنه ، وهذا مايفسر الإبتعاد الحالي عن دعم المؤشرات مثلا في بيئه NET. لايوجد دعم كامل للمؤشرات كما كان في Win32 حتى الآن الدعم يتم بطريقه غير مباشره وبحالات خاصه ولا أعرف إذا كانت توجد نظره مستقبليه لدعمها ولكني لا أظن ذلك . كل من VB.Net و #C و Delphi .NET لايدعم المؤشرات ، ولكن توجد التفافات برمجيه تسمح لك بالقيام بأعمال المؤشرات بطرق آمنه أكثر ، بشكل عام بإستخدام الأغراض ، أو مثلا بالعمل تحت نمط Unsafe (كما في #C مثلا) الذي يتيح إستخدام المؤشرات . ولكن نادرا ما يستخدم إلا إذا أردنا معالجه شفره قديمه وتستخدم بدل ذلك المراجع reference وهي مشابهه لما هي موجود في الـــ C++ لكن مع تخطي بعض القصور .

مع أن المؤشرات قويه جدا وسريعه جدا ومهمه جدا ، قامت لغه عريقه مثل Java بالإستغناء عنها ، لصالح مبرمج الجافا .

كما أن جافا تسمح بالتمرير_بواسطه_القيمه فقط (pass-by-value) . وبالتالي تجعل جافا شفرتك أقل عموميه وأكثر أمانا فهي لا تسمح بالتمرير بواسطه المرجع (pass-by-reference) .

اللغه C القديمه تشاركها هذه الخاصه ، لكن كل من C ++ و باسكال يختلفان معها.

- حتى الإداره الآليه للذاكره ومجمع النفايات المتقدم في جافا تعتبر مزايا مهمه من ناحيه أمن اللغه ، فهي تحمي المستخدم من الوقوع في أخطاء إداره الذاكره . بلا شك لها سلبياتها هي أيضا ، فهي تفقد اللغه بعض مزايا القوه والتحكم وقد تسبب لها بطيء إضافي في التنفيذ، ولكن النتيجه أن تطبيقات مبرمجي جافا خاليه من أخطاء المستخدم في إداره الذاكره إلى مدى بعيد مقارنه بمبرمجي C أو ++C .

دلفي تملك نظره مشابهه ولكنها مختلفه عمليا عن جافا . فهي تعتمد مبدأ تغليف إحتياجاتك من نظام التشغيل بإدوات خاصه – (تذكر الكميه الكبيره من الأدوات الموجوده في شريط أدوات دلفي) – وهذا يجعلك تحل المسأله من وجهه نظر دلفي نفسها وبطريقتها الخاصه ،مما يزيد من أمن برنامجك ضد الأخطاء التي قد ترتكبها ، أو ضد نقص في معالجتك لكامل الحالات البرمجيه .

كما أن دلفي أستفادت من النحو الدقيق للغه ObjectPascal . لاحظ مثلا القيود الصارمه التي تفرضها على نسب وقلب المعاملات . الحاله الإفتراضيه أنك ستبرمج على دلفي بالطريقه التي تجعلك هي تبرمج بها ، آخذه بيدك بعيدا عن العديد من المخاطر ، ما لم تقم أنت بطلب دخول أكثر تعمقا ، إنها ليست كالجافا فهي تسمح لك بالدخول في نهايه المطاف ولكنها تجعل الحلول الآمنه هي الحلول الإفتراضيه . وعندما تقرر أن لاتعمل وفق هذه الحلول عليك ان تتحمل مسؤوليه شفرتك .

++C بلا شك هي خاسر مهم لهذه الناحيه ، ودون أدنى تردد أكرر ماقيل سابقا "للقوه ضريبتها" وعلى C أن تدفع ثمن التفصيل الكبير واليدويه اللتان تجبر مستخدمي السي على إتباعهما . مايزعجني في ++C أنها عكس جافا تماما ، فهي تجبرك على فهم المؤشرات والتعامل معها عندما تريد كتابه برامج مصفوفات مثلا ؟؟؟؟ لاحظ الفرق ، لاتريد مؤشرات ولكنها تقحمها أمام عينيك دائما . أنا لاأقول أن كل إستخدام للمؤشرات سيولد أخطاء برمجيه من المستخدم ، ولكن على الأقل نسبه من المستخدمين (غير الخبراء) سيقعون بالكثير من الأخطاء ويغرقون ببعض تفاصيل إداره الذاكره إذا قررو التعامل مع مشاريع من الحجم الكبير .

كما أن ++C تزودك بكافه أدوات القوه دون أدنى قيد و شرط .. تخيل أن يستطيع الأطفال في المترل الوصول إلى السكاكين والأسلحه والعبث بها دون رقيب أو حسيب .. النتيجه ستكون كارثيه ما لم يكن المبرمج على قدر المسؤوليه .

إنطباعي عن لغه ++C أنها لغه الخبراء ، ولا أستطيع تخيل تطبيقات مستخدمي ++C لازالو في بدايه الطريق من دون أخطاء .

#C قفزت قفزه مهمه في هذا المجال إذا ما قارناها ب++C القديمه فهي تولد شفره آمنه أكثر منها بكثير ، لن تستطيع في #C إنجاز تحويل غير آمن مثل تحويل قيمه رقميه (Double) إلى قيمه منطقيه (Boolean) ، كل الأنماط ذات القيم (Value Types) يتم تهيئتها إلى صفر وكل الأنماط ذات المرجع يتم تهيئتها إلى Null أوتوماتيكيا من المترجم .
ملاحظه: على كل حال لاتزال NET. مشوبه بكثير من الأخطاء والثغرات الأمنيه ، ونحن عند وعد مايكروسوفت بحل كل ذلك .

لغه Perl لغه جميله وجيده ، ولكنها ببساطه تسمح لك أن تكتب شفره بغايه السوء بنفس البساطه التي تكتب بها شفره أنيقه ، وهي مثل ++C يمكن أن تتساهل مع أخطاء المستخدم ويمكن أن تسمح له بإنتاج شفره غير آمنه .
يقول (Larry Wall) منتج Perl : " The very fact that it's possible to write messy programs in Perl is also what makes it possible to write programs that are cleaner in Perl than they could ever be in a language that attempts to enforce cleanliness " .

Pythono : لغه متساهله كثيراً مع أخطاء المبرمج . لدرجه أنها لن تظهر لك أي تنبيه في حال لم يتم تعريف التابع أو الحقل أو المتحول ، أو عند تمرير بارامتر غير صحيح للتابع أو أي شيء آخر وبذلك فإنه لا سبيل لإكتشاف الخطأ إلا بالوقوع به عند الأمر بتنفيذ البرنامج .
مع العلم أن بويثون يؤمن معالجه جيده جداً للإستثناءات وذلك في زمن التنفيذ .

Lisp لاتتسامح أبداً مع أخطاء المبرمج وسوف يظهر لنا تنبيه لجميع الأخطاء النحويه على خلاف بويثون وذلك في زمني التصميم والتنفيذ .

وصول منخفض المستوى low-level access :

يقصد به قدره لغه البرمجه على إنجاز وظائف غير تقليديه مع العتاد أو قدرتها على الوصول بمستوى منخفض إلى تفاصيل نظام التشغيل وتفاصيل المشغلات والمنافذ والمكونات .

أمور كثيره تلعب دورا في تحديد قدره لغه البرمجه على الوصول المنخفض . ومنها دعم أساليب الولوج المنخفض مثل دعم كتابه شفرات بلغه الأسمبلي ضمن اللغه (الأسمبلي المضمن InLine Assimpler) ، كذلك الدعم الجيد للمؤشرات لا يقل أهميه عن سابقه ، بالإضافه إلى تكوين اللغه بشكل عام وآليتها بتنفيذ المهام ، هل تعتمد التفصيل والشموليه ، أم أنها تدعم التقنيات من مستوى المستخدم نفسه ، أو مايسمى القشره .

أيضا مره أخرى تفوز ال C/C++ ، ..
طبعا تملك السي وصولا منخفض المستوى إلى مناهج ودقائق العتاد ! ، أصلا نظام التشغيل مبني عليها بشكل شبه كامل (بعض الترقيعات بلغه Assimply لاتضر أحيانا) .
الوصول منخفض المستوى في كثير من الأحيان ينظر إليه أنه أهم عوامل قوه لغه البرمجه ، لست هنا بصدد مناقشه ذلك ، فأنا ميال لعوامل أخرى مثل ReUsibility و Extensibility أكثر منه ، ولكن رأيت أن ذلك يستحق الذكر والأخذ بعين الحسبان .
و++C تملك كل العوامل التي تجعلها لغه مناسبه لتنفيذ وصول منخفض والمنافسه معها في هذا المجال تنحسر كثيرا .

جافا لاتملك دخولا منخفض المستوى ، فلاتستطيع أن تتعامل مع مواقع ذاكره ، أو تقرأ من منافذ الخ .. ربما منع ذلك من أجل السريه ، ولكنه بالنهايه غير مسموح ولايمكن تحقيقه ، لذلك لايمكن أن تفكر بإن تكتب مشغلات أجهزه في جافا (device driver) ، ربما يمكن الإلتفاف على كل ذلك بإستخدام مناهج محليه ، ولكن تقنيا لايمكني إعتبار أن ذلك يندرج تحت إطار الوصول المنخفض .

دلفي أفضل بكثير من جافا ، فهي تدعم التقنيات السابقه الذكر بشكل ممتاز ، يمكنك كتابه أجزاء بلغه الأسمبلي أو إستخدام المؤشرات بأقصى الحدود ، يمكنك الوصول إلى مختلف المنافذ وتحقيق كثير من الأمور التي لم تعتد عليها ، ولكن للأمانه العلميه نقول أن لغه الدلفي مختلفه في تكوينها عن لغه ++C وأنها ليست جديره أن تصمم برامج مشغلات تقارن ببرامج ++C . وإنما ماتمنحه من الوصول المنخفض ينصب أساسا لخدمه المبرمج في بعض الحالات وهدفه شحن الدلفي

جرعه إضافيه من القوه التي قد يضطر المستخدم إلى كشف الغطاء عنها أحيانا لتنفيذ وصولات محدده . وعدنا إلى نفس المقوله السابقه ، دلفي ليست اللغه الأقوى ولكنها لغه قويه بشكل كافي .

لغات أخرى مثل الـــ Poython تملك قصورا واضحا هنا أيضا . من الأفضل استخدام البويثون لانشاء التطبيقات الخدميه للمستثمر .بمختلف أنواعها مع الأخذ بعين الإعتبار الإبتعاد عن البرامج التي تتعامل مع بنيه الحاسب بشكل مباشر فلغه البويثون لغه عاليه المستوى وغير مناسبه للتعامل مع ركائز النظام والحاسب .(برامج تعاريف الأجهزه مثلاً) .

الســـرعـــه Speed :

يعتبر هذا المعيار من أهم معايير تقيم لغات البرمجه ، ويعتبر من المعايير البديهيه التي تخطر على بال أي منا عند الحديث عن لغات البرمجه . ودائما عند الوصول في الحديث إلى مقارنات بين اللغات الجميع يهتف قائلا ماذا عن سرعه لغه البرمجه هذه أو تلك .

السرعه مفهوم أكبر قليلا مما يبدو عليه للوهله الأولى ، سأحاول أن أشرح نظرتي لمفهوم السرعه .

تقسم السرعه إلى قسمين :

- سرعه تطوير التطبيق ((الإنتاجيه))
- سرعه تنفيذ التطبيق

أولاً : سرعه تطوير التطبيق "الإنتاجيه" Productivity :

إذا أردنا تعريف الإنتاجيه كمعادله رياضيه فالإنتاجيه هي ناتج قسمه النتيجه\الزمن . وكلما أستطعنا تحقيق نتائج أفضل بزمن أقل كلما زادت إنتاجيتنا الصرفه . إنتاجيه لغه برمجه ما تزداد بإزدياد قدرتها على بناء تطبيق جيد ومستقر في زمن أقل ، ويدخل في تقييم الإنتاجيه كل من

- **جوده التطبيق** (سرعه وثوقيه إستقراريه حجم محموليه الخ ...)
- **زمن التنفيذ**

إذا أردنا سرد العوامل المؤثره على إنتاجيه لغه ما بناء على ماسبق ، ربما كانت النتيجه كالتالي :

١ - سهوله اللغه Simple & Easy :

لماذا علينا أن نختار لغه برمجه قويه جدا غير قادره على بناء التطبيق الذي نريده بسرعه كافيه ؟

قوه لغه البرمجه كثيرا ما تتولد نتيجه التفصيل الكبير في الدوال والمكتبات ، حيث تتيح لنا هذه اللغات العمل بمستوى أدنى لنحصل على جرعه من التفصيل الشديد وبالتالي التحكم الكامل . وعندها سنجد حلول لأي حاله تعترضنا ، ولكن

للأسف نكون قد خسرنا السهوله . وعندها علينا أن نلاحظ أن هذه الصعوبه المرافقه للغه البرمجه القويه التي تزيد بشكل ملحوظ من خطوات العمل و تعقيده تؤثر من جانبين :

- زمن تطوير أكبر ، مما يخفض الإنتاجيه .
- مزيد من الوقوع في الأخطاء مما يخفض مستوى جوده النتيجه ، حيث جرت العاده في اللغات التي تتطلب شفره طويله ومعقده أن المستخدمين يقعون في كميه كبيره من الأخطاء البرمجيه ، والتي تطلب سنوات من الخبره للتغلب عليها ، في حين أن اللغه الأسهل و الأكثر إنتاجيه تسمح للمبتدئين ببناء تطبيقات أكبر وأكثر إستقرارا بسرعه كأنهم يعملون منذ سنوات ولديهم خبره جيده .

إذا كانت لغتان قادرتان على بناء تطبيق ما وإحداهما تحتاج زمن أكثر لتحقيق ذلك ، فاللغه الثانيه هي الأكثر إنتاجيه وهي الأفضل لهذا المشروع لإنها توفر الوقت والجهد والمال .

الإنتاجيه تعتمد مبدأ أساسي في العمل : **((إتبع أبسط وأسهل طريق يحقق كامل الهدف))** . لاحظ كلمه "كامل الهدف" ، لايجب أن نتبع طرق سهله ولكنها غير قادره على تحقيق متطلباتنا على أكمل وجه .

وهذا المبدأ مستوحى من الأفكار التي تقول . لماذا علينا أن نضيع الوقت بكتابه الشفره ، يجب أن يضيع الوقت على التفكير بالبرنامج وتكون كتابه الشفره أسهل مايمكن .

الخلاصه :

لاأريد أن أقول أن القوه والإنتاجيه متعاكستين ، وهذا فهم خاطىء لكلامي . ولكن ماأريد قوله أن الصعوبه والإنتاجيه متعاكستين . إذا كانت لغه البرمجه التي تعتمدها تحقق القوه عن طريق الصعوبه والتعقيد فهي أقل إنتاجيه . ومن الطبيعي على كل حال أن تكون اللغات التي تهتم للقوه بشكل أساسي أقل إنتاجيه من غيرها ،لإنها بحاجه لتفصيل كافي. ولكن ذلك ليس حاله عامه .

تعتبر كثير من اللغات مثل جافا و ++C ضعيفه الإنتاجيه . وهذه اللغات عندما بنيت تم تصميمها لتكون أكثر قوه أو أكثر محموليه مثلاً ، وبالتالي أهمل إلى مدى بعيد موضوع إنتاجيه اللغه ،.

لغه Visual Basic تعتبر لغه سهله ، وهي لغه عاليه الإنتاجيه نظرا لسهولتها وبساطتها. مشكلتها أنها لاتقدر على التعامل مع المشاريع الكبيره ولكن في حال كان المشروع مناسبا لها فإنها تعتبر ذات إنتاجيه مميزه ، وهذا هو سر إنتشارها الواسع ، ولاأرى في ذلك عيبا أبدا . اللغه مناسبه لمشاريع دون أخرى ولكنها تخدمك بشكل أنيق في المشاريع التي تناسبها .

٢ - قابليه إعاده الإستخدام ReUsibility .

وهي تمثل قدره اللغه على الإستفاده من موارد سابقه عند الحاجه لها ، مثلا : الوراثه من الأصناف تسمح لنا بإعاده إستخدام شفره صنف سابق كنا قد أعددناها في الصنف الجديد دون الحاجه لإعاده كتابتها من جديد مما يضاعف سرعه العمل ويختصر تكرار الشفره ويسمح لنا ببناء نموذج متطور عن سابقه بزمن قياسي . لغتات OOP بشكل عام داعم أساسي لهذه التقنيه ، تختلف بدعمها لها بحسب نوع وطريقه تمثيلها لمفهوم الـــ OOP . لمزيد من المعلومات راجع بند الـــ OOP .

٣ - التوسعيه Extendibility .

وهي القدره على التخاطب مع نظم وتقنيات مختلفه مما يسمح بالإستفاده من الميزات المتوفره بكل نظام بدلا من إعاده بنائها ، كما يسمح دعم التقنيات المختلفه بتحقيق أعمال معقده ومتطوره بمستوى أعلى من السرعه والأمن .
سأشرح ذلك :

- **القدره على التخاطب مع نظم مختلفه :**

تسمح بالتكامل بين هذه النظم ودمج الميزات المهمه في كل نظام . لايجب أن تكون لغه البرمجه منطويه على نفسها وغير قادره على التواصل مع لغات ونظم أخرى .وكثيرا ما نحتاج بناء برامج متكامله تغطي عده جوانب تقنيه ، ولكل جانب من هذه الجوانب نظم عمليه وقياسيه قادره على التعامل معه مثل نظم إداره قواعد البيانات وخدمات الويب و مشغلات الملتيميديا و نظام التشغيل الخ ....

مثلا تملك دلفي دعم خاص في شريط أدواتها لقواعد بيانات Dbase ، مما يعطي إمكانيه هائله لإداره هذا النوع من قواعد البيانات ، حيث يستثمر تطبيقنا ميزات Dbase بشكل أمثل ويحقق لنا أفضل أداء تقدر Dbase عليه ، كما أنه يسمح لنا بالتحكم الكامل بنظام إداره قواعد البيانات العلائقيه هذا ، ويضيف إمكانيات الإداره التي لاتستطيع أدوات أخرى إضافتها دون أطنان من الشفره .

كل هذه الميزات يمكن تحقيقها يدويا دون الحاجه للتكامل مع نظام Dbase ولكنها ستأخذ شهور من البرمجه وستكلفنا كميه كبيره من الأخطاء أي وببساطه ستسلب الإنتاجيه العاليه منا وستصبح خيارا سيئا عندها .

الإنتاجيه في هذه الحاله تعني أن المبتدء الجديد على اللغه يستطيع بناء برنامج قواعد بيانات يعتمد على Dbase ويحقق أداء أمثليا وأقل نسبه ممكنه من الأخطاء خلال زمن تطوير صغير نسبيا .

والخبير يستطيع تحقيق نتائج تامه من الجوده والموثوقيه بزمن قياسي . وبأقل تكلفه ممكنه للجهد والمال .

كذلك يمكن لدلفي التخاطب مع لغات برمجه أخرى . فهي تتشارك مع لغه C_Builder بمكتبه المكونات (VCL) ، كما أنها يمكنها إستخدام وبناء مكتبات ActiveX (ملفات OCX) التي تدعمها لغات Visual Studio مثلا . حتى في الإصداره الحديثه من دلفي Delphi9 أصبح بإمكانك كتابه تطبيقات #C داخلها .

لاحظ مثلا تكامل لغات الـــ NET. هذا ما أسميه توسعيه حقيقيه ، حيث أن القدره على التخاطب بين اللغات المختلفه يجعل من السهل تقسيم العمل إلى أجزاء وبناء كل جزء باللغه الأكثر مناسبه له .ومن السهوله بمكان إستخدام كومبونينت من لغه NET. في لغه أخرى منها .

فكلّ لغات VS.Net تستخدم واجهه واحده، مليئه بالأدواتِ التي تُسهّل بطريقه مدهشه عمليّه تصميم البرنامج.. إنّ هذه الميزه تسمح لك بإنشاء تطبيقات تدخل فيها أكثر من لغه برمجه، دون أن تحتاج لفتح أكثر من واجهه.. إنّها واجهه واحده فقط لكلّ المبرمجين.

لغه Python مثلا تملك حوار خاص مع C++ التي كتبت بها بعض أجزاء الPython ، وتسمح Paython بإستخدام شفرات C++ في بعض الأماكن التي تعاني فيها Paython من البطيء من أجل تسريع البرنامج ، ممايعطي سرعه كبيره في تطوير التطبيق (Paython) وسرعه في تنفيذ التطبيق (C++) .وكذلك إمكانيه تطوير وتعديل وحدات كتبت بلغات مثل C و C++ و Java و Fortran .

حتى لغه Perl يمكنها أن تترجم إلى ملفات C سريعه التنفيذ أكثر . كما أنها تملك تكامل جيد مع نظام التشغيل . حيث تسهل من عمليات معالجه النصوص والولوج لتوابع نظام التشغيل .

كذلك الأمر كل من لغتي lisp , smalltalk تسمح لنا بتضمين شيفره بلغه الـــ C .

دعم تقنيات مختلفه Many tech Support:

دعم لغه برمجه ما لتقنيه يعني توفير واجهه جاهزه لإستخدام ميزات هذه التقنيه بشكل مريح . مثلا دعم اللغه لتقنيات الويب ، تدعم C# تقنيات ASP.Net مما يجعلك تصمم تطبيقات ويب تفاعليه بمنتهى السهوله والمتعه ، الدعم الجاهز والمضمن مع اللغه لهذه التقنيات يسمح لك بالعمل مع التقنيه من خلال واجهه مريحه جدا تغلف إمكانيات التقنيه بشكل سهل ، حيث يكفي السحب والإفلات بالفأره مثلا لتصميم واجهات ويب WYSIWYG متكامله وضبط خصائصها دون الحاجه لبرمجه ذلك من البدء وكتابه آلاف السطور.

تقنيات مختلفه وكثيره مثل COM, DCOM, CORBA, XML, SOAP, Web, Databases, الخ ... يمكن أن تكون مدعومه بشكل جيد من لغه البرمجه مما يزيد بشكل كبير القدره على بناء تطبيقات متطوره ، ويخفض الزمن والتكلفه اللازمه لذلك .

٤ - توفر مكتبه أدوات واسعه Wide Component Library :

بغض النظر عن دعم التقنيات المختلفه والتكامل مع النظم المختلفه . تحوي لغات البرمجه مكتبه أدوات تسمح لك بإنجاز المهام المتكرره وتصميم برامج مفيده بسهوله أكبر وبمجموعه أكبر من الخيارات .

أنا أؤمن بالمقوله التي تقول :

" When all you have is a hammer, everything looks like a nail. And when you have a complete tool chest, you can choose the right tool for the job instead of just pounding away"

مكتبه الأدوات هي الترجمه الفعليه للبندين السابقين ، حيث يتم دعم نظم وتقنيات مختلفه بتوفير مجموعه أدوات مناسبه في مكتبه أدوات اللغه .

توفر مكتبه أدوات واسعه تغطي مجموعه كبيره من الإحتياجات يفيد في ثلاث جوانب :

- يوفر الوقت والجهد .
- يقلل من الأخطاء ويزيد من الفاعليه ، لإننا نستخدم شفره قياسيه وضعها أخصائيون وجربت حول العالم لوقت كافي .
- يسمح لنا ببناء أنواع جديده من التطبيقات لم نكن قادرين على بنائها قبل توفر الأدوات بسبب عدم قدرتنا على إنتاج هذه الشفره المعقده المكونه لها .

Perl مثلا تملك مجموعه مكتبات (Module) لفعل أي شيء تريده كالتعامل مع الرجيستري وتصميم الواجهات ، التعامل مع الملفات ، وإستخدام Win APIs وإلى ما هنالك .

- خلاصـــه :

إنتاجيه لغه برمجه ما برأيي هي أحد عوامل الفصل التي تقرر إمكانيه إستخدامنا للغه البرمجه هذه أو تلك . ولاينكر أحد أهميه وجود لغه برمجه قادره على تحقيق ماتريد في زمن أقل من غيرها .

من اللغات الأكثر إنتاجيه لغه البرمجه دلفي (Borland Delphi) التي تدعم الأفكار السابقه بشكل فريد ، ربما تكون دلفي اللغه الأشهر التي تتميز بالإنتاجيه كصفه أساسيه . فهي منتج بورلاند للتطوير السريع للتطبيقات -RAD– (Rapid Application Developiment) وقد صممت دلفي منذ البدايه تحت رايه الإنتاجيه العاليه . فهي رغم أنها لغه قويه جدا ، ولغه تملك وصول منخفض المستوى وقدره ممتازه على العمل تحت الحطام ، أستطاعت إنتزاع السهوله والبساطه عن طريق فلسفه التغليف بالأدوات . حيث قام مطورو دلفي بتزويدها بكميه ضخمه من

الأدوات والمكتبات القادره على تغليف كميه غير مسبوقه من الإحتياجات البرمجيه ، مما أخفى التعقيد والصعوبه تحت قناع الأدوات والمكتبات الجاهزه التي تستطيع القيام بالأمور الصعبه بجهد أقل . والأجمل من ذلك أن مكتبه الأدوات هذه كلها مفتوحه المصدر ومبنيه بشكل كامل على دلفي نفسها ، مما أعطى مستخدمي دلفي القدره على الإضطلاع على آليه تنفيذ مكون ما وبالتالي الإستفاده منها والتعديل فيها بما يناسبهم . وبنفس الوقت ظلت دلفي قادره على العمل بالمستوى الأدنى متى لزم الأمر .

لغه VB كذلك تعتبر ذات إنتاجيه عاليه ، ولو أنها تختلف مع دلفي بفلسفه الإنتاجيه ، حيث حفظت دلفي خط الرجعه وبقيت محتفظه بمخزون كافي من القوه ، أما VB أعتمدت على البساطه النحويه والسلاسه في الإستخدام ، والتكامل مع ميزات Office ونظام التشغيل .

لغه C++ ذات إنتاجيه ضعيفه ، ورغم الكميه الكبيره من البرامج التي بنيت على C++ إلا أن مواقع الويب والكتب ونشرات المجلات المعنيه تؤكد مرارا أنها لغه ذات إنتاجيه منخفضه ، السبب أن C++ بنيت أساسا من أجل أهداف أخرى فهي تركز على القوه والسرعه وصغر حجم التطبيق ولاتعير بالغ إهتمام إلى موضوع الإنتاجيه التي قد تتعارض أحيانا مع المزايا السابقه .
جافا كذلك إنتاجيتها منخفضه ، إنها مثل الـــ C++ في كثير من الجوانب ولطاما تشابهت معها . وهاهي الجافا تفضل البقاء مع صديقتها C++ في الخندق المعادي للإنتاجيه ، .
ويعتبر الكثيرون أن إنتاجيه جافا أفضل من إنتاجيه C++ في الكثير من الجوانب وهذا هو رأيي الشخصي كذلك .

SmallTalk تعتبر لغه ذات إنتاجيه عاليه كذلك ، لعده أسباب منها :
سهوله اللغه نحويا فهي من أقرب اللغات إلى اللغه البشريه المحكيه .
سرعه تنقيح الشفره .
الإستغناء عن Type وبالتالي عدم إهدار الوقت في عليات التعريف للمتحولات والأغراض .
الواجهه التصميميه المساعده على تصميم واجهات سهله وسريعه .

لغه Paython ذات إنتاجيه جيده ، وتأخذ البرامج المكتوبه بواسطه Paython وقت أقصر بحوالي ٣ — ٥ مرات من البرامج المكتوبه بواسطه جافا مثلا .

ثانيـــاً : السرعه في التنفيذ Executing Speed :

وهي سرعه التطبيق الناتج من لغه البرمجه في تنفيذ التعليمات المرمزه داخله ، في كثير من الحالات تعتبر سرعه التطبيق الناتج العامل الأهم والأول للأخذ بعين الإعتبار عند البحث عن مزايا لغه برمجه ما .حيث يلزمنا في كثير من الأوقات تطبيقات أو أجزاء من تطبيقات يجب أن تنفذ بأقصى سرعه ممكنه لنحصل على أداء أمثل ، مثلا في الألعاب ثلاثيه الأبعاد حيث يضطر مطورو هذه الألعاب في كثير من الأحيان الإستغناء عن بعض المؤثرات البصريه (كالإستغناء عن الظلال والإناره وتخفيف دقه العرض) من أجل الحصول على نسبه جيده لما يسمى FPS (Frame peer second) والذي يعبر عن عدد الإطارات التي تتم معالجتها وإظهارها في الثانيه الواحده . وهذه بلا شك تتعلق بشكل مباشر بسرعه تنفيذ التعليمات ،.

وعلى كل حال سرعه تنفيذ تطبيقات لغه برمجه ما يحتل مواقع متقدمه في مقارنه وتقييم أداء وأهميه هذه اللغه .

تتفاوت لغات البرمجه بشكل كبير في سرعه تنفيذ تطبيقاتها بتأثير عوامل مختلفه تتعلق بفلسفه اللغه والهدف الذي صممت من أجله ، والكثير من اللغات تبدي خصائص ومميزات مهمه أخرى على حساب السرعه .
مثل لغه الجافا ، كذلك لغات منصه NET. التي أنخفض فيها منسوب السرعه مثلا .

لغه C/C++ من اللغات الرائده في سرعه التنفيذ ، دعنا نقل أنها الأفضل في هذا المجال ، لغه برمجه سريعه جدا في التنفيذ مناسبه للعمل بسرعه وفعاليه في حالات ومنصات مختلفه . صممت أساسا وموضوع السرعه كان أحد أهم العوامل التي روعيت وأخذت بعين الأعتبار كأولويه أساسيه إلى جانب القوه الخارقه .
كما لاحظنا في بند الـــ OOP وفي كثير من الحالات تستخدم C الطريقه الساكنه في التنميط ، وهذا الشيء يزيد بشكل فاعل من السرعه على حساب بعض الأمور الأخرى ، ما أردت توضيحه هنا أن C مستعده للتضحيه بالكثير من أجل الحصول على سرعه عاليه ، وبإختصار هذه اللغه سريعه بما فيه الكفايه وتطبيقاتها تشهد بذلك .

لغات NET. الحديثه مثل C# مثلا أو VB.NET للأسف تعتبر أبطأ من غيرها ، البعض يقول حوالي ٣ مرات ، والبعض من ٥-٧ مرات ذهب آخروه إلى أنها قد تكون أبطأ بمعدل ٢٠ مره في بعض الحالات . أي أن تطبيق C# أبطأ بالتنفيذ من تطبيق C أو C++ بعده مرات على المؤكد . وهذه إشاره كبيره على أن السرعه رغم أهميتها الفائقه في كثير من الحالات قد لاتكون العامل رقم ١ في تحديد اللغه المناسبه أكثر لحالتنا .

أنا مثلا إذا أردت شراء سياره (وسيطه نقل) لن أقول أني أريد أقوى وسيطه نقل (شاحنه بضائع مثلا) ولن أقول أنني أريد أسرع وسيطه نقل (سياره سباق باهظه الثمن ، غير إقتصاديه ، لايمكن التحكم بها بسهوله) ، بل سأختار السياره المريحه والعصريه ذات الشكل الجميل والتقنيات الحديثه ، والسياره الإقتصاديه بالإستهلاك والسهله القياده ... ؟؟

ربما يختار مزارع شاحنه ، ويختار أبن مسؤول سياره السباق ... ولكل منا مذهب وكل منا محق فيما يرى .

وهذا مايبرر إستغناء مايكروسوفت عن بعض السرعه في منتجها من أجل تقنيات أخرى مهمه .

لغه Pascal / Delphi تعتبر لغه سريعه بشكل كافي ، إنها ليست أسرع من C++ . ولاتحاول أن تكون كذلك . لكن نسخ دلفيWin32 أسرع من نسخ لغات NET. (أسرع كذلك من Delphi8.net مثلا) . دلفي ليست الخيار الأفضل إذا كنت تبحث عن السرعه فقط ، ولكنها خيار سريع على كل حال ، . يوجد ألعاب ثلاثيه الأبعاد مبنيه على دلفي بالكامل وتعمل بسرعه جيده وتقنيات مقبوله تماما . كما أنها توفر سرعه خاصه في بعض الحالات مثل تطبيقات قواعد البيانات وتطبيقات الشبكات .

SmallTalk سريع في التنفيذ ، يمكن أن تنفذ كتل برمجيه أسرع عده مرات من تنفيذها في لغات شائعه .
Perl تعتبر بطيئه ، حيث تستغرق برامجها فتره حتى تبدأ حيث يجب تحميل مفسر ضخم قبل البدء بتنفيذ أول سطر كتبه المستخدم ، وعلاوه على ذلك إذا كان الرنامج يستخدم مكتبات محمله بشكل ديناميكي فهنا يوجد تأخير إضافي في زمن التشغيل لقراءه المكتبات من القرص .

جافا لغه بطيئه لإنها تعمل على virtual machine وهي فكره قديمه أستخدمت في UCSD Pascal ، Prolog وبعض اللغات الأخرى . تتولى الآله الإفتراضيه عمليه التفسير لكل بنيه تشغل جافا ، مما يعطي الإستقلاليه في التعامل ، وهو سر المحموليه العاليه التي تتمتع بها جافا ، ولكن بالمقابل فإن أي مرحله وسيطيه أو طبقه تفسير إضافيه ستضيف عمل إضافيا وتسبب بطيء إضافي .
يقول Jan Newmarch (Distributed Information Laboratory) في جامعه University of Canberra : إن برنامج مبني على جافا قد يعمل أبطأ ٢٠ مره من برنامج C++ . ولكنه يعقب أن ذلك ليس دائما يشكل مشكله لإنه في برامج ويندوز سيتم تنفيذ كميه كبيره من شفره بناء النوافذ المعموله على C وبالتالي تختلف النسبه .

قرأت في كتاب C++ Programming How To مايلي :

"تعمل C++ بسرعه فائقه وهي في الحقيقه أسرع من جافا بحوالي ١٠ – ٢٠ مره . تعمل جافا ببطىء شديد لإنها لغه byte–code–interpreted language وتعمل فوق منصه آله إفتراضيه . ستعمل جافا بسرعه أكبر مع مترجم JIT (Just–In–Time) compiler)) ولكنها رغم ذلك ستبقى أبطأ من C++ ، وسيبقى تطبيق C++ محسّن (optimized) أسرع ٣ إلى ٤ مرات من جافا مع مترجم JIT ."

ليست Java الأسوأ دائما في موضوع السرعه ، لغه Paython أبطأ من الجافا بشكل عام .

وبكل تأكيد مفهوم البطىء والسرعه يختلف بإختلاف حالات كثيره ويتأثر بعدد من العوامل ، وقد تكون لغه أسرع من أخرى في حاله وأبطأ منها في حاله ، وهذا يتطلب دراسه تفصيليه لسنا في صدد دراستها في هذا القسم .

حجم التطبيق Application Size :

قدره لغه البرمجه على إنتاج تطبيقات صغيره الحجم سريعه التنفيذ كان ولازال من أهم معايير تقييم اللغه ، حجم التطبيقات التي تنتجها اللغه يلعب دورا حاسما في كثير من الحالات . ماأروعها لغه البرمجه التي تشفر الكود الذي تكتبه على شكل برنامج صغير ، يسهل عليك نسخه ونقله ، إرساله بالبريد ، نشره على إنترنت ، يوفر بمساحه التخزين ويزيد بالمحموليه .

حجم تطبيق مكتوب بـــ C/C++ صغير جدا .

اللغات التي تعتمد على منصه ، قد تكون حجوم تطبيقاتها صغيرها ، ولكن لاشيء يضمن لنا توفر نسخه من المنصه عند المستخدم مما يجبرنا في كثير من الحالات لتضمين نسخه من المنصه مع مشروعنا ، على الأقل أول مشروع يحتاج لتنصيب المنصه وستعمل بقيه المشاريع دون مشكله عليها نفسها . وبالتالي قد يعتبر البعض أن حجم المنصه سيضاف إلى حجم التطبيق وهذا يجعل لغات البرمجه التي تعتمد على المنصه ذات حجم ملف كبير نسبيا .

اللغات التي تنشر مكتبات RunTime مع الملف التنفيذي يمكن مناقشتها بنفس المبدأ تماما مثل لغات المنصه ، نحتاج لملفات الـــ RunTime على الأقل أول مره ، ولاشيء يضمن توفر نسخ منها عند المستخدم أو توافق هذه النسخ مع الإصدارات اللاحقه .

smallTalk حجم تطبيقها يتراوح من ١/٣ إلى ١/٢ حجم التطبيق نفسه في اللغات الشائعه .

دلفي ذات حجم ملفات أكبر من C++ بالحاله العاديه ، يمكن خفض حجم الملفات بحيله معروفه في عالم البرمجه وذلك بالإعتماد على توابع API بدلا من بناء كل شيء من الصفر ،

ملف دلفي لايحوي سوى زر واحد سيكون أكبر بكثير من نفس الملف مكتوب بC++ ، ولكن عند زياده حجم المشروع فإن ملف الدلفي لن يزيد كثيرا وستبقى الغلبه لصالح C++ .

على كل حال من أجل إنتاج ملفات Console صغيره جدا لاتعتبر دلفي مناسبه .

دعم المجاري المتعدده Multi Threading :

نقصد بذلك إمكانيه إنشاء مجرى معالجه جديد منفصل عن مجرى جسم البرنامج الأساسي ، مما ينتج سرعه إضافيه نتيجه العمل المتزامن لإكثر من بلوك برمجي . والأهم من ذلك يتم تنفيذ المجرى الجديد بغض النظر عن توقفات المجرى القديم ، فإذا كان تطبيقنا يعمل على مجرى واحد وحدثت حاله عدم إستجابه بسبب إنتظار رد من طرفيه ما فإن برنامجنا كله سيتوقف عن الإستجابه ، حتى الواجهه الرسوميه ستتأثر بذلك وسنحصل على حاله تجمد (Freez) في البرنامج . تقنيه المجاري المتعدده تعالج ذلك ، فإذا قمنا بفصل إجراء قراءه الطرفيه بمجرى مستقل فإن توقف إستجابه هذا المجرى لن يؤثر على مجرى البرنامج الرئيسي بنفس الأثر ، وسيصبح الأمر مشابها لحاله برنامج آخر على سطح المكتب قد توقف عن الإستجابه ولكني لازلت أستطيع الوصول إلى بقيه البرامج ولو حدث بعض البطيء الإضافي الناجم عن إســـتـــنـــزاف طاقه المعالج .

تعتبر هذه الميزه من مزايا اللغات القويه ، وتدعمها الكثير من اللغات مثل C و Delphi و #C و VB.net وJava (يمكنها تحديد أولويه المجاري برمجيا).

Paython تدعم thareads ولكنها ليست بقوه وإتقان دعم Java لها مثلا .

Smalltalk تدعم هذه التقنيه في بعض إصداراتها وليس جميعها .

أما اللغه Perl فإنها لاتدعم المجاري المتعدده .

معالجه الإستثناءات exception handling :

لنفرض أن بلوك برمجي (قطعه من الشفره) يجب أن ينفذ دفعه واحده ، أي أما ان تنفذ كل التعليمات الموجوده ضمنه معا أو لاينفذ أي منها ، مثلا إما ان نسحب مبلغ من الرصيد الأول ونودعه في الرصيد الثاني مباشره أو إذا حدث أي خطأ نتراجع عن عمليه السحب بالكامل ، فلايجوز نتيجه الخطأ أن نحصل على مبلغ مسحوب من البنك وغير مودع في البنك الثاني .

ولنفرض مثلا أن جزء من الشفره يجب أن ينفذ بعد كتله من التعليمات سواء نفذت التعليمات بالكامل أو لا ، مثلا بعد حجز الذاكره لمكون ما يجب تفريغها في نهايه المطاف من الذاكره مهما حدث من إستثناءات وأخطاء في إستخدام هذا المكون ، وسواء نفذت بشكل صحيح أو لم تنفذ - لايجب أن يتوقف التنفيذ في نقطه قبل تحرير الذاكره ، وفي حال حدث خطأ في هذا البلوك فإن التنفيذ لن يخرج من البرنامج قبل أن ينفذ جزء تحرير الذاكره.

معالجه الإستثناءات تحل كل هذه الأنواع من المشاكل وأكثر منها بكثير ، وتعتبر من الأدوات المهمه لكتابه تطبيقات قويه ، كما أنها تسمح لك بمعالجه أخطاء متوقعه على طريقتك الشخصيه بدلا من تركها تعالج عل يد نظام التشغيل .

لم أجد الكثير من الفروقات لتذكر في هذا البند ، مع أن معالجه الإستثناءات من العوامل الغايه في الأهميه ، إنما خلال بحثي عن المعلومات لم أجد نتائج مميزه تختلف فيها اللغات الحديثه عن بعضها البعض في هذه النقطه ...

تدعم معظم اللغات السابقه معالجه جيده للإستثناءات ، ويبدو أنها تستعير من بعضها الكثير وتقلد بعضها بعضا .

دعم الرسوميات والواجهات GUI و graphics :

ربما يعتبر البعض أن هاتين نقطتين مختلفتين .. نعم ربما

ولكن بالنهايه الدعم الجيد لتقنيات الرسوم يسهل بشكل كبير الدعم الجيد للواجهه المرئيه ويبسط تعقيداتها ، ويسهل من إنتاج أدوات مرئيه مناسبه .

جافا سيئه بدعم هاتين النقطتين . فهي لاتملك سوى بعض إجرائيات الرسم مثل drawLine(), drawRect(), وأخرى مشابهه ، وهذه الإجرائيات لها تقييدات كثيره ، (الخط لايمكن أن يرسم سوى بعرض 1pixel وإذا أردت عرض أكبر عليك القيام برسم العديد من الخطوط المتجانبهمهما وجدت حلول لذلك فإن هذه الحاله تمثل تصرف اللغه الإفتراضي)

كما أنها لاتملك دعم 3_D ولاتملك العده الكافيه من الأدوات المرئيه GUI Objects التي ربما يحتاجها برنامجنا .

مزايا ++C السابقه الذكر كالوصول المنخفض والقوه والتفصيل ، تجعل إمكانيات الرسم جيده جدا وتؤمن دوال كافيه إلى حد ما لتصميم برامج متكامله معنيه بالرسم والتصميم الهندسي . مجموعه الأدوات المرئيه في ++VC جيده إلى حد ما ، ولكنها ليست الأفضل . كمالاحظنا من فقره (توفر مكتبه أدوات واسعه)

دلفي جيده في مجال Graphics وتؤمن دوال كافيه وملائمه تختلف بين السرعه والسهوله والقوه يمكنك الإختيار بينها ، وهي لاتقل شأنا عن ++C في مجال معالجه الصور ولو أن الأخيره أفضل منها برأيي الشخصي نظرا للكميه الكبيره من البرامج المتعلقه والموارد والكتب المتوفره والتي تكون عاده عن ++C في حين أنك ستجد كميه أقل من أجل Delphi (حاول أن تبحث عن كتب OpenGL مثلا ، ستجد فارق كبير بين نسبه الكتب والأمثله المتوفره لصالح ++C عنها المتوفره لصالح Delphi)
ولكن دلفي تتفوق في مكتبه الأدوات ليس فقط على ++C بل على معظم اللغات الأخرى ، إنها تملك مكتبه أدوات ضخمه وفاعله ، مفتوحه المصدر . هي سر إنتاجيه دلفي العاليه ، هذا المخزون من الأدوات يجعل من السهل عليك بناء أنواع مختلفه من التطبيقات وستجد كل مره المكتبه الكافيه من الأدوات التي تؤمن لك إحتياجاتك .

لغات MSVisual Studio .NET حققت قفزه في هذا المجال ، ودعمت مكتبه أدوات قياسيه وملائمه لكميه أكبر من التطبيقات ، مكتبه الأدوات الجديده قياسيه لكل لغات MSVS.Net ، وهذا يحقق نقطه مهمه في تكامل هذه اللغات العضوي والمنطقي مع بعضها البعض . الجديد هو في مكتبه +GDI هناك إمكانيّات هائله في مجال الرسم والتلوين تمنحها لك مكتبه GDI+.. يكفي أن تعرف أنّ بإمكانك الآن رسم منحنيات معقّده، وتكوين أشكال مركّبه من مجموعه خطوط ومضلّعات ومنحنيات، وتلوين السطوح بألوان متدرّجه، وتحديد شكل مساحه الرسم، وتحديد درجه الشفافيه، وتدوير الرسوم وتغيير مقاييسها تكبيرا أو تصغيرا.... إلخ.

كما أن مكتبه أدواتها المرئيه جيده بإعتراف الجميع . وتملك ميزات عديده لاتقل أهميه عن مكتبه الأدوات المرئيه في دلفي

يقدم البويثون أدوات رسوميه مميزه تسهل بناء البرامج الرسوميه بشكل كبير
وكذلك يتميز الـــ smalltalk بدعم مثالي للبرامج الرسوميه بمختلف أنواعها وحتى الثلاثيه الأبعاد فهو يمتلك مجموعه مميزه من الأدوات للقيام بذلك كما أنه يمتلك واجهه تصميميه سهله جداً .

الإنتشار ، التسويق و الدعم الفني Marketing and Support :

لاتنغش بورود هذا البند في آخر البنود السابقه ، فهو من أهم البنود التي تؤثر على إنتشار ورواج لغه برمجه ما .
إنتشار لغه برمجه ما يؤثر على توفر الدعم الكافي لهذه اللغه على كافه الأصعده ، وبشكل خاص الإنترت . اللغه المنتشره يمكن أن تجد لها مئات الكتب وألاف المواقع وكميه كبيره من فرص العمل . كل ذلك يسهل كثيرا تعلم اللغه لوجود كميه كافيه من الموارد والأمثله ، كما أنه يشعرك بالأمان عندما تتعامل معها ، حيث ستجد وظيفه وعمل محترم بواستطها وستشعر أنها لغه تستقطب كميه كبيره من الناس وفي حال حدث أي مكروه فإنه سيصيب كل هؤلاء جميعا ولست لوحدك وبالتالي نضمن على الغالب توفر حلول بديله لإعاده تخديم هذه الشريحه الواسعه من الناس .

برأيي أن لغات Microsoft بشكل عام تملك هذه الشعبيه الواسعه ، ليس من الضروره أن تجتمع الأسباب التقنيه لنقول أن لغات MS أفضل من غيرها .. لا أبدا ، إنها لغات الشركه التي تملك نظام التشغيل ، أكبر شركه ، الشركه الأكثر تشعبا . تقف ورائها دول ومنظمات وشركات وهيئات الخ ... مايكروسوفت تلملك الكثير ، على الأقل أنها تملك نظام التشغيل الأكثر إنتشارا وقادره أن تستعبد الناس تحت لوائـــه .
أي شخص يعمل مع أحد منتجات مايكروسوفت يشعر أنه أمتلك مايكروسوفت كلها ، وغالبا مايشعر مستخدمو مايكروسوفت بالأمان .
حيث أن خطر توقف مايكروسوفت عن دعم عملائها منخفض كثيرا ، ولن يحدث مالم تتوقف قبلها شركات كثيره .
كما أن مايكروسوفت بارعه بالتسويق بشكل لايصدق ، وتملك أقسام دعم فني متفانيه بالعمل ، لاحظ شهادات مايكروسوفت المصدقه التي أصبحت بديلا مهما للتقنيين والمختصين ، لاحظ أنك تستطيع بسهوله من معظم يلدان العالم الحصول على هذه الشهادات (في حين أني تعبت كثيرا لإجد بلدان قريبه تقدم شهاده DCP من بورلاند ، حيث أقتصر الموضوع على البلدان الأوربيه الشهيره وأميركا وبعض بلدان الشرق مثل الهند) . هذا التسويق الذكي يزيد من أهميه منتجات مايكروسوفت ، والأدهى من ذلك أن مايكروسوفت تؤمن دعم التكامل بين أدواتها المختلفه ، .
نعم ربما يكون أوراكل أفضل من MS_Sql server البعض يرون ذلك . البعض يرون دلفي أفضل من C# البعض يرون Java أفضل من C++ البعض يرون لينكس أفضل من ويندوز ... ربما أنا لاأقول لا ... ولكن من المؤكد أنك إذا أخترت أن تكون كل الحلول من مايكروسوفت فإنها جميعا ستكسب قوه إضافيه ، لاحظ إمكانيه إنشاء منظومه متكامله من نظام التشغيل وبرامج إداره قواعد البيانات ولغات البرمجه وتقنيات الويب الخ ... مما يجعل حلول مايكروسوفت المتكامله أحد أهم الإختيارات أمامك ، مع أن كل منها لوحده قد لايكون الحل الأمثل ، ولكنها تملك قوه الإنتشار وقوه الدعم الفني ، التي نتحدث عنها الآن .

لغات شركه بورلاند Borland تعتبر من أهم لغات العصر ، وربما تحتل بعد مايكروسوفت المرتبه الثانيه مباشره ، لإنها تقدم عدد من الحلول ولغات البرمجه (أيضا المتكامله إلى مدى بعيد فيما بينها) ماذا عن C Builder و Delphi و J Builder و Kylix و Dbase و FireBird الخ ...

لكنها بالنهايه تحتل المركز الثاني وليس الأول ، وبفارق ملحوظ . إذا حدث يوما وتوقفت بورلاند لن يكون السبب تقنيا أو علميا ، لطاما كانت هذه الشركه المميزه شركه إبداعات وثورات تقنيه . برأيي سيكون السبب ضعف الإنتشار والتسويق أمام عملاق السوق وعدم تملكها الأدوات اللازمه لإذهال المستخدم عن بقيه الشركات . بورلاند ليس بارعه بالتسويق ، وأغرب ماسمعته حولها أن شركه مايكروسوفت تملك نسبه معقوله من أسهم بورلاند . وهذا بالتالي يجعل مايكروسوفت من مالكي بورلاند ويسمح لها بالتدخل في سياستها وهذا مايفسر عدم قيام بورلاند بأي خطوات عدائيه أو ضاره ضد مايكروسوفت مؤخرا ، وإنضمامها تحت جناح مايكروسوفت . ربما يكون لذلك بعض الفوائد ، فقد لاحظنا سرعه بورلاند بدعم منصه NET. (جائزه أول شركه دعمت NET. بمنتجها دلفي ٧ "دعم تقنيه NET. كان موجود منذ النسخه ٦" أي قبل أن تقوم مايكروسوفت بإصدار التقنيه بشكل رسمي بحوالي ٦ أشهر . وهذا حسب ماورد على موقع مايكروسوفت) كيف تفسر ذلك ؟؟ .

ذهبت بعض مواقع الويب مثل الموقع الشهير About للقول بإن تقنيه NET. تعود أصولها لشركه بورلاند قامت مايكروسوفت بشرائها مقابل مبالغ ضخمه ، وأضافتها إلى منصه NET. وخلقت مايسمى غسيل الدماغ الكلي والإنقلاب المسمى NET. ، إذا كان ذلك الكلام صحيحا فإنه أكبر مثال على كون الإنتشار والتسويق والدعم الفني مهم أكثر من المزايا التقنيه للغه البرمجه ، ولو كانت NET. من إنتاج بورلاند لما سمع بها العالم هكذا .

شركه Sun تملك حصه كبيره من الكعكه أيضا مثلها مثل بورلاند ، وربما أفضل منها ولكني فضلت بورلاند لإن لها مجموعه كبيره من لغات البرمجه التي تنتجها ومجموعه كبيره من الحلول ، في حين تركز Sun على لغه JAVA بالدرجه الأساسيه ، ولكن لها رصيد مهم من المنتجات الأخرى ، و كل ماينطبق على بورلاند من الأفكار السابقه ينطبق عليها ، سأذكر مثلا المشكله بين مايكروسوفت وصن ، حيث توقفت مايكروسوفت عن دعم وتضمين منصه JVM في نظم تشغيلها ، وسيضطر المستخدم إلى تنصيبها من الإنترنت مثلا ... لاحظ ذلك مايكروسوفت قادره على لوي ذراع صن لإنها شركه غول كبيره أكثر منها وتملك سلاح فتاك هو عباره "نظام تشغيل مايكروسوفت ويندوز".

رغم تفوق لغات مايكروسوفت تعتبر كل من لغات بورلاند ولغات صن ، خيارا حيا ومهما للغات البرمجه . في حين تنحسر المنافسه كثيرا مع منتجات ولغات أخرى حتى لاتكاد تسمع بها أو بمن يستخدمها إلا بالجامعات والمنظمات اللاربحيه والمراكز ومالإلى ذلك

ناحيه سلبيه لـــ paython مثلا هي قله المراجع الرسميه كالكتب المتعلقه بالغه فهناك على الاكثر بضعه كتب فقط (لاأعرف ربما ثلاث كتب فقط) مهما كانت هذه الكتب ممتازه فهي عامه ، بإمكانك أن تجد في لغات أخرى كتاب كامل يتحدث عن موضوع واحد بكافه تفاصيليه (عن الإنترنت أو API أو الشبكات أو الألعاب أو او ..) ومهما كانت هذه الكتب الثلاث شامله فهي لن تقدم معلومات تفصيليه بنفس مستوى كتاب مخصص .

لكن ومن ناحيه أخرى يوجد دعم كبير لهذه اللغه على الإنترنت من حيث إمكانيه إيجاد حلول للمشاكل والدروس وغيرها عبر مقالات ونشاطات لمبرمجي البويثون أنفسهم.

يمكن أن تجد أرشيفات مجانيه على الإنترنت عن Perl مثلا والتي أصبحت مناسبه من أجل Unix ، كما أن لها توثيق جيد وموسع للتعلم على شكل HTML أو PDF ..

وجدير بالذكر أنه توجد عوامل أخرى عديده تؤثر على إنتشار لغه البرمجه ، خذ مثلا ملائمتها للغات وهيئات مختلفه حول العالم . كدعم اللغه الصينيه والعربيه وغيرها ، وما إلى ذلك من القدره على التراصف لليمين او من الأعلى للأسفل في كتابه النصوص .

الكل يعلم أن تأخر دعم لينكس للغه العربيه كان من أهم أسباب ضعف إنتشاره في أوساط منطقتنا العربيه .(التأخر بسبب العرب أنفسهم الذين تكاسلو في العمل على هذه النظم المفتوحه وتطويرها ، وما لم ندعم اللغه العربيه على هذا النظام لن يدعمها من أجلنا الغرباء)

خلال إضطلاعي على لغات مثل smalltalk أو python أو Power Builder أو أو ..

وجدت أنه في العالم الكثير من اللغات التي لانعرفها و لم نسمع بها ، وغير منتشره في منطقتنا ومن الجدير بنا الاضطلاع على هذه اللغات القويه والمجهوله لا بل تعلمها أيضاً . وربما تبقى المشكله خلف سوء إنتشارها هي عدم دعمها الكامل للغتنا العربيه أو عدم توفر مراكز عربيه تنشر هذه اللغات وتعلمها بشكل جيد .

المحمـــوليـــــه Portability :

١- على مستوى نظم التشغيل .
٢- على مستوى إصدارات نظام التشغيل .
٣- على مستوى نظام تشغيل واحد_أجهزه مختلفه .

أولا : المحموليه على صعيد نظم التشغيل . تعدد المنصات Cross_PlatForm :

هناك بعض اللغات القادره على العمل على عدّه نظم تشغيل ، ومن أبرز هذه اللغات اللغه Java ، والتي تستطيع العمل حتى على أجهزه ذكيه أخرى غير الحواسيب كالموبايلات والساعات مثلا ..
تختلف فلسفه لغات البرمجه بين بعضها لتحقيق هذه الغايه :

أولاً - اللغات التي تعمل على منصات خاصه :

وتكون لهذه اللغات منصه خاصه بها ، تتعامل اللغه معها بشكل كامل بدلا من نظام التشغيل ، وتقوم هذه المنصه بدورها بترجمه إحتياجات اللغه إلى نظام التشغيل . يمكن إعتبار المنصه طبقه برمجيه بين اللغه ونظام التشغيل ، بحيث تلبي إحتياجات اللغه حسب نظام التشغيل الذي تعمل عليه ، وفي هذه الحاله يكون للمنصه نفسها عده إصدارات من أجل نظم تشغيل مختلفه ، مثلا إصداره من المنصه تعمل على لينكس وأخرى تعمل على ويندوز وهكذا ....
وبما أن المنصه طبقه برمجيه لايراها المستخدم ولايتأثر بتغيراتها ، لإنه يهتم فقط بالواجهه المرئيه للغه البرمجه والتي تبقى ثابته ، عندها لن يتأثر المستخدم بشكل كبير بالإصدارات المختلفه للمنصه من أجل كل نظام تشغيل .

ومن هذه اللغات :

١ - جافا (Java) . Multiplatform

تعتبر جافا أكثر لغه برمجه تدعم هذه المحموليه ، وصممت أساسا من أجل مراعاه دعم التطبيقات الموزعه وتطبيقات الإنترنت مثلا - في هذا النوع من الحالات تكون الزبائن تعمل على نظم تشغيل مختلفه ومتباينه وبالتالي من المهم القدره على التخاطب مع هذه النظم المختلفه من لغه واحده ، وقدره هذه اللغه على العمل عليها جميعـــــاً . وهذا هو حال جافا ملكه المحموليه الأولى بين اللغات .

وليس فقط ذلك بل أن جافا تدعم أنواع أخرى عديده من الأجهزه الإلكترونيه وقادره على العمل عليها ، لوجود نسخ من منصه جافا قادره على العمل في هذه الأجهزه ، كلنا نرى تطبيقات الموبايل المبرمجه بواسطه جافا ، أو الساعات الرقميه الحديثه . أو الإجهزه الكهربائيه التي تعمل على وحدات معالجه صغيره ،، (هل تستغرب إمكانيه جافا من العمل على ثلاجه مثلا ؟)

والخلاصه أن جافا لغه تدعم محموليه تطبيقاتها على نظم مختلفه بنطاق واسع .

-لغه Paython أيضا متميزه في هذا المجال حتى أن عشاق Paython يرون أن شفرتها أكثر محموليه من جافا نفسها ، حتى أنك تستطيع تشغيل كود بويزون من منصه جافا نفسها (Java Virtual Machine)

أو تشغيله من الأنظمه التي تدعم JVM (Java Virtual Machine) بإستخدام JPython .

وتستطيع Python العمل على Windows, Linux, Unix, Macintosh, OS/2, Amiga, وأكثر من ذلك بدون تعديل على شفرتها .

لغه SmallTalk تعمل كذلك وفق منطف الـــVM (Virtual Machine) بشكل مشابه لما ورد . والفرق أن SmallTalk تحفظ كل الكود في ملف (Image) واحد فقط ، وتحفظ فيه حال النظام كله (المجاري النشطه ومالإلى هنالك) ،.

تستطيع Squeak العمل على Macintosh,Unix,OS/2,Ms-Dos,Win9x/Nt/2000, and windows CE .

المنصه الخاصه بها تم عملها بواسطه C من أجل سهوله نقلها إلى نظم مختلفه .

٢ - **لغـــات الـــ NET.** التي قدمتها مايكروسوفت (Microsoft) ، تعمل هذه اللغات على فلسفه شبيهه بفلسفه منصه جافا مع بعض التغييرات البينيه ، حتى أنه يشاع أن منصه NET. تقليد لفكره منصه جافا .

أصدرت مايكروسوفت لغات جديده تدعم هذه التقنيه ، مثلا كل من #C و VB.NET كما قامت بعض اللغات الأخرى بدعم هذه التقنيه مثل Delphi .NET و C# Builder.Net من شركه بورلاند ، وغيرها كثير .

Net. لازالت حديثه العهد ، ولازالت الإصدارات منها المخصصه للعمل على نظم أخرى قيد التجريب والتطوير (توجد نسخه NET. تجرب من أجل لينكس مثلا) وبالتالي في حال تمّ إعداد نسخ من المنصه قادره على العمل تحت نظام تشغيل ما فإن هذه اللغات تستطيع على الأرجح العمل على هذه النسخه .

ثانيـــاً – نسخ من اللغه مخصصه للعمل على نظم مختلفه :

وهنا يطرح مفهوم Multiplatform vs CrossPlatform نفسه .

لاحظ Cross تملك دعم خاص لكل نظام مما يمكن اللغه من التعامل مع النظام الحالي بشكل جيد جدا ، حيث توجد نسخه لكل نظام، أما Multi فهي نفسها لكل النظم وبالتالي لايمكنها دعم الأمور الخاصه

وهذا حال اللغه دلفي ، حيث دعمت شركه بورلاند بعد الإصداره السادسه منها (Delphi 6) مكتبه CLX المخصصه لدعم تعدد-المنصات Cross-Platform ، والذي مكّن بورلاند من إنتاج نسخه من دلفي (تمّت برمجتها على دلفي نفسها) قادره على العمل تحت لينكس ، وسمتها Kylix.

أي أن دلفي لاتملك منصه خاصه بها كما حدث مع الحالتين السابقتين ، ولكن يوجد منها نسخه للعمل على نظام مايكروسوفت ويندوز ، ونسخه للعمل على نظام لينكس .

فكره CLX هي الإبتعاد قدر الإمكان عن إستخدام توابع API النظام ، لإن هذه التوابع لن توجد في نظام آخر (Api ويندوز غير موجوده في لينكس) ، لذلك على دلفي تضمين كل شيء في الملف التنفيذي نفسه ،(وهذا هو سبب عدم صغر حجم ملفات دلفي) .

نتيجه ذلك أننا إذا بنينا برنامج على CLX_Delphi ضمن ويندوز ، فإننا سنستطيع فتح شفرته مباشره من كايلكس ضمن لينكس وإنتاج نسخه من المشروع تعمل تحت لينكس .

أي شفره واحده _ ملفين تنفيذيين : الأول لويندوز الثاني للينكس .

ثالثـــاً – محررات مختلفه تدعم نظم مختلفه :

حيث تقوم شركات ومنظمات مختلفه بدعم النحو (Syntax) الخاص بلغه برمجه ما ، مثلا تستطيع أن ترى محررات للغه C/C++ موجوده على لينكس ، ومحررات من جهات أخرى موجوده على ويندوز . هذه المحررات ليست نسخه من المحررات السابقه بل هي منتجات جديده تدعم شفره لغه برمجه ما . ولاعلاقه لها بالمحررات السابقه سوى بالتشابه النحوي للغه .

ثانيًا : المحموليه على صعيد إصدارات نظام التشغيل المختلفه :

تظهر في كثير من الأحيان إشكالات في عمل البرامج على إصدارات مختلفه من نظام التشغيل .

مثلا Win 95 و Win 98 و Win 2000 و Win XP .

كيف أضمن أن التطبيق الذي يعمل على Win 95 قادر على العمل على Win XP ؟ وهل للغه البرمجه دور في ذلك .

،،،،، في الحقيقه نعم

- لغات البرمجه التي تعتمد المنصات ستكون قادره على ذلك بالتأكيد لإنها تتعامل مع منصتها ، ولكن ستكمن المشكله بالمنصه نفسها ، فإذا لم تتمكن المنصه من العمل على الإصدارات الجديده يجب إنتاج إصداره جديده من المنصه تدعم ذلك .

في هذا المضمار تعتبر جافا حتى الآن أفضل من NET. من ناحيه توافقيه المنصه مع الإصدارات المختلفه .

- اللغات التي تستخدم الحد الأدنى من توابع API النظام هي الرابح الأكبر في هذا المجال .

C++ ستحتل مرتبه متقدمه في ذلك ، لإنها تعتمد على توابع اللب (Core) والتي نادرا ما تتغير لإنها أساس نظام التشغيل من المستوى الأدنى ،

VB سوف تتراجع كثيرا ، لإن فلسفتها المبنيه على أساس المفسر (Interrupter) تقوم بكميه كبيره من الإستدعاءات والتي قد تحوي كميه كبيره من توابع القشره (Skin) ، وهذه عاده ما تتغير كثيرا وتحدث فيها تطويرات دائمه . وهذا يولد إحتمال أكبر أن لايتوافق تطبيقنا مع إصدارات مستقبليه من نظام التشغيل .

تعتمد برامج الملتيميديا على كم كبير من توابع القشره ، لذلك كثيرا ما نلاحظ الحاجه لتنصيب نسخه محدثه من البرنامج لتعمل على نظامنا الجديد .

الملفت للنظر في هذه المقارنه أن دلفي ستحقق مرتبه متفوقه في هذا السبق ، السبب الأساسي كما ورد سابقا هو مبدأ دلفي في العمل القائم على أساس " نظام التشغيل عدو لذلك لا أحتاج منه شيء وأستطيع بناء كل ذلك بنفسي " . المقاربه أدبيه إلى حدما فلا بد لدلفي من التعامل مع نظام التشغيل شائت أم أبت ، ولكن من المؤكد أنها تقتصد قدر الإمكان في ذلك .

وسأورد هنا مثالا من تجربتي الخاصه يوضح تعلق اللغه بالنظام بالنسبه للغات السابقه :
عندما نبني تطبيق على VB تحت Win Me ، ونقوم بتشغيل التطبيق تحت Win XP . فإن التطبيق يتصرف بشكل إفتراضي مثل تطبيقات XP الأساسيه ، ونلاحظ هنا أن أزرار التطبيق تصبح كلها جذابه ومضيئه مثل أزرار XP نفسها ... أي أنها أخذت شكل XP .

عندما نكرر نفس التجربه مع دلفي ، فإن أزرار دلفي تبقى كما هي تماما عند تشغيل التطبيق تحت XP ، ولاتأخذ الشكل الجذاب والمضيء تلقائيا ...
((تسمح دلفي بالحصول على شكل XP بوضع الأداه XP_Man على الفورم ، ولكن بالنهايه لم تتم العمليه بشكل تلقائي)) . لماذا :
تعقيب :
VB تعاملت مع الصنف زر Button الموجود في نظام التشغيل ، وعندما تغير نظام التشغيل تغير صنف الأزرار في VB تلقائيا لإنها تستمده من النظام .
Delphi لا تتعامل مع الصنف Button الموجود في النظام ، بل تقوم ببناء أصنافها الخاصه . يمكن ببساطه الإضطلاع على شفره الصنف TButton الذي تزوده دلفي ، وفي حال قمنا بأي تغيير في هذا الصنف ستتغير أزرار تطبيقاتنا بالتوافق مع هذا التغيير ، حتى أننا يمكننا أن نلاحظ شفره رسم الزر على الفورم والتعديل بها ، مثلا لجعل لون الزر الإفتراضي أحمر .
أي أن تطبيقات دلفي تولّد الزر ولا تأخذه من النظام .

ميزه / سيئه : تعتبر التقنيه السابقه التي تحدثنا فيها عن دلفي ميزه مهمه ومفيده من مزايا دلفي ، ولكن لها آثارها السلبيه ، عدم إعتماد دلفي على النظام يعني أنها لن تستطيع دعم آخر تحديثات النظام وتقنياته بنفس سرعه اللغات الأخرى ، وسننتظر دلفي في كثير من الأحيان حتى الإصداره المقبله حتى تدعم تقنيه جديده . وهذا ما يفسر الكم الكبير من الإصدارات التي تنتجها دلفي (من ١٩٩٦ حتى اليوم توجد ٩ إصدارات دلفي)

ثالثا : المحموليه على صعيد نظام تشغيل واحد_أجهزه مختلفه :

ماذا عن توزيع تطبيقنا على زبائن مختلفه . حتى لو أستخدموا نفس نظام التشغيل . هل تتساوى لغات البرمجه بدعم هذه الحاله من المحموليه ؟

١ - مكتبات زمن تشغيل (Run Time) :

وهذا هو حال لغه Visual Basic .....

تحتاج بعض لغات البرمجه إرفاق مكتبات خاصه تعتمد عليها التطبيقات المبنيه بواسطتها ، ولاتعمل هذه التطبيقات بدون توفر نسخه من هذه المكتبات على الجهاز .

وبالتالي سنضطر لتوفير نسخه من هذه المكتبات مع برنامجنا ، لإننا لا نضمن أن المستخدم يملك نسخه من هذه المكتبات ، وبالتالي زياده بالحجم من جهه

ومن جهه أخرى ، تنسخ هذه الملفات إلى مجلدات خاصه بالنظام ، ولن نضمن ان يعرف المستخدم تنصيب البرنامج وأن يستطيع نسخ هذه الملفات إلى أماكنها الخاصه ، وبالتالي لابد من توفير برامج التحزيم (برامج التنصيب Install Managers) لكي نقوم بإعداد نسخه Setup من برنامجنا . مما يعني القليل من الحجم الزائد أيضا .

كذلك الأمر دعنا نفرض أن المستخدم قام بإعاده تنصيب نظام التشغيل مره أخرى ، هذا يعني أن مكتبات زمن التشغيل المخزنه فيه كلها محيت ويجب إعاده تنصيبها في النظام ، ولن تعمل برامجنا من دونها حتى لوكانت منسوخه إلى سواقه أقراص مختلفه عن سواقه نظام التشغيل ؟؟؟؟

ببساطه عليك إعاده تنصيب البرنامج كلما حدثت مشكله بالنظام أو كلما قمت بإعاده التنصيب

٢- الأدوات التي إستخدمناها في برنامجنا .

في كثير من الأحيان نستخدم أدوات غير الأدوات القياسيه المرفقه مع اللغه ، مثلا نتزلها من الإنترنت أو نبنيها نحن لتسهيل البرمجه او من أحد الأصدقاء ،،

بعض اللغات مثلا VB التي تستخدم مكونات Active X (ملفات ذات إمتداد OCX) ، عندما نقوم بإستخدام أحد هذه المكونات في تطبيقنا ، فإننا مجبرين على إرفاق ملف المكون (.ocx) مع برنامجنا ونسخه أيضا إلى مكتبات نظام التشغيل ... ؟؟؟؟ . وبالتالي كل العيوب التي تحدثنا عنها بالبند السابق من الحاجه لإعاده التنصيب .

ليت الأمر ينتهي هنا .. ستحتاج إلى تسجيل هذه الأدوات في النظام (الرجستري) ، والرجستري هو قاعده بيانات تحوي كل توصيفات وإعدادات الجهاز ، وكلما زاد حجم هذا البيانات كلما زاد بطء النظام الذي سيضطر للبحث في كميه اكبر من البيانات .

كما أنها أصبحت متاحه لجميع التطبيقات وفي حال نجح أحد ما بتسجيلها وإستخدامها ، مبروك عندها يستطيع إستخدام هذه الأداه بحريه (يوجد متخصصين كسر حمايه لهذه الملفات) دون إذن منك .

أصلا لوناقشنا الموضوع نظريا لوجدنا أن هذه الأدوات يتم إستدعائها خارجيا (ملفات مستقله عن تطبيقنا يتم إستدعائها من النظام) وبالتالي سيشكل ذلك بطيء في تحميل المشروع .

ربما يبدو كل ذلك عاديا ، حتى أن VC++ تملك نفس المشاكل .

ولكن إذا قارنا ذلك بالغات التي تؤمن دعم ملفات Stand Alone Exe's ، لوجدنا فارق كبير .

لاننسى أن المحموليه معيار مهم في تقييس لغات البرمجه .

لغه الدلفي الرائده في مجال المحموليه تسمح لك ببناء تطبيقات Stand_Alone . دون الحاجه لإرفاق مكتبات زمن تشغيل ، كما أتفقنا تطبيق دلفي يعتمد على نفسه قدر المستطاع .

على سبيل المثال إذا بنينا تطبيق بسيط في دلفي ، لايحتاج لملفات موارد (صور أو ما شابه) عندها يكفي نسخ الملف التنفيذي(EXE) وحده ولصقه في أي مكان على جهاز الزبون . تخيل الدرجه العاليه من المحموليه ،

كما أن نموذج مكونات دلفي (خذ VCL مثلا) يسمح بدمج الأدوات التي إستخدمناها وتضمينها مع النسخه التنفيذيه دون الحاجه لنقلها مع برنامجنا وتنصيبها كما في الحاه الأولى .

إذا كنت تعد برنامج كبير يحوي العديد من ملفات الموارد ، تستطيع نسخ ملفاتك كلها وتشغيلها من عند الزبون . ونادرا ماتحتاج لإعاده تنصيب برنامجك عند إعاده تنصيب نظام التشغيل ،

أما حاله مكتبات DLL وبعض مشغلات قواعد البيانات فيفضل نسخها إلى النظام بدلا من المجلد الحالي ، لكي تستفيد منها تطبيقات أخرى بدلا من إعاده نسخها كل مره وهذه حاله عامه في كل لغات البرمجه .

التكامل مع نظام التشغيل OS integrity:

جميل جدا أن نملك أدوات تملك محموليه عاليه وقادره على التعامل مع نظم مختلفه ، تحدث الإستقلاليه عن نظام التشغيل نتيجه عدم الإعتماد على أي من المميزات الخاصه بنظام تشغيل ما ، والإبتعاد عن كل خاصيه غير موجوده في النظم الأخرى ... لكي تكتب تطبيقات تتوقع نقلها إلى نظم تشغيل أخرى لا تستخدم أبدا توابع API الخاصه بنظام ما ، لإنها ستتغير من نظام لآخر ولايوجد مايضمن لك أن نسخ مشابهه لهاتعمل على نظام تشغيل آخر .
أن ذلك كما أتفقنا سيكسبنا عموميه وقدره على التعامل مع أكثر من نظام تشغيل ، ولكن حذاري فإن الإنعزال عن نظام التشغيل سوف يحرمنا من المزايا المميزه بكل نظام . ألا ترى معي أن نفس الأسباب السابقه التي تعطي المحموليه العاليه والإستقلاليه عن نظام التشغيل هي نفسها التي ستحرمنا من إستخدام الجوانب الإيجابيه التي يتميز بها نظام ما عن غيره ، وستجعلنا مجبرين على التعامل بشكل عام مع النظام دون القدره على إستتراف طاقاته والغوص بمميزاته وتقنياته التي ينفرد بها عن غيره .

نعم .. تملك جافا محموليه مذهله ، ولكنها كما هو متوقع غير قادره على الغوص في ثنايا نظام التشغيل نتيجه لذلك .
كما أنها في كثير من الأحيان غير قادره على دعم أمور تعتبر سهله وبسيطه في لغات أخرى ، لاتحوي جافا حتى الآن مشغلات MPEG أو حركات أخرى غير GIF متلاحقه (مع أنها تستطيع القيام بذلك بشكل أو بآخر مثلا محرك Macromedia مكتوب بالجافا) .
خذ على سبيل المثال صعوبه إنشاء تحكمات تقبل السحب والإفلات بواسطه جافا (drag and drop)
وبما أن جافا تحاول الإستقلال عن نظام التشغيل فهي ستدعم التقنيات التي تعمل على كل النظم بشكل رائع ولكنها ستخسر الأدوات المميزه بكل نظام ، ماذا عن دعم OLE في مايكروسوفت مثلا . حتى تطبيقات MDI (Multiple Document Interface) ستصتدم بها .

C++ تملك ميزه سحريه ، أن نظام التشغيل مكتوب بها . وبالتالي كميه كبيره من الوثائق والشروحات التي توضح خفايا نظام التشغيل وآليه التعامل مع توابعه وشرح لهذه التوابع والأمثله عليها ، كلها مكتوبه بلغه C++ نفسها التي بني عليها النظام . لاحظ وثائق MSDN الأساسيه من أجل أي مستخدم ، وكتب توابع API ويندوز كلها تدعم C++ في البدايه ثم يتم ترجمتها إلى لغات برمجه أخرى .
دليل Win API الذي أعتمده من أجل دلفي ، يرفق المثال الأصلي بالـــ C++ لمزيد من التوضيح والدقه ، مارأيك بتكامل C++ مع نظام التشغيل الآن .

لغات Microsoft تملك هذا الدمج الكبير ،

والسبب أنها لغات من إنتاج نفس الشركه التي أنتجت النظام (أهليه بمحليه = MS*MS) . وبالتالي تملك لغات مايكروسوفت ميزه تحسد عليها من بقيه اللغات أنها لغات أخوه بالرضاعه مع نظام التشغيل التي تعمل عليه ، وتتشرب كل تقنياته وميزاته . لاحظ لغه VB التي يجمع الجميع أنها لغه ضعيفه نحويا ولاتملك مزايا اللغات القويه ، ولكنها رغم ذلك قادره على فعل الكثير الكثير بالنظام ويندوز ، وتسمح لك بدرجه جيده من التحكم به .. أنا متأكد لو أنها تعمل على نظام ثاني لما ملكت هذه القوه الإضافيه أو لو أنها من إصدار شركه غير الشركه الغول Microsoft ، لإنها عندها ستخسر تكامليتها مع نظام التشغيل . أصلا تحاول مايكروسوفت أن تدعم VB من خلال نظام التشغيل ، لاحظ مثلا إمكانيه كتابه VB Script من برامج أوفيس ويندوز ، ربما إذا كتبت برنامج على الـــ Access تحتاج لبعض شفرات VB ضمنه ؟

في حين أن لغات أخرى أهم منها تقنيا غير قادره على القيام بما تقوم به من إندماج مع ثنايا نظام التشغيل التي تعمل عليه .

دعم الويب Web Supporting :

لاحظ أن دعم الويب بالآونه الأخيره لم يعد مجرد دعم لتقنيه عاديه ، حيث أصبح التطور التقني يركز على قدره الأفراد والمؤسسات على التواصل فيما بينها وتبادل شتى أنواع الوثائق الإلكترونيه والمعلومات ، ويتجه العصر لإعتبار الإتصال بالويب ضروره حتميه لأي حاسوب وربما أجهزه أخرى كالتلفزيون التفاعلي . لذلك أفردت دعم الويب كنقطه منفصله لكي أضع خطين تحت هذا البند .

دعم الويب هو تسهيل القدره على التحكم فيه وإستثمار كافه تقنياته .

جافا تملك دعم خيالي للويب ، وعلى حد تعبير SUN : "جافا هي لغه الويب والشبكات" ، حيث يمكن تنفيذ Applets ضمن وثائق الويب ، معظم مكونات واجهه المستخدم المرئيه GUI يمكن أن تستخدم في مستندات الويب لإنتاج تطبيقات تفاعليه من جانب المستعرض .

لغات NET. كذلك توجهت لدعم التقنيه الجديده (NET.) كخيار أستراتيجي يضمن تفوقها على غيرها من لغات العصر .

باتت تطبيقات ASP.NET تتصدر قائمه "المشاريع الجديده" في لغات NET. .

عندما أختار مشروع جديد في Delphi 8 فإن أول خيار يظهر أمامي هو "Delphi ASP Projects" ، والخيار الفرعي منه إفتراضيا يكون ASP.NET Web Application . ثم تدرج بقيه أنواع التطبيقات المعتاده التي يمكنك بنائها ؟؟

منذ متى كان الإهتمام بتطبيقات الويب بهذا الشكل .

أظن .. لابل أجزم أن هذا الإهتمام في تزايد مستمر ، وربما يكون إختصاص العصر القادم .

اللغات مفتوحه المصدر Open Source:

البرامج المفتوحه الشفره تملك بطاقه ضمان مفتوحه القيمه ، وتؤمن مقدره التطوير المستقبلي والتحسين والإضافه على البرنامج . كما أنها تؤمن القدره على إصلاح الأخطاء التي تظهر متأخره وسد الثغرات التي لم تأخذ بالحسبان ، أو مثلا القدره على تهيئه نسخ جديده من المشروع تعمل وفق شروط وبيئات جديده .

أهميه اللغات المفتوحه المصدر لاتقل أهميه عن البرامج المفتوحه المصدر ، بالعكس إنها تزيدها ميزه سحريه .. فهي تسمح للمبرمج من فهم الدقائق الداخليه التي تتألف منها اللغه ومكتباتها وتوابعها ، وتساعده في فهم مبدأ واليه عمل الكثير من الدوال والإجرائيات مما يجعله يستخدمها بالشكل الصحيح تماما ويبني دوال ومكتبات جديده فائقه بمواصفات مشابهه للمواصفات التي وضعها مبدعو اللغه . وتؤمن له توريث صالح ودقيق لأصناف اللغه الموجوده . كم أنه يستطيع التعلم من شفره اللغه المفتوحه بلا شك .

Python هي لغه برمجه مفتوحه المصدر (أنشأها Guido van Rossum) " أنا معجب بكل شيء مفتوح المصدر" وكل الشفرات التي تتكون منها Paython مفتوحه ويمكن تفحصها والتعديل عليها .

لغه دلفي ليست مفتوحه المصدر بالكامل ، ولكن مكتبه أدوات دلفي مفتوحه المصدر . ومكتوبه بدلفي نفسها .

.. انتهت ..

إعـداد :

عروه علي عيسى :

Web Site : www.orwah.net

E_Mail : WebMaster@orwah.net .

جمع المعلومات :

لغات Paython و SmallTalk و Perl و Lisp :

علاء الخطيب

E_Mail : Dolphin2@scs-net.org .

لغات C# و VB.net :

حسان اسماعيل

E_Mail : HassanMi@scs-net.org .

Syria . 5 / 12 / 2004 .