برمجة التطبيقات الشبكية

د. عمار جوخدار

مقدمة

- العرض موجهة لكل من يعمل بالتطبيقات الشبكية (LAN, WAN, Internet) برمجة أو إدارة أو تحليل أو تصميم أو تقييم أو استلام
- يهدف المحتوى ل،
  - التعريف بالتطبيقات الشبكية وخصوصيتها مقارنة بتطبيقات سطح المكتب وما ينجم عن تعدد المستخدمين من متطلبات غير وظيفية مثل الأمن والتنافس وإدارة الموارد والمتاحية وغيرها
  - التعريف بالمتطلبات غير الوظيفية للبرمجيات
  - التعريف بالتصميم البنيوي Architectural Design كحل للقضايا غير الوظيفية
  - التعريف بالبرمجة المشتقة من النماذج والتي تسمح بتسبيق التصميم البنيوي على التحليل
  - التعريف بالبرمجيات الوسيطة لربط البرمجيات المختلفة الداخلة في البنيان

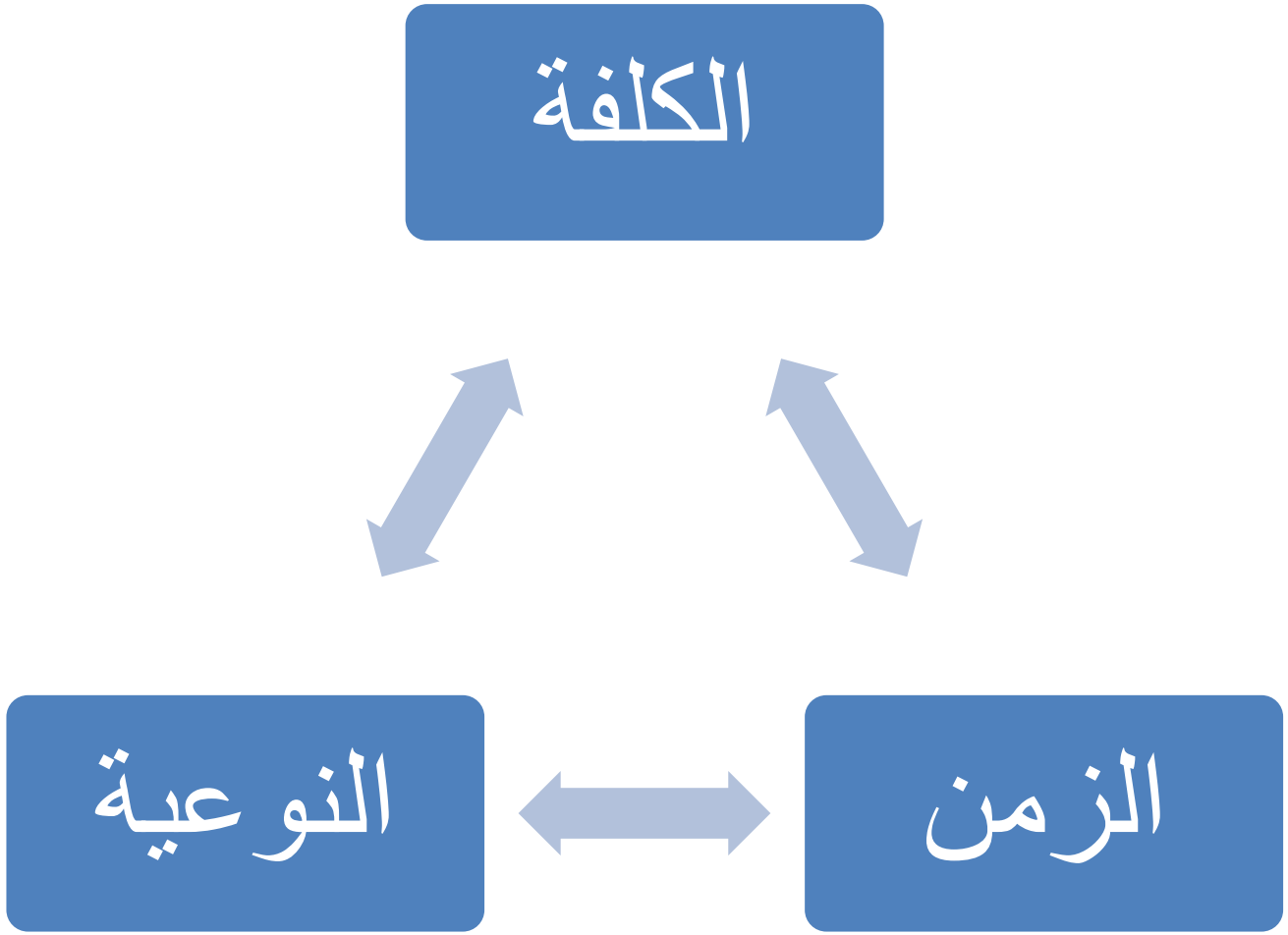
برمجة التطبيقات الشبكية

- نقول عن تطبيق أنه شبكي إذا تم نشره على عدة (مخدم أو زبون) أجهزة متصلة فيما بينها عبر شبكة (LAN or WAN)
- التطبيقات الداعمة للوب هي من أشهر التطبيقات الشبكية والتي تعمل على شبكة الانترنت وغالباً ما يستعمل فيها الزبون Browser أو تطبيق موبايل
- تسمى الزبون Thin Client عندما لا يحتاج إلا لمتصفح لاستعمال البرمجيات
- ويسمى Thick client عندما يحتاج إلى إرساء برمجيات خاصة لاستعمال البرنامج

خصوصية التطبيقات الشبكية

- **تعدد المستخدمين, وينتج عنه:**
  - **قضايا لها علاقة بالأداء**
  - **قضايا لها علاقة بالأمن.**
  - **تنافس على الموارد**
  - **ادارة الموارد:** مثل : CPU - Memory - Database - Hard Disk - Bandwidth Threads
- **التصعدية Scalability**
  - عدد المستخدمين سيزداد
  - عدد الخدمات سيزداد
- **التسامح مع الأعطال Fault tolerance**
  - يجب أن أعالج الأخطاء بحيث يؤثر الخطأ على أقل عدد من المستخدمين ولا يؤدي الخطأ إلى انهيار المنظومة البرمجية ككل.
- **صعوبة إدراكها**
  - من الناحية الستاتيكية فالرماز كبير ومعقد
  - من الناحية الديناميكة فعدد المناقلات كبير جداً
- **كلفة عالية** جداً كونها تحتاج لتخصيص وتحقيق
- ضررة **الربط والتكامل** باستعمال برمجيات وسيطة Middle Wares ربما لاتكون داعمة للمناقلا

الخصائص غير الوظيفية



أمثلة عن المتطلبات غير الوظيفية

- الأداء Performance
- الوثوقية Reliability
- الأمن Security
- التسامح مع الأخطاء Fault Tolerance
- التعافي من الأعطال Fail Over
- إدارة الموارد (CPU, Memory)
- المتاحية

$$\text{Availability} = \frac{\text{time to fail}}{\text{time to repair} + \text{time to fail}}$$

- Platform independence
  - نظام تشغيل
  - قواعد بيانات
  - مخدمات وب ومخدمات تطبيقات
- تعدد المداخل والصيغ Muti-Views
- Usability

- Accessibility
- المرونة Flexibility
- التصعدية
  - عدد المستخدمين
  - الوظائف والسمات Services and features
- سرعة التطوير والنشر والاختبار
- سهولة الصيانة
- التنصت والتعقب والتنبيه
- التخاطب والتكامل interoperability
- ACID
  Clustering
- Load Balancing

الأداء

- من وجهة نظر الجهة الطالبة Requirement
  - أن ينفذ المنظام مهام محددة بزمن محدد على عتاديات محددة (CPU، RAM، Hard Disk Bus, Cache Memor، Bandwidth) بوجود عدد محدد من المستخدمين المتنافسين يقومون بعدد محدد من العمليات hit rate وعلى قاعدة بيانات تتضمن حجم معطيات يمثل 10 سنوات للأمام, ويؤخذ بعين الاعتبار المعطيات الفعالة وغير الفعالة في قاعدة البيانات.
  - **يجب أن تكون استجابة النظام خطية مع ازدياد حجم الموارد.**
  - **Test Script**: هي العمليات التي سينفذها كل مستخدم و زمن الاستجابة المطلوب ويتم تحديد الزمن عبر دراسات مقارنة Bench Marking لبعض الأنظمة الموجودة لمعرفة الأداء الطبيعي.
- من وجهة نظر الجهة الراغبة بالتعاقد
  - يجب توفر bench marking لحالات مشابهة تسمح للشركة بالتأكد من قدرتها على التجاوب مع متطلبات الأداء
- **من وجهة نظر المصمم**
  - يجب على الشركة المنفذة تحديد مواصفات البرنامج قبل البدء بتنفيذه ووجود تصور كاف حوله وأنه سيحقق الأداء المطلوب .
  - من نماذج التضميم التي تسمح بالتحكم بالأداء
    - Pooling
    - caching
    - Multi-threading
    - Load balancing
    - AOP: لفصل المتطلبات الوظيفية عن غير الوظيفية فالمبرمج المسؤول عن خدمة معينة يتقنها
- من وجهة نظر الجهة المختبرة
  - تحتاج لآلية لتجهيز البيئة المحددة بالمتطلبات وهو عمل صعب نسبيا
  - وضع سيناريوهات ومحاكاة لعدد كبير من المستخدمين

الأمن الوظيفي

- يقصد به الحماية من المستخدمين المعَرّفين على النظام والذين يملكون صلاحيات دخول عليه. ويندرج تحته :
  - التعرّف Authentication: اي معرفة هوية المستخدم؟
  - السماحية Authorization : أي إعطاء صلاحيات للمستخدم؟
  - Monitoring

Authentication الأمن الوظيفي/

- اسم مستخدم وكلمة مرور.
- Password Policy : تتحدد بـ :طول كلمة السر وعمرها وصيغتها
- تخصيص عنوان IP تُقبل منه عمليات الدخول لبعض المستخدمين
- تجميد الحساب في حال تكرار الخطأ
- التحقق من أن المستخدم هو إنسان وهو الشخص الذي يدعي أنه هو ويتم ذلك بعدة طرق:
  - لوحة مفاتيح ذات مفاتيح متغيرة المواقع
  - Captcha:عبارة لا يستطيع فهمها سوى الإنسان
  - OTP (One Time Password)
  - أسئلة عن معلومات خاصة سبق إدخالها للنظام
- الوقت: نحدد أوقات معينة لدخول بعض المستخدمين

الأمن الوظيفي / Authorization

- **عناصر الأمن الوظيفي**
  - **عملية**: أي عملية يمكن لمستخدم أن يقوم بها مثل طلب إجازة، موافقة على إجازة، رفض إجازة، إنشاء امر شراء أو أمر بيع أو الإطلاع على مبيعات سابقة وهكذا
  - **معلومة**: وهي المعطيات التي تنشأ عن العمليات مثل الإجازة أو الدفعة أو عقد بيع أو أمر دفع.
  - **الدور**: هو مسمى افتراضي لمن يستطيع القيام بجموعة مترابطة من العمليات أو يطلع على معلومات محددة من الأدوار مسؤول الدفع ومسؤول القبض ومسؤول البريد الوارد ومسؤول البريد الصادر.
  - **الصلاحية:** وهي ربط الدور بمجموعة من العمليات والمعلومات ضمن أزمنة وشروط محددة.
- **من الأمثلة عن صلاحيات العمليات**
  - مسؤول الدفع يستطيع تسدد مبغ مستحق على الشركة سواء أكان بسبب عملية شراء بضاعة أو أصل أو غيره
  - مسؤول القبض يستطيع قبض مبلغ مستحق
  - مسؤول بريد وارد يستطيع إنشاء بريد وارد وتعديله والإطلاع على البريد السابق
  - مسؤول بريد صادر يستطيع طباعة بريد صادر والإطلاع على البريد المعلق
- **من الأمثلة عن صلاحيات المعلومات**
  - مسؤول البريد الصادر لا يستطيع الإطلاع إلا على البريد الخاص بالمجموعة التي يعمل ضمنها
  - مسؤول القبض لا يستطيع رؤية إلا عمليات القبض التي تمت على صندوقه
- **من الأمثلة عن أزمنة الصلاحيات وشروطها**
  - لا يحق لموظف تعديل إجازته بعد الموافقة عليها من مديره
  - يمكن لأمين المستودع تسليم البضائع في حال وجود أمر تسليم فقط
  - يمكن لأمين المستودع استرداد بضاعة شرط أن تكون قد خرجت من نفس مستودعه
  - يمكن لموظف تقديم إدازة من أي مكان يريد وفي أي زمان ولا يستطيع إجراء عملية بيع إلا من داخل مقر الشركة وفي أوقات الدوام الرسمي
- لا بد من محرك إجراءات لضبط الأمن المرتبط بتسلسل العمليات

Flexibility المرونــة

- المرونة هي القدرة على تعديل سلوك البرمجية دون العودة للبرمجة
- لابد من تحديد مستوى المرونة المطلوب في مرحلة المتطلبات والتحليل. هل هي على مستوى الحسابات أم على مستوى الإجراءات أو غيرها
- يتم تحقيق المرونة برمجياً عبر تحويلها لمجموعة إعدادات settings و صلاحيات يستطيع المستخدم ضبطها والتحكم بها
- يفيد الـAspect في إضافة سمات جديدة عجزت مرونة التحليل عن تغطيتها دون الحاجة لتعديل الرماز القديم بحيث نحول دون حدوث regression
- كل ما ازدادت المرونة تزداد التكلفة لأن حجم الـ codeالبرمجي سيزداد حجمه وسيمر بحالات متعددة وستزداد مسارات الحلول وعدد مرات الاختبار وبالنهاية سيزداد تعقيد النظام وتنخفض نوعيته وأداؤه.

Audit, Alert, Tracing

- يجب تحديد الطبقات التي نرغب بالتنصت عليها أثناء وضع المتطلبات مثل واجهة التخاطب أو قاعدة البيانات أو منطق العمل BL وتحدد المعلومات المراد جمعها نتيجة التنصت
- معلومات التنصت قد تفيد إما بحفظ Log للقيام بعملية Tracing لاحقاً
- أو لتحليلها آنياً وإرسال تحذيرات Alert .
- من أهم نماذج التصميم التي تفيد في التنصت هي AOP.

إدارة المناقلات "Transactions"

ACID

تعريف المناقلة: نقول عن عملية أنها مناقلةTransaction إذا حققت:

- Atomicity : جميع العمليات الجزئية تتم معاً أو لا تتم أبداً.
- Coherence - التماسك : يجب أن تبقى البيانات منسجمة ومتجانسة مع بعضها البعض قبل وبعد المناقلة(ليس فقط نجاح وفشل المناقلة).
  - منع المستخدم من حذف عملية والنظام لم يحذفها بعد من قاعدة البيانات.
  - منع إضافة مناقلة لمستخدم وهو لم يطلبها (شخص حول مبلغ مالي وبالحقيقة هو لم يحول شي)
  - ارسال تنبيه لعملية لم تكتمل.
- Isolation – العزل: أن تكون نتيجة المناقلة مستقلة عن المناقلات الأخرى التي تتم على التوازي معها.
- Durability – الديمومة: نتيجة المناقلة لا تتغير سلبا ولا إيجاباً. (لا يمكن أن يرسل تقرير بنجاح عملية ثم يرسل تقرير فشل بإتمامها)

مثال عملية سحب مبلغ مالي من ATM .

إدارة المناقلات "Transactions"

- يوجد نوعيم من المناقلات User Managed Transaction و Container Managed Transactions أو المناقلات التصريحية وهي الأسهل تحقيقاً وأكثر مرونة
- **المناقلات التصريحية (أو المعالجة من قبل الـ container وهي البيئة المُشغّلة للخدمات).** من أجل كل تابع نحدد نوع المناقلة المرغوبة, وكل مناقلة تدعم أحد التصريحات التالية :
  - **Required:** تعني أن ينضم التابع عند طلبه لمناقلة أخرى مفتوحة, أي يتم تنفيذه مع التابع الذي استدعاه بنفس المناقلة, وإن لم توجد مناقلة مفتوحة (أي أن التابع الذي استدعاه non transactional) يتم فتح مناقلة جديدة له.
  - **Not Required:** لا يتأثر التابع بوجود المناقلة أو عدم وجودها (لا يحتاج لمناقلة ولا يضره وجودها).
  - **Required New:** ينشئ التابع مناقلة مستقلة خاصة به بصرف النظر عن التابع الذي استدعاه
  - **Supported:** إذا كان التابع المستدعِي له مناقلة مفتوحة فإن التابع المُستَدعَى ينضم إليها وإن لم يكن للتابع المستدعي مناقلة مفتوحة فلا يتم فتح مناقلة جديدة للتابع المستدعى.
  - **Not Supported:** التابع لا يدعم المناقلة لذلك يجب عدم استدعاؤه بمناقلة وبحال كان للتابع المستدعي مناقلة يتم تعطيلها ريثما ينتهي التابع من التنفيذ ثم يتم استئناف المناقلة.
  - **Never:** يظهر التابع Exception بحال تم استدعاؤه وكان هناك مناقلة مفتوحة.
  - **Mandatory:** التابع يحتاج لمناقلة موجودة مسبقاً وبحال عدم وجودها يظهر Exception.

Functional Scalability

- هو من أهم الخصائص الغير وظيفية لأي تطبيق.
- قد يؤدي أي تطوير أو تعديل على برنامج موجود إلى تراجع عن شيء كان محقق مسبقا وقد يكون زمن التعديل على برنامج معين مساو لزمن تطوير هذا البرنامج من الصفر.
- المطلوب: القدرة على إضافة وظائف إضافية على البرنامج دون الاضطرار إلى تغيير شيء سبق وتم برمجته وهذا ما يسمى functional scalability أي التصاعدية الوظيفية بمعنى أن يكون البرنامج قابل للتوسع وإضافة وظائف جديدة دون التأثير على الوظائف السابقة وبالتالي تصبح كل وظيفة بحد ذاتها قابلة للبرمجة من قِبَل أحد المبرمجين وهذه الوظائف تتكامل سويا دون تدخل بشري ودون تراجع.
- أهم طريقة وجدت وأفضل طريقة هي(Aspect Oriented Programming)AOP.

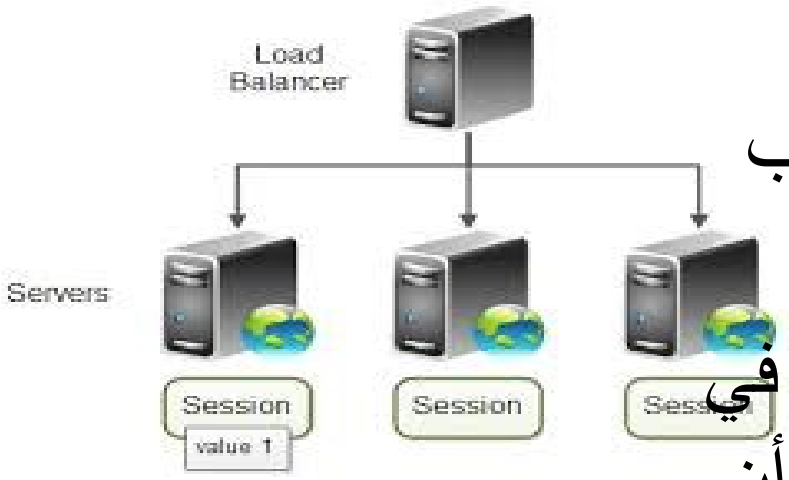
User Scalability

- القدرة على تحمل عدد متزايد من المستخديم دون العودة للبرمجة من جديد ودون تعديل العتاديات الحالية وفقط بإضافة عتاديات جديدة
- هذا المفهوم مرتبط بالـ load balancing
- في البنيان ثلاثي الأرتال 3Tiers يتم ذلك في الطبقة الوسطى BL Server والقابل للتكرار
- لابد للمصمم من لحظ استهلاك الذاكرة والـCPU لكل مستخدم
- هناك أدوات خاصة بال bench marking ومحاكاة عدة مستخدمين مثل Jmeter, Open STA, Web Runner وبعضها مجاني ويوجد غيرها تقوم هذه الأدوات بمحاكات عدد من المستخدمين.

Resource Management

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

Clustering, Load Balancing and Fault Tolerance



- **وتفيد في تقسيم الطلبات بين مجموعة من المخدمات** ونميز بين حالتين هما Clustering وLoad Balancing
- Load Balancing: في حال انقطاع أو توقف المخدم يجب على المُستخدم المحاولة مرة أخرى.
- Clustering: مكلفة إلا أنها شفافة بالنسبة للمستخدم لأنه في حال توقف المخدم الذي يعمل عليه ينتقل لمخدم آخر دون أن يشعر لأن المخدمات تكون مشتركة بالـ Sessions.

الإتاحية Availability

- الإتاحية

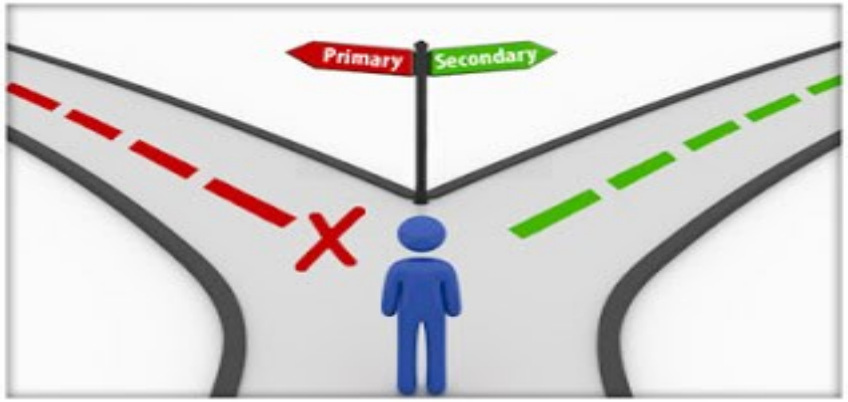
$$\text{Availability} = \frac{\text{time to fail}}{\text{time to repair} + \text{time to fail}}$$

- من وجهة نظر الجهة الطالبة
  - يجب ألا يتوقف النظام أكثر من ساعة في اليوم أو في السنة أو ألا يتوقف أبدا.
  - يجب أن لا يتوقف في أوقات معينة مثل الدوام الرسمي

- من جهة المصمم(التنفيذ) كيف يتم ضمان التسامح مع الأعطال؟
  - *Thread pool*: حيث في حال حدوث عطل مع مستخدم فلا يتوقف النظام بل يتوقف النيسب المسؤول عن تخديم هذا المستخدم.
  - *Clustering* أو *load Balancing*: تعطي نوع من التسامح مع الأعطال في حال كان على مستوى المُخدمات.
  - إذا كانت المشكلة software سيتوقف نيسب وحيد، وإذا كانت hardware سيتوقف مُخدّم وحيد.

الاستعادة من الأعطال Fail Over

- هو الانطلاق مرة ثانية بالنظام بشكل صحيح وذلك بعد توقفه نهائياً
- زمن إصلاح العطل : عندما يحدث عُطل معيّن فكم الزمن الازم لإعادة تشغيل البرنامج و القيام بعملية back upللDB والزمن الازم لإعادة تشغيل الapplication server.
- يؤثر على الـFail Over عوامل عديدة:
  - عدد المُخدّمات.
  - استعمال الـACID بشكل صحيح
  - وجود Replication على مستوى DB.
  - القيام بـ Back up يومي لقاعدة البيانات



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

Usability: سهولة الاستعمال.

- يبدأ التفكير بسهولة الاستعمال أثناء وضع المتطلبات وذلك
  - بتحديد المستوى التعليمي للمستخدم النهائي
  - الزمن اللازم لإدخال معطيات محددة وتكرار الإدخال اليومي أو الشهري
  - أنواع التقارير المطلوبة
  - التنبيهات والتحذيرات
  - زمن التدريب المناسب لطبيعة عمل المؤسسة
  - سهولة الإلغاء والتعديل
- يختلف مفهوم سهولة الاستعمال تبعا لنوع البرنامج
  - برنامج مصرفي لا يحتاج لسرعة في إجراءات العمليات بل على العكس يرغب دوما بالقيام بالعملية على مرحلتين إنشاء وتدقيق.
  - برنامج شبكة اجتماعية لا بأس بالسرعة مقابل احتمال أكبر لأخطاء الإدخال

Accessibility

- **Accessibility** : أي القدرة على الوصول للمعلومة حتى لو كان الإنسان يعاني من مشاكل معينة في
النظر أو السمع أو الحركة أو غيرها
  - وجود المكبرة في نظام تشغيل *windows*، لضعيفي النظر
  - إمكانية ثبيت زر الـshift لمن لا يستطيع استعمال كلتا اليدين

التنـافس Concurrent Access

- مثال الـATM الشهير أدناه

- يطلب الزوج 100
- تتحق الآلة من توفر المبلغ
- تسحب الآلة المبلغ
- تقدمه للزوج

- تطلب الزوجة 100
- تتحق الآلة من توفر المبلغ
- تسحب الآلة المبلغ
- تقدمه للزوجة

لماذا نحتاج لبنيان Architecture؟

- يمكن لأي سطر في البرنامج أن يؤثر على أي من الخصائص غير الوظيفية للنظام
- التفكير في جميع المتطلبات غير الوظيفية في آن واحد أثناء عملية التطوير معضلة كبيرة
- البينان يساعد في تحقيقها من خلال فصلها عن بعضها البعض وضعها ضمن مجموعة من نماذج التصميم
- بيسمح البينان الجيد بتطوير برمجيات ذات نوعية مستقلة نسبيا عن نوعية المبرمجين

مستويات البنيان السبعة

Global architecture

Enterprise architecture

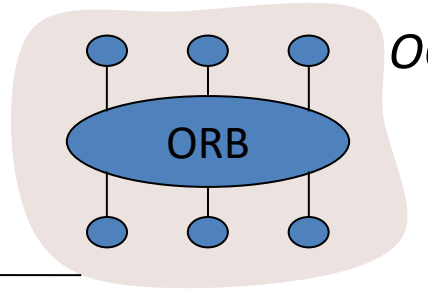
System architecture

Application architecture

Macro-architecture

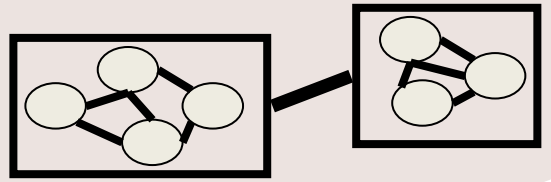
Micro-architecture

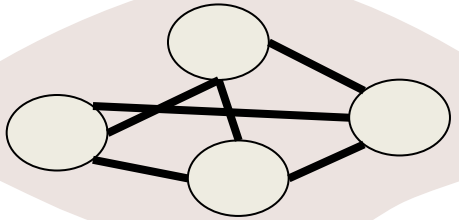
Objects



OO architecture

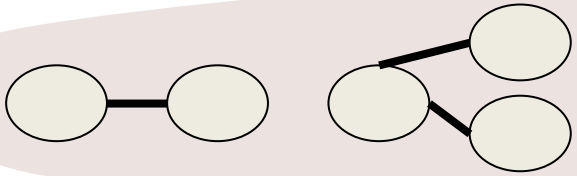
Subsystem

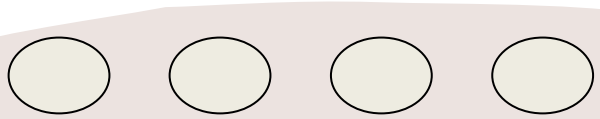




Frameworks

Design patterns





OO programming

* Mowbray and Malveau

Design Patterns

- MDA: Model Driven Architecture
- Meta Data
- Strategy: DB driver
- Observer: Business Rules/ Listeners/frame
- Lazy List: SMS result set
- Frames.

- 3 tiers
- MVC: Model View Controller
- Façade
- Message
- ORM: Object Relational Mapping
- DAO: Data Access Object
- AOP: Aspect Oriented Programming
- SOA: Service Oriented Architecture
- Factory: DAO, DB connection, etc.

Design Pattern vs Framework

- Patterns are lower-level than frameworks
- Frameworks typically employ many patterns (Elixir)
- Architectural patterns tend to be focussed on middleware. They are good at capturing:
  - Concurrency
  - Distribution
  - Synchronization
  - scalability

Enterprise Architecture البنيان المؤسساتي

See https://www.youtube.com/watch?v=qDI2oF1bASk&desktop_uri=%2Fwatch%3Fv%3DqDI2oF1bASk&app=desktop

Client

USER

. Browser
. GUI
. Mobile

APP

. Web Service
. Email
. Fax
. Queue

Business logic tier

View (Presentation)

Connectivity

- http request •
- SMS •
- Email •
- Fax •
- Mobile •
- App •
- other •

Transformer

- http •
- XML •
- Text •
- Excel •

Controller(Business Logic)

FAÇADE

Business Services

S1
S2
.
.
.
Sn

Aspects

ACID . Security . Tracing . Auditing . Logging . BRs

Model

ORM

Object PIF(tables, relations, fields..)

DAO (physical)

Drivers
SQL Server
XML
Oracle
MySQL

Infrastructure

Web component (containers)

Servlet . JSP . JAF . JDBC . EJB . MDB . EB

Java Active Framework (fax, mail)

Java Database Connectivity

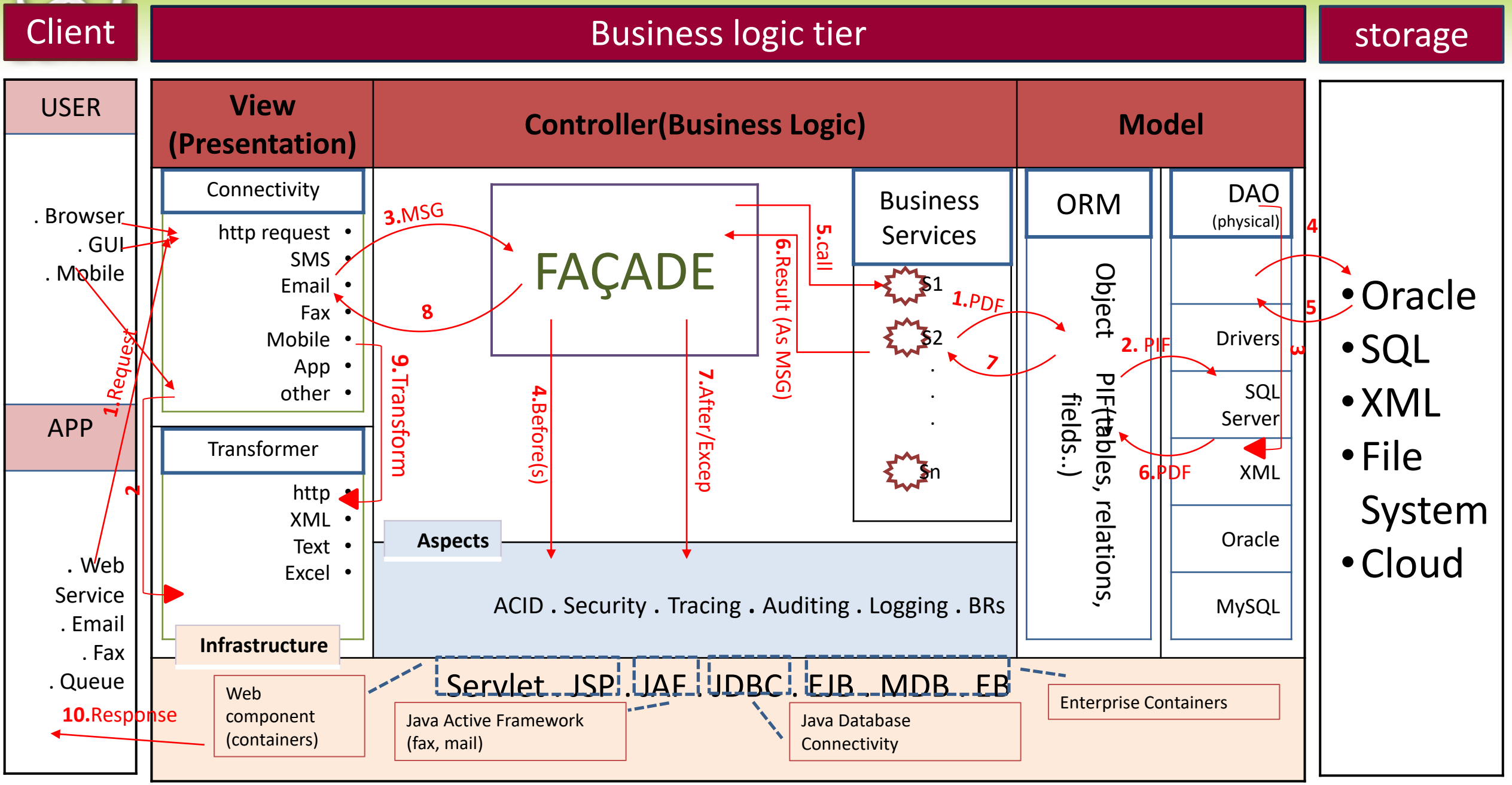
Enterprise Containers

storage

- Oracle •
- SQL •
- XML •
- FS (File System) •
- Cloud •

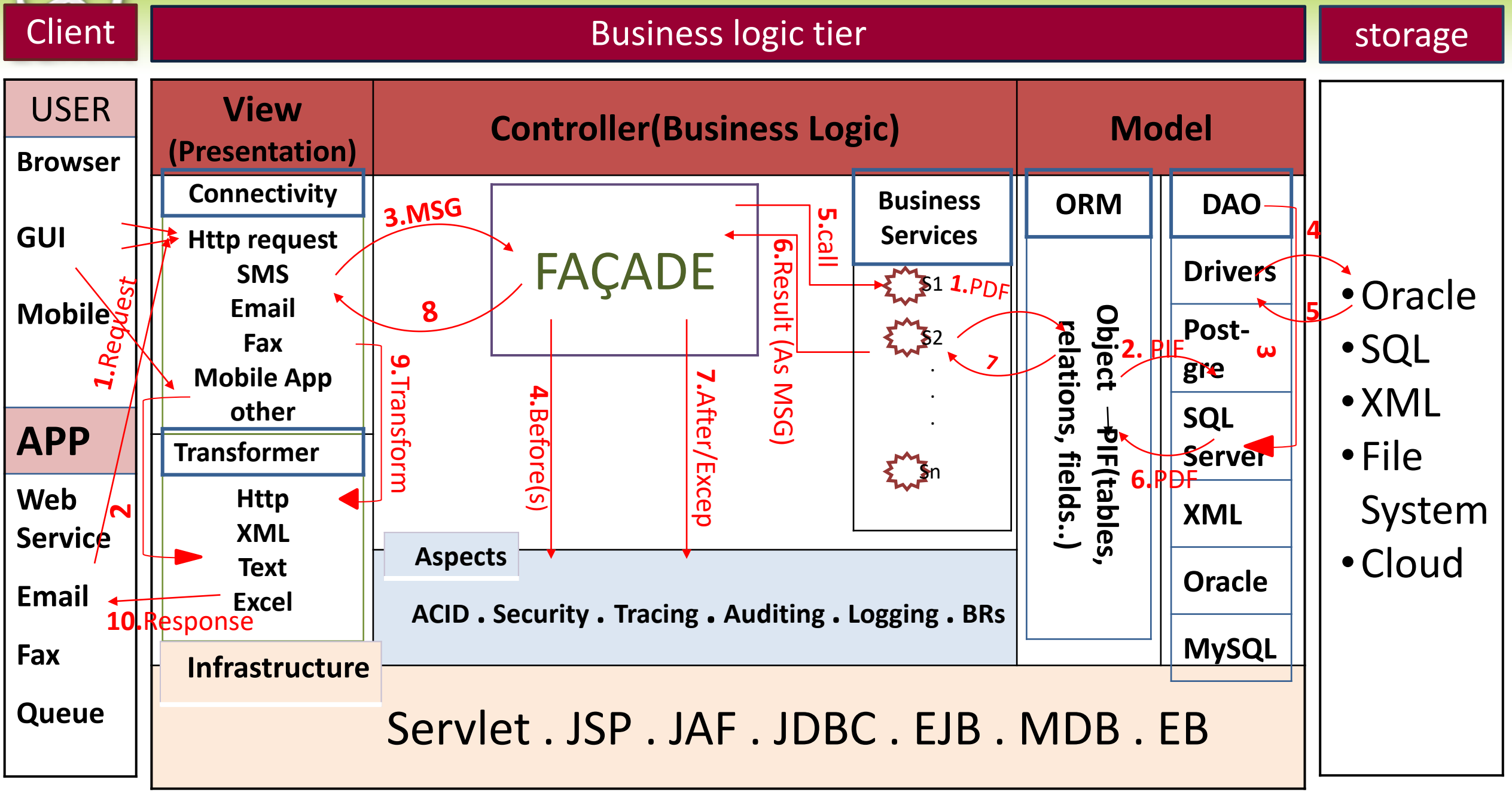
Enterprise Architecture البنيان المؤسساتي

See https://www.youtube.com/watch?v=qDI2oF1bASk&desktop_uri=%2Fwatch%3Fv%3DqDI2oF1bASk&app=desktop



Enterprise Architecture البنيان المؤسساتي

See https://www.youtube.com/watch?v=qDI2oF1bASk&desktop_uri=%2Fwatch%3Fv%3DqDI2oF1bASk&app=desktop



Façade Design Pattern

- تصل الـ Request من طبقة الزبون إلى طبقة presentation مباشرة إلى البوابة الموافقة (http, SMS, Email, etc)
- تقوم هذه الطبقة باستعمال الـ transformer المناسب لتحويل الرسالة إلى Message وتضمين اسم الخدمة والمعاملات ونوع المناقلة والبوابة Port وإرسالها إلى الـFacade
- تقوم Facade بـ
  - تطبيق السمات القبلية before المناسبة عليها
  - وتوجيهها للخدمة الموافقة (بالاسم) عبر البوابة المطلوبة Middleware (EJB, Queue, Mail, etc)
  - تطبيق السمات البعدية after المناسبة عليها في حال نجاح الإرسال
  - تطبيق عمليات onException المناسبة في حال فشل الإرسال

AOP implementation Example

- Interface Aspect {
  - Msg before(Msg input);
  - Msg after(Msg result);
  - Msg onException(Exception ex);
- }

AOP implementation Example

- Class Façade {
  - static List aspects;
  - static void register(Aspect a) {aspects.add(a);}
  - Msg execute(Msg m) {
    - For each a in aspects do
      - A.before()
    - Try {
      - Msg r= msg.execute();
      - For each a in aspcets do { a.onafter(r); }
    - } catch (Exception e) {
    - For each a in aspcets do { a.onException(ex); }
- }
- }

AOP implementation Example

- Class Tracing implements Aspect {
  - Msg before(Msg input) {print(msg);}
  - Msg after(Msg result) {print(result);}
  - Msg onException(Exception ex) {ex.printStackTrace();}
- Static {
  - // IoC design pattern or Hollywood principle
  - Façade.register(new Tracing());
- }
- }

AOP Business example/Before

```
// simple business logic: transfer from bank account
to another

Boolean transfer( acc1, acc2, amount) {
  debit(acc1, amount);
  credit(acc2, amount)
}
```

AOP Business example + tracing/Before

```
// same with tracing
Boolean transfer( ac1, acc2, amount) {
  try {
        print(“in”+ transfer+”:”+acc1+”,”+acc2+”,”+ amount)
        debit(acc1, amount);
        credit(acc1, amount)
        print(“out”+ transfer+”:”+acc1+”,”+acc2+”,”+ amount)

      }
Catch (exception e) {
    print(“ex”+ transfer+”:”+acc1+”,”+acc2+”,”+ amount+”\n”+message)
}
}
```

AOP Business example + ACID/Before

```
// same with ACID transaction
Boolean transfer( ac1, acc2, amount) {
  try {
        print(“in”+ transfer+”:”+acc1+”,”+acc2+”,”+ amount)
        startTransaction();
        debit(acc1, amount);
        credit(acc1, amount)
        print(“out”+ transfer+”:”+acc1+”,”+acc2+”,”+ amount);
        commitTransaction();

      }
Catch (exception e) {
    print(“ex”+ transfer+”:”+acc1+”,”+acc2+”,”+ amount+”\n”+message)
     rollback();
}
}
```

AOP Business example + Security/Before

```
Boolean transfer( ac1, acc2, amount) {
If(!cuurentUser has right to transfer > amount ) throw exception;
// Other security rules such as role and other amount values

  try {
        print(“in”+ transfer+”:”+acc1+”,”+acc2+”,”+ amount)
        startTransaction();
        debit(acc1, amount);
        credit(acc1, amount)
        print(“out”+ transfer+”:”+acc1+”,”+acc2+”,”+ amount);
        commitTransaction();

      }
Catch (exception e) {
    print(“ex”+ transfer+”:”+acc1+”,”+acc2+”,”+ amount+”\n”+message)
     rollback();
}
}
```

AOP Business example + Security/After

- Class Tracing implements Aspect {
  - Msg before(Msg input) {print(msg);}
  - Msg after(Msg result) {print(result);}
  - Msg onException(Exception ex) {ex.printStackTrace();}
- Static {
  - // IoC design pattern or Hollywood principle
  - Façade.register(new Tracing());
- }
- }

لاحظ ان الرماز اصبح مستقلا
على معاملات الخدمة وقابل
لاعدادة الاستعمال

AOP Business example /After

```
// simple business logic: transfer from bank account to
another
Boolean transfer( msg) {
  debit(msg.acc1, msg.amount);
  credit(msg.acc2, msg.amount)
}
```

AOP Business example /After

```
Class Tracing implements Aspect {
      Msg before(Msg input) {print(input);}
      Msg after(Msg result) {print(result);}
      Msg onException(Exception ex) {ex.printStackTrace();}
Static {
      // IoC design pattern or Hollywood principle
      Façade.register(new Tracing());
}
}
```

AOP Business example /After

```
Class ACID implements Aspect {
    Msg before(Msg input) {startTransaction();}
    Msg after(Msg result) {commitTransaction();}
    Msg onException(Exception ex)
     {rollback()
Static {
    // IoC design pattern or Hollywood principle
    Façade.register(new ACID (), *);
}
}
```

AOP Business example /After

```
Class Security implements Aspect {        }
    Msg before(Msg input) {checkUser);}
    Msg after(Msg result) {;}
    Msg onException(Exception ex) {}
    Static {
    // IoC design pattern or Hollywood
    principle
    Façade.register(new Security (), *);
}
```

بعد البنيان المؤسساتي

```
Class BusinessSecurity implements Aspect {
    Msg before(Msg msg) {if(user does not have the right to transfer >
    msg.amount) throw exception ;}
    Msg after(Msg result) {;}
    Msg onException(Exception ex) {}
    Static {
    // IoC design pattern or Hollywood principle
    Façade.register(new BusinessSecurity (), transfer);
}
}
```

بعد البنيان المؤسساتي

- جميع الخدمات أصبحت قابلة للتعميم اعتمادا على مبدأ الاستدعاء بالرسائل Message
- بالتالي فإن الفائدة ليست فقط بفصل المتطلبات الوظيفية عن غير الوظيفية بل ايضا في إعادى استعمالها
- حجم الرماز قبل استعمال البنيان المؤسساتي **هو عدد الخدمات الوظيفية مضروبا بعدد الخدمات غير الوظيفية**
- حجم الرماز بعد استعمال البنيان المؤسساتي **هو عدد الخدمات الوظيفية مجموعا له عدد الخدمات غير الوظيفية**

المكسب كبير جدا من ناحية توزيع العمل وحجم الرماز والصيانة

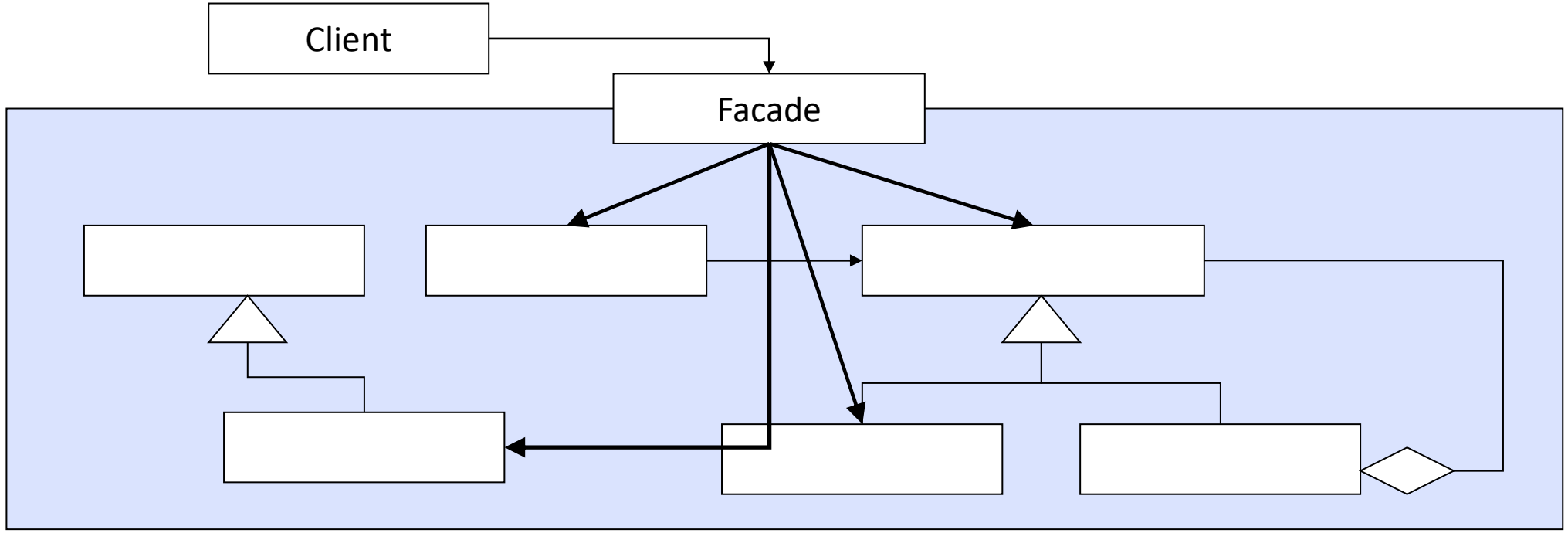
3 Tiers vs 2 Tiers

Client	Server	
GUI + BL	DB	المنهجية 1 Thick Client
GUI	DB + BL	المنهجية 2 Thin Client

Client	Server	Storage
GUI	BL	DB

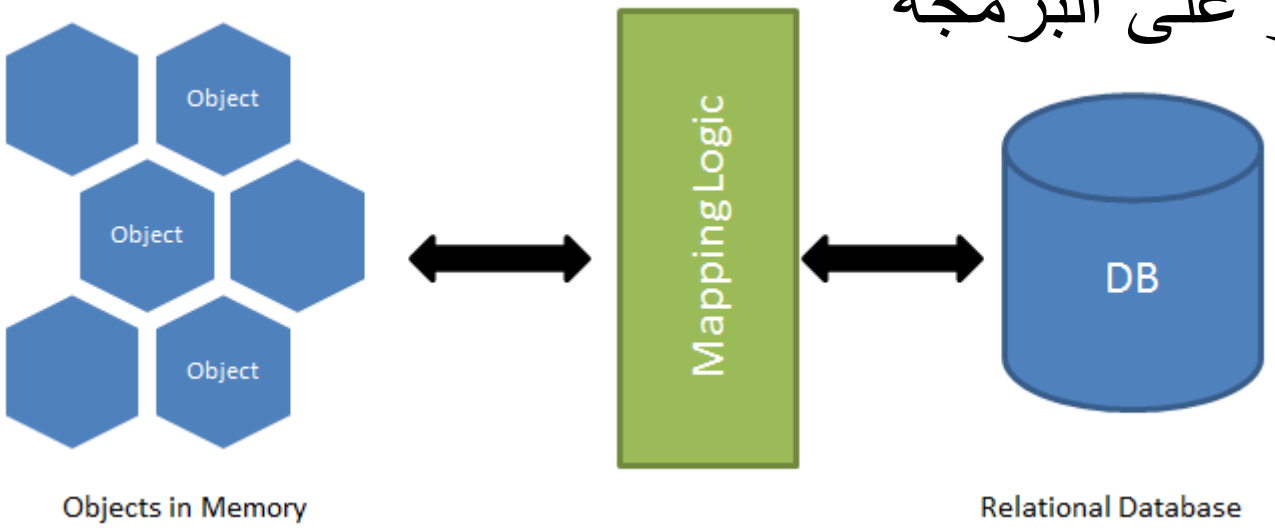
Façade Design Pattern

- Provide unified interface to interfaces within a subsystem
- Shield clients from subsystem components
- Promote weak coupling between client and subsystem components



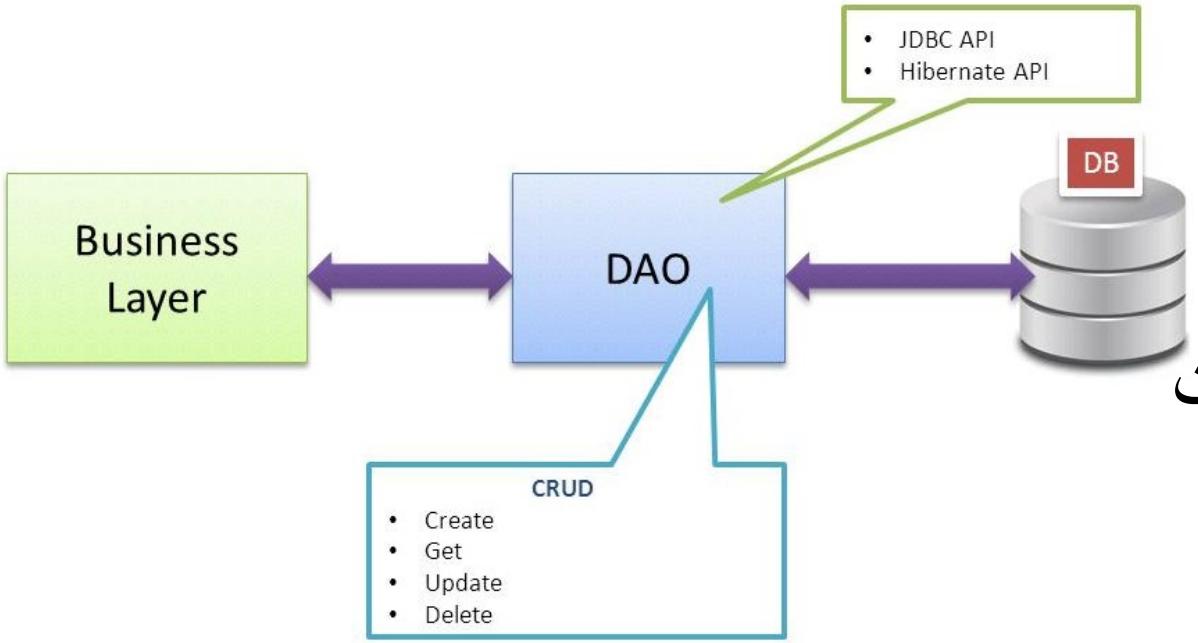
ORM Design Pattern

- اختصار لـ object relational mapping
- يسمح بالانتقال آلياً من الأغراض في لغة البرمجية (جافا أو C++ أو غيرها) إلى شكل علاقاتي يمكن تخزينه في قاعدة البيانات وبالعكس طبعاً
- يساعد في توفير الوقت والجهد والتركيز على البرمجة



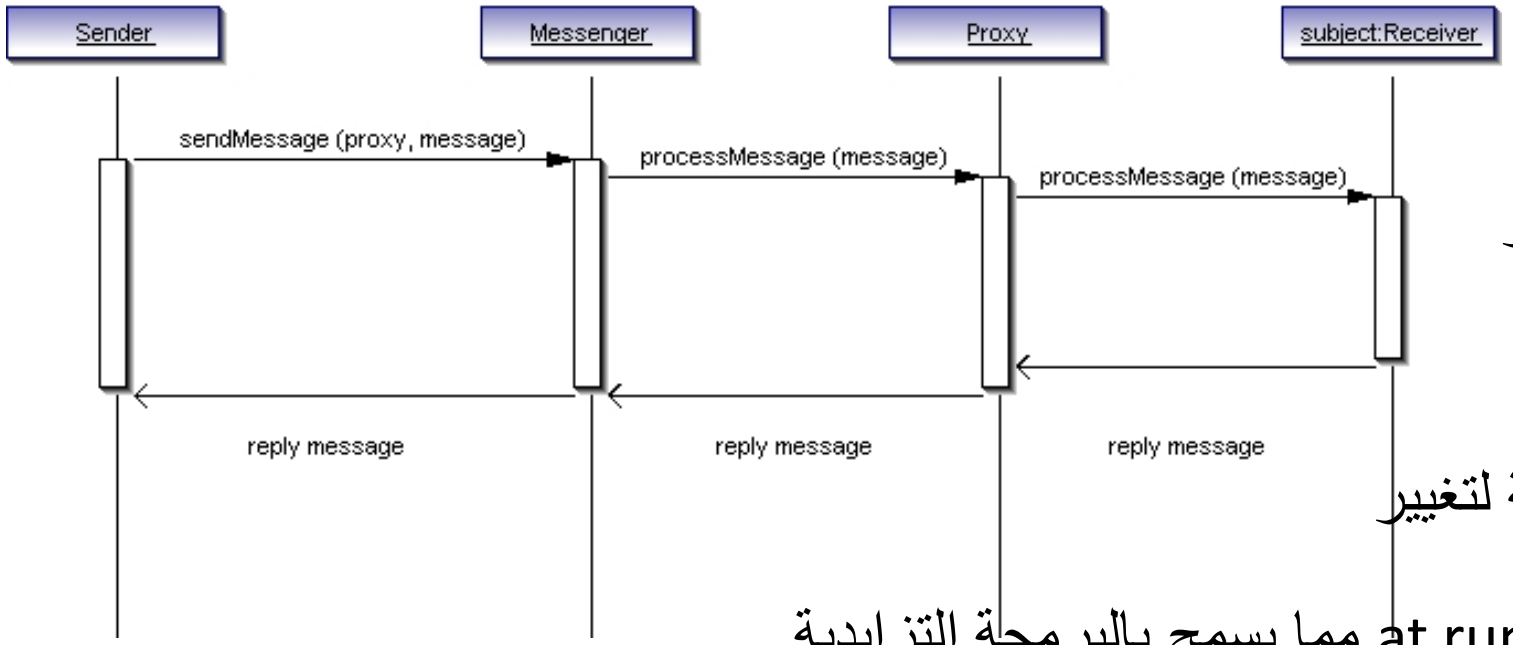
DAO Design Pattern

- ***Data Access Object :DAO***
- تسمح بالانتقال الآلي من الترميز العلاقاتي الناجم عن ORM إلى قاعدة البيانات وبالعكس
- تسمح بالتعامل الشفاف مع قاعدة البيانات
- تسمح بالتعامل مع عدة قواعد بيانات
- تسمح بتحقيق الاستقلال عن قواعد البيانات

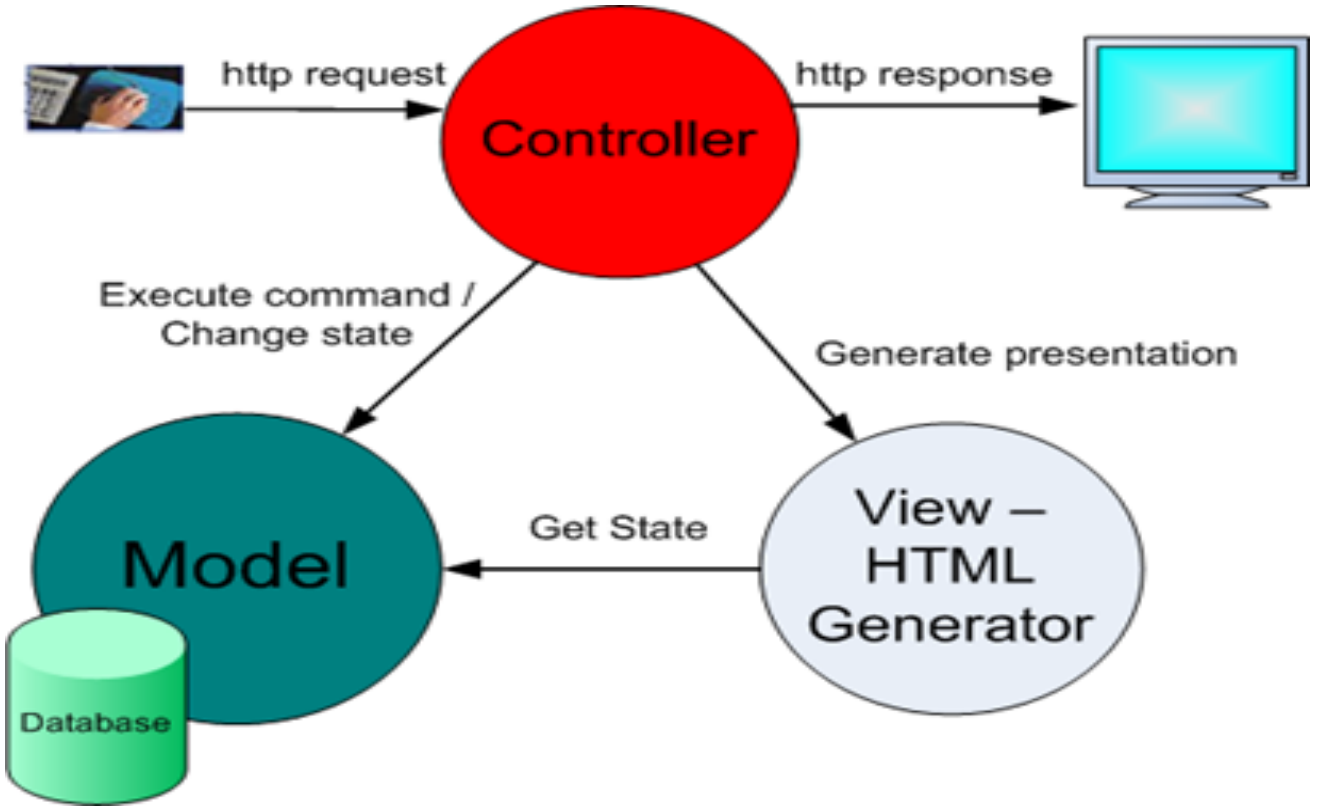


Message Design Pattern

- تسمح بتحويل الاستدعاء التقليدي للتوابع باستعمال عمليات قفز Jump إلى رسالة Message وهذا يسمح بـ
  - تحويل الاستدعاء إلى أمر قابل للمقاطعة
  - جعل الاستدعاء قابل للحفظ Serializable وبالتالي يمكنه عبور الشبكة عبر Middleware ولا يقتصر على المعالج نفسه
  - إمكانية التنصت والتسجيل
  - إضافة معاملات إضافية دون الحاجة لتغيير صيغة الاستدعاء
  - التحقق من صحة الاستدعاء at run time مما يسمح بالبرمجة التزايدية
  - توحيد interface لجميع الخدمات مما يعزز إمكانية إعادة الاستعمال

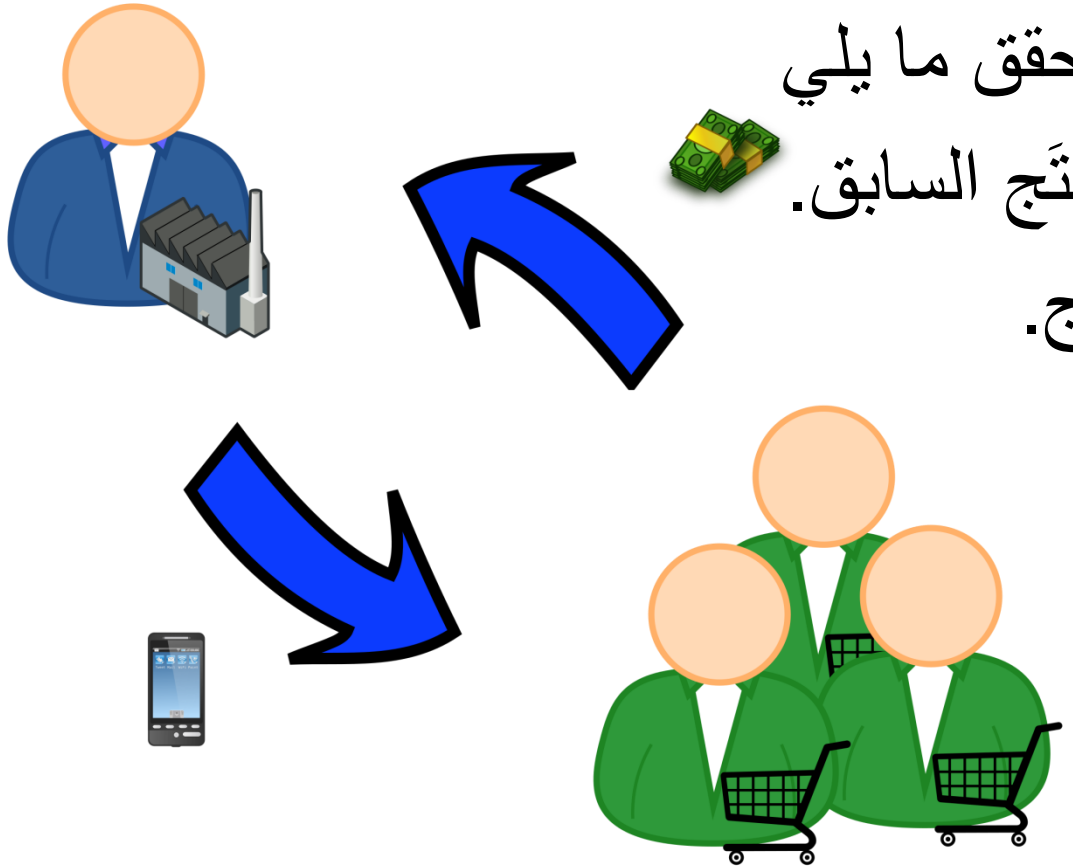


MVC



- تهدف إلى فصل طبقة التخاطب مع الزبون client وهي طبقة view عن طبقة التعامل مع النموذج model عن التحكم controller وتسمح بتعدد الـViews وModels

معالجة التنافس Concurrent Access
نموذج Producer/Consumer



لحل التنافس بين المنتج والمستهلك يجب أن يتحقق ما يلي

- يجب ألا يُنتج المُنتج منتجاً قبل استهلاك المنتَج السابق.
- ألا يستهلك المستهلك منتجاً غير كامل الانتاج.
- المستهلك لا يستهلك منتجاً مرتين.

من الأمثلة

- برامج المحاكاة
- التجارة الإلكترونية

Producer - Consumer

كيف نحل مسألة المنتج/المستهلك

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

```
void main(String[] args) {
   Product p=new Product();
   Producer pr=new Producer(p);
   Consumer c=new Consumer(p);
   pr.start();
   c.start(); }
}
```

كيف نحل مسألة المنتج/المستهلك

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

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

حل مشكلة استهلاك منتج غير كامل

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

حل مشكلة استهلاك المنتج مرتين أو عدم استهلاكه باستعمال الانتظار الفعال

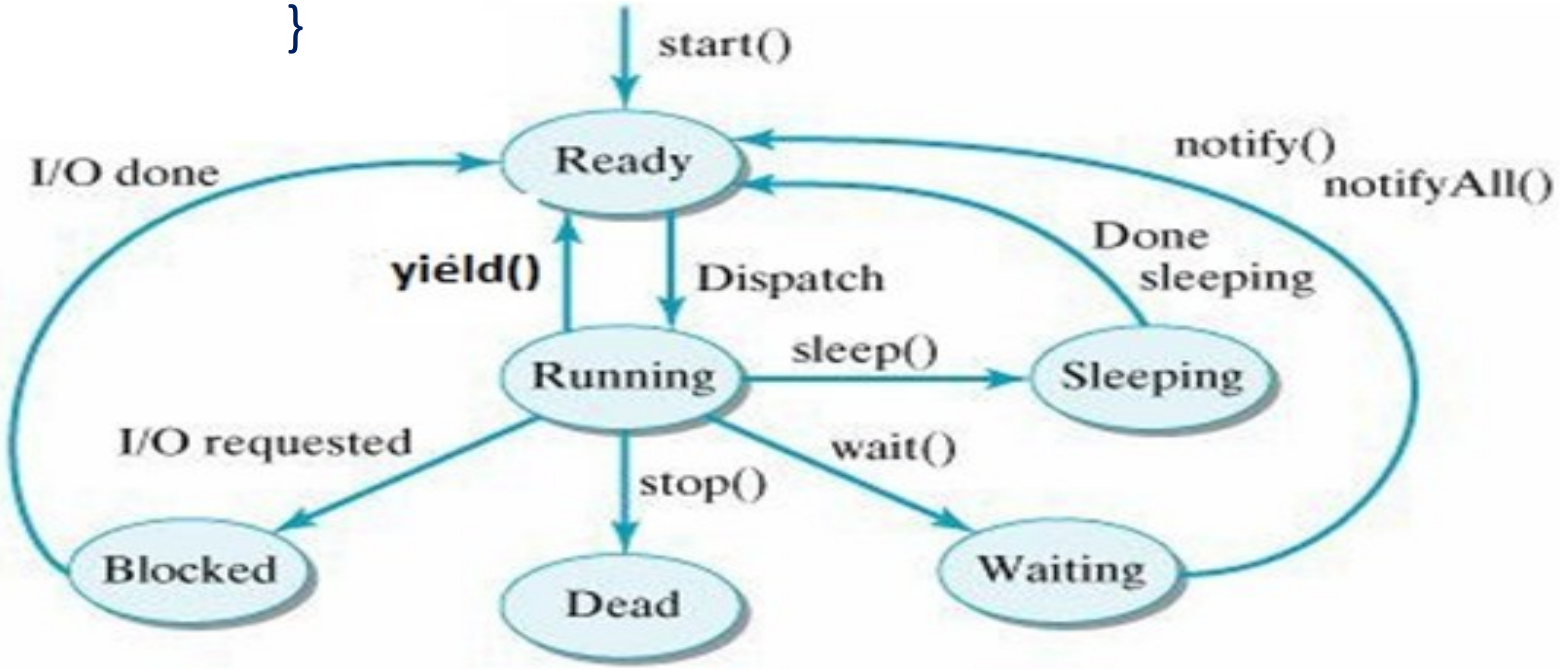
```
class Product{
  int p[10]; int k=0;
  boolean isAvailable=false;
   synchronized void produce() {
      if (isAvailable) { return; }
      for (int i = 0; i < 10; i++) {
       p[i]=k; } k++;
     }
  isAvailable=true;
 }
```

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

حل مشكلة استهلاك المنتج مرتين أو عدم استهلاكه
باستعمال حالات الترقب Monitoring status

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

```
synchronized void consume() {
    while(!isAvailable) { wait(); }
    System.out.println(p);
    isAvailable=false;
   notifyAll();
  }
}
```



معالجة التنافس في حال وجود أكثر من Instanceللغرض في الذاكرة ونسخة واحدة في قاعدة البيانات

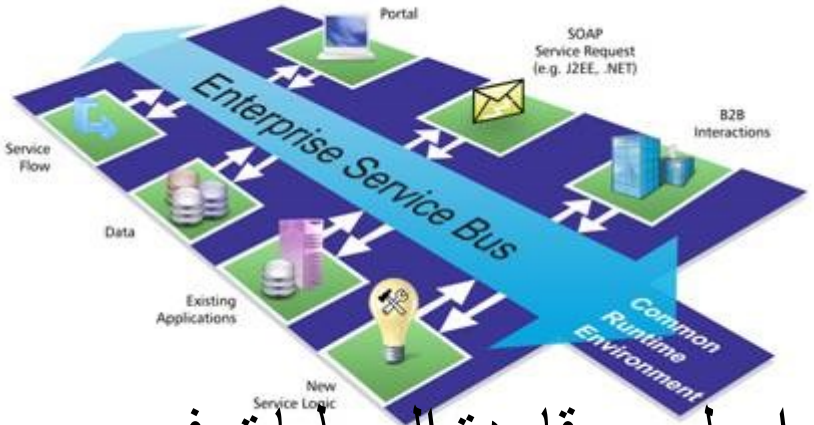
- **هل تحل مشكلة ATM هنا ؟**
  - في حالة ال ATM فان الغرض المتنافس عليه موجود بقاعدة المعطيات(data base) وليس بالذاكرة كما في الحالة السابقة لكن ما الفرق ؟
- **يوجد استراتيجيتين لحل المسالة في حال وجود الغرض المتنافس عليه في قاعدة المعطيات ؟**
  - **الإستراتيجية المتشائمة: وتفترض أن** أن التنافس على الموارد سيكون عالي مثل عداد للمستخدمين على الانترنت. استعمال أدوات مثل select for update سوف تؤدي لهبوط كبير في الأداء
- **الإستراتيجية المتفائلة :** متفائلة بأن الاحتمال قليل للتنافس على الموارد وتعتمد على إعطاء رقم إصدار للغرض المستخدم وتقوم بتعديل الإصدار عند كل تعديل يطرأ على هذا الغرض. ولا تقوم بقفل الغرض إلا أنها تضحي بالمناقلة فيما لو تغير الإصدار بين عمليتي القراءة والكتابة لأن ذلك يعني قيام طرف ثالن بتعديله

مواضيع أخرى في إدارة التنافس على مستوى قاعدة البيانات

- كيف تعمل المناقلات في قاعدة البيانات؟ متى تبدأ المناقلة ومتى تنتهي ومتى يحدث القفل وما هي الأغراض التي تقفل ومتى يتم تحريرها
- كيف يعمل ال Queue وكيف يدير التنافس بين المهام المختلفة وكيف نوجهه للقيام بذلك
- ما دور ال partitioning في التخفيف من التضارب ورفع عدد الاقفال ورفع الأداء

التنافس في حال وجود أكثر من قاعدة بيانات

- **وجود الغرض المتنافس عليه في أكثر من قاعدة بيانات وذلك بهدف**
  - أمني Replication في هذه الحالة يوجد قاعدة واحدة تعمل
  - أمني مع سرعة إضافية في القراءة ستعمل قاعدتي بيانات ولكن ستكون الكتابة أبطأ
  - كتابة في قاعدة واحدة والقراءة بدون تزامن مع قواعد البيانات الأخرى



حالة العمل online/offline

- مثلا في نظام بنك له عدة أفرع ولكل فرع قاعدة بيانات منفصلة وتتواصل مع قاعدة المعطيات في الفرع الرئيسي وبالتالي اذا أراد المستخدم سحب مبلغ من المال وكان الاتصال مع الفرع الرئيسي مقطوعا(لا يوجد شبكة انترنت) عندها سيتسبب بمشكلة لأنه يجب أن تتم هذه العمليات بشكل متزامن.
- إدارة المناقلة على الوب Web Transaction

Lazy Design Patterns: Lazy List

- Lazy List: هي إحدى الأليات للحد من استهلاك الذاكرة أثناء نقل المعطيات الكبيرة من قتاعدة البيانات والمستخدم.
- تعتمد على جلب كميات محدودة تباعاً والتخلص من الكمية السابقة قبل جلب الكمية التالية
- إذا كان التخاطب من مستخدم عبر GUI يظهر عادة للمستخدم زرين next/prev ليقوم بالتجوال

Lazy Design Patterns: Frames

- نربط بكل حقل Field ثلاثة عفاريت demons
  - If-Needed –
  - If-Added –
  - If-Removed –

Frame

- Class Rect {
  - Int length, width, area=length*width;
- }
- Frame Engine is composed of three demons
  - If added
  - If removed
  - If needed

Frame

	length	width	area
If needed	Length	width	Length*width
If added	Remove area	Remove area	-
If removed	Remove area	Remove area	-

Frame Advantages

- Easy to read
- Unified model for attributes and fucntions
- Better performance using “lazy fetch”
- Smaller code size, because the engine is predefined
- More intelligent using a third type of values “unkown”

How to define a frame using Java? “Basic solution”

- Class Rect {
  - Int width,length, area;
  - Boolean isAssW=true,isAssL=true,isAssA=false;
  - Rect (in w,l) {
  - Width=w; length=l;
  - }
  - getWidth() { return width;} // the same for getLegth
  - getArea() {
    - if(!isAssA) { area=length*width; isAssA=true;}
      return area;
  - }
  - setWidth(int w) { // the same for setLength
  - width=w; isAssA=false;
  - }

Student is asked to re-implement this program using JavaBeans listeners

Example of frame as a component

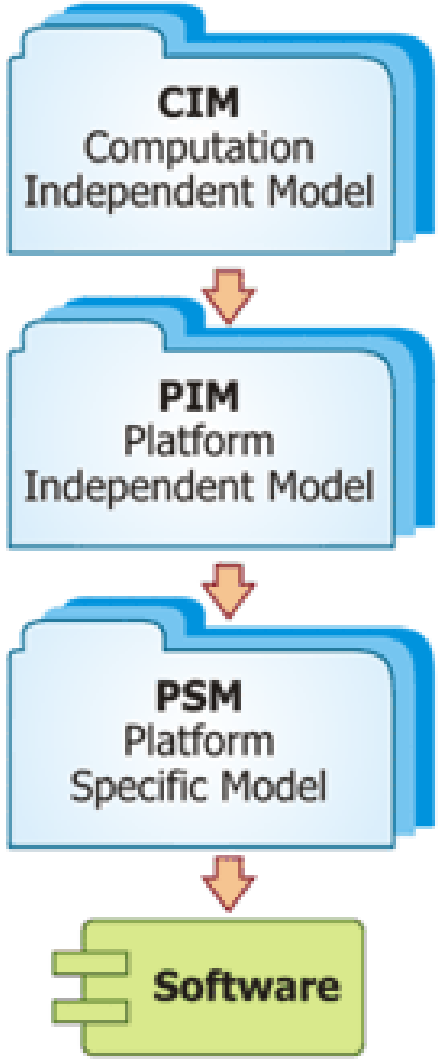
- Class Rect {
- int length, width, area=length*width;
- }

What are design patterns related to Reliability

- To increase reliability we need to
  - Decrease code size
    - Using interpreted programing languages such as Java Script, Lisp
    - Using declarative languages such as html, Prolog
    - Using Low code frameworks
    - Using MDP
    - Using Frame instead of normal classes
  - Enumerate Input
    - Using formal model for **recursively enumerable** languages not if-then-else (as we do in compilers)
    - Most important formal models: deduction, abduction and induction
    - AI Alg such as vision, recognitions, etc are not **recursively enumerable**

MDA

http://www.omg.org/mda/mda_files/Cephas_MDA_Fast_Guide.pdf



- **Computation Independent Model (CIM**)
  A CIM is also often referred to as a business or domain model because it uses a vocabulary that is familiar to the subject matter experts (SMEs). It presents exactly what the system is expected to do, but hides all information technology related specifications to remain independent of how that system will be (or currently is) implemented. The CIM plays an important role in bridging the gap which typically exists between these domain experts and the information technologists responsible for implementing the system. In an MDA specification the CIM requirements should be traceable to the PIM and PSM constructs that implement them (and vice-versa).
- **Platform Independent Model (PIM)**
  A PIM exhibits a sufficient degree of independence so as to enable its mapping to one or more platforms. This is commonly achieved by defining a set of services in a way that abstracts out technical details. Other models then specify a realization of these services in a platform specific manner.
- **Platform Specific Model (PSM)**
  A PSM combines the specifications in the PIM with the details required to stipulate how a system uses a particular type of platform. If the PSM does not include all of the details necessary to produce an implementation of that platform it is considered abstract (meaning that it relies on other explicit or implicit models which do contain the necessary details).

MDA and Induction

طرق التفكير المنطقي الصوري Formal

- الاستنتاج Deduction ويفيد التوقع (الاستنباط)
  - modus ponens: a,a->b -> b (KBS)
  - Modus tollens: !b, a->b -> !a (KBS)
  - Abduction: b,a->b -> a (Expert System, Fuzzy Logic)
- الاستقراء induction ويفيد في بناء قواعد استنتاج جديدة (Machine Learning)
  - a(1), a(2), a(3), … a(n) -> a(n+1)

المجموعات القابلة للعد بشكل عودي
recursively enumerable

- تقوم البرمجة على إيجاد علاقة بين الدخل والخرج أي بين المقدمات والنتائج لذلك لا بد من تعداد حالات الدخل ولدينا 3 حالات
  - مجموعات دخل قابلة للعد (برامج بسيطة جدا)
  - مجموعات قابلة للعد بشكل عودي (99% من البرمجيات)
  - مجموعات غير قابلة للعد (مسائل ذكاء صنعي)
- يفيد الاستقراء في عد عناصر مجموعة بشكل عودي

العد بالاستقراء البنيوي Structural induction

- يسمح يمكن تعريف مجموعة بالاستقراء من خلال بديهيات Axiom وقواعد بناء Rules
- الأعداد الطبيعية N

  – Axiom: $0 \in \mathbb{N}$

  – Rule1: $x \in \mathbb{N} \Rightarrow x+1 \in \mathbb{N}$

- مجموعة من الأعداد L

  – Axiom: $[] \in L$

  – Rule1: $l \in L \ \& \ x \in \mathbb{N} \Rightarrow [x|l] \in L$

Structural induction الاستقراء البنيوي

- مجموعة من مجموعات GL
  - Axiom: [] $\in$GL
  - Rule1: L$\in$GL & x $\in\mathbb{N}\Rightarrow$[x|L] $\in$GL
  - Rule2: L$\in$GL & c $\in$ GL $\Rightarrow$[c|L] $\in$GL

التقطيع البنيوي Structural decomposition

- يسمح بكتابة غرض ما بدلالة مكونات أبسط وفقا لعكس قواعد الاستقراء
- يكتب xعلى الشكل 1+(x-1)
- تكتب [3 4 5] على الشكل ((5 4)|3]
- تكتب [[6 7] 5 [2 3]] على الشكل
  [[[6 7] 5] | [2 3]]

البرمجة والاستقراء

- إذا كتبنا غرض ما بدلالة مكوناته فإننا نستطيع حل أي مسألة بطريقة عودية كما يلي
- If $x = x_0 \circ x_1 \circ \ldots \circ x_n$

$$f(x)=g(f(x_0),f(x_1),\ldots,f(x_n))$$

- بمعنى آخر فإن عدد الحالات في أي عملية مهما تكن هو بعدد احتمالات التقطيع البنيوي لا أكثر
- أمثلة
  - $n! = n(n-1)!$
  - $\sum n = n + \sum(n-1)$
  - $\sum[x|L] = x + \sum L$
  - $In[a,[x|L]] = [x==a]||in[a,L]$

البرمجة والاستقراء

- تفيد نظرية الاستقراء في كتابة البرامج بطريقة موحدة دون تفكير
- يسمح ذلك بسرعة في البرمجة
- سهولة في الصيانة
- سهولة في العمل ضمن فريق
- سهولة في نقل البرمجيات من جيل لآخر

مع أن النظرية غير مرتبطة بلغة إلا أنه من الاسهل رؤيتها ضمن لغات الذكاء الصنعي مثل الLisp وProlog

أمثلة بسيطة بجافا

- long fact(int n) {
  - If(n==0) return 1;
  - return n*fact(n-1);
- }

- long sum(int n) {
  - If(n==0) return 0;
- return n+sum(n-1);
- }

أمثلة بسيطة Prolog

- fact(0,1).
- fact(n,X):- X1 is n-1, fact(X1,X2), X is n*X2.

- sum(0,0).
- sum(N,M):-N1 is N-1,sum(N1,M1), M is N+M1.

-

أمثلة على مجموعة بسيطة بجافا

- long sum(List l) {
- - if(l.isEmpty()) return 0;
- - return l.first()+sum(l.rest());
- }

- Boolean find(int n, List l) {
- - if(l.isEmpty()) return false;
- - return n==l.first() || find(n,l.rest());
- }

- List reverse (List l) {
- - if(l.isEmpty()) return l;
- - return l.addLast(l.first(), reverse(l.rest()));
- }

أمثلة على مجموعة بسيطة بجافا

- List addLast(int n, List l) {
- - if(l.isEmpty()) return l.add(n);
- - return l.add (l.first(), addLast(n,l.rest()));
- }

- List union(List l1, List l2) {
- - if(l1.isEmpty()) return l2;
- - return find(l1.first(),l2) ? union(l1.rest(),l2) : union(l1.rest(),l2).add(l1.first()) ;
- }

- List intersect(List l1, List l2) {
- - if(l1.isEmpty()) return l1;
- - return find(l1.first(),l2) ? intersect(l1.rest(),l2).add(l1.first() : intersect(l1.rest(),l2);
- }

أمثلة على مجموعة بسيطة Prolog

- sum([],0).
- sum([X|Y],M):-sum(Y,M1), M is X+M1.

- find(_,[]):-!,fail.
- find(X,[X|_]):-!.
- find(X,[_|Y]):-find(X,Y).

- reverse([],[]):-!.
- reverse([X|Y],Z):-reverse(Y,Y1),addLast(X,Y1,Z).

أمثلة على مجموعة بسيطة Prolog

- addLast(X,[],[X]):-!.
- addLast(X,[C|CDR],[C|Z]):- addLast(X,CDR,Z).

- addFirst(X,Y,[X|Y]).

- union([],Z,Z):-!.
- union([X|Y],Z,Z1):-find(X,Z),!,union(Y,Z,Z1).
- union([X|Y],Z,[X|Z1]):-union(Y,Z,Z1).

- intersect(_,[],[]).
- intersect([Y|Y1],X,[Y|Z1]):-find(Y,X),!,intersect(X,Y1,Z1).
- intersect([_|Y1],X,Z):-intersect(X,Y1,Z).

أمثلة على مجموعة عميقة بجافا

```
long sum(List l) {
- if(l.isEmpty()) return 0;
- return isInteger(l.first())? l.first()+sum(l.rest()):
-                              sum(l.first())+sum(l.rest());
}

Boolean find(int n, List l) {
- if(l.isEmpty()) return false;
- return isInteger(l.first()) ? (n==l.first() || find(n,l.rest())) :
-                                find(n,l.first()) || find(n,l.rest())) ;
}

List reverse (List l) {
- if(l.isEmpty()) return l;
- return l.addLast(reverse(l.first()), reverse(l.rest()));
}
```

أمثلة على مجموعة عميقةProlog

- sum([],0).
- sum([X|Y],M):-number(X),!,sum(Y,M1), M is M1+X.
- sum([X|Y],M):-sum(X,M1),sum(Y,M2), M is M1+M2.

- find(_,[]):-!,fail.
- find(X,[X|_]):-!.
- find(X,[Y1|Y]):-number(Y1),!,find(X,Y).
- find(X,[Y1|Y]):-find(X,Y1);find(X,Y).

أمثلة على مجموعة عميقة Prolog

- reverse([],[]):-!.
- reverse([X|Y],Z):-number(X),!,reverse(Y,Y1),addLast(X,Y1,Z).
- reverse([X|Y],Z):-reverse(X,X1),reverse(Y,Y1),addLast(X1,Y1,Z)

نماذج أخرى باستعمال المجموعات المعممة

- مصفوفة
  - [[a11, a12, a13],[a21, a22, a23],[a31,a32,a33]]
- بيان
  - [[n1, n2, n3, n4],[[n1,n2],[n1,n3],[n2,n4]]
- صف مثال طالب باستعمال associative list
  - [[name, ahmad], [last name, haddad], [age, 18], .
- شجرة ثنائية
  - [value, left, right]
  - [10,[5,[3],[6]],[15,[11],[16]]]

Meta Data نموذج الأنماط السامية

- Axiom: int, double, string $\in$MetaData
- Rule: l1 $\in$MetaData & l2 $\in$Meta Data$\Rightarrow$[x|L] $\in$MetaData

Remarque : Simple, Composite, Collection are semantic level

Example of Meta Data using MDP

Car	Code id		Unique id,		Label mark + manufacturer+id	Integrity productionDate< registration	isReadOnly if we did use the car
Field name	**ID**	**Type**	**isMandatory**	**isUnique**	**isModifiable**	**description**	
id	**id**	**String**	**yes**	**yes**	**no**	**A string starts with a character and has no spaces**	
mark	**mark**	**Mark**	**yes**	**no**	**Yes before using the car**	**One of predefined marks**	
manufacturer	**manufact**	**Manufacturer**	**yes**	**no**	**Yes before using the car**	**One of predefined manufacturer**	
color	**color**	**Color**	**yes**	**no**	**Yes before using the car**	**One of predefined color**	
production date	**productionDate**	**Date**	**no**	**no**	**yes**	**A date greator than 1900**	
registration	**regostration**	**Date**	**no**	**no**	**yes**	**A date than 1900**	

The student is asked to try to implement PMI to PSM during TP

Example1: of workflow steps

- apply [view,edit,dailyLeave,hourlyLeave] on Vacation [?fromDateRep, ?toDateRep,employee , employee.mainPosition.posAgency, vacationType, vacationKind, vacationFinType , !comment, !fromDate, !toDate] -RW where [employee.login=="GM"] -m

Example1: of workflow steps

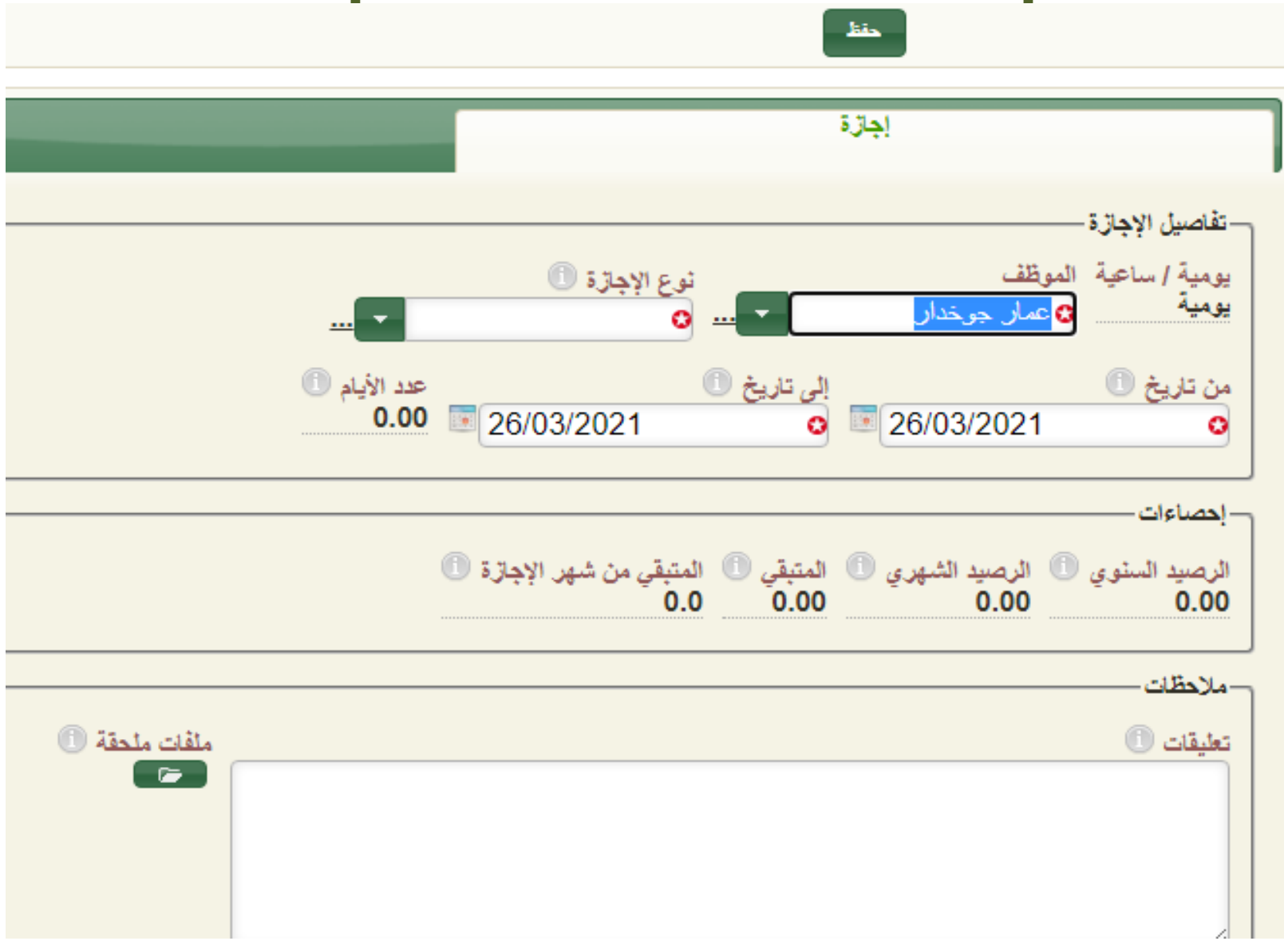
#	الموظف	الفرع	يومية / ساعية	نوع الإجازة	مدفوعة / غير مدفوعة	تعليقات	من تاريخ	إلى تاريخ
1	GM GM	Head Quarter	يومية	Annual Leave	مدفوعة		20/03/2016	20/03/2016
2	GM GM	Head Quarter	يومية	Annual Leave	مدفوعة		21/03/2016	24/03/2016
3	GM GM	Head Quarter	يومية	Annual Leave	مدفوعة		21/03/2016	21/03/2016
4	GM GM	Head Quarter	يومية	Annual Leave	مدفوعة		22/03/2016	22/03/2016
5	GM GM	Head Quarter	يومية	Annual Leave	مدفوعة		22/03/2016	22/03/2016
6	GM GM	Head Quarter	يومية	Annual Leave	مدفوعة		23/03/2016	23/03/2016
7	GM GM	Head Quarter	يومية	Annual Leave	مدفوعة		23/03/2016	23/03/2016
8	GM GM	Head Quarter	يومية	Annual Leave	مدفوعة		23/03/2016	23/03/2016
9	GM GM	Head Quarter	يومية	Annual Leave	مدفوعة		01/03/2016	01/03/2016
10	GM GM	Head Quarter	يومية	Annual Leave	مدفوعة		01/05/2016	01/05/2016
11	GM GM	Head Quarter	ساعية	Annual Leave	مدفوعة		01/04/2016	01/04/2016
12	GM GM	Head Quarter	ساعية	Annual Leave	مدفوعة		15/11/2018	15/11/2018
13	GM GM	Head Quarter	يومية	Annual Leave	مدفوعة		28/03/2019	28/03/2019
14	GM GM	Head Quarter	ساعية	Annual Leave	مدفوعة		30/04/2019	30/04/2019

Example2: of workflow steps

- apply [!save] on Vacation(employee$id=RTC.currentUserId();
- vacationType="daysVacation")

[{"LeaveDetailes",{{!vacationType,~employee,~vacationKind},{~*fromDate,~*toDate,!vacDurationInDays}}},{"Statistics",{{!vacationTypeBalance,!monthlyvacTypeBalance,!remaining,!monthRemaining}}},{"Other",{{comment,file}}},!monthlyVac] -RW

Example2: of workflow steps

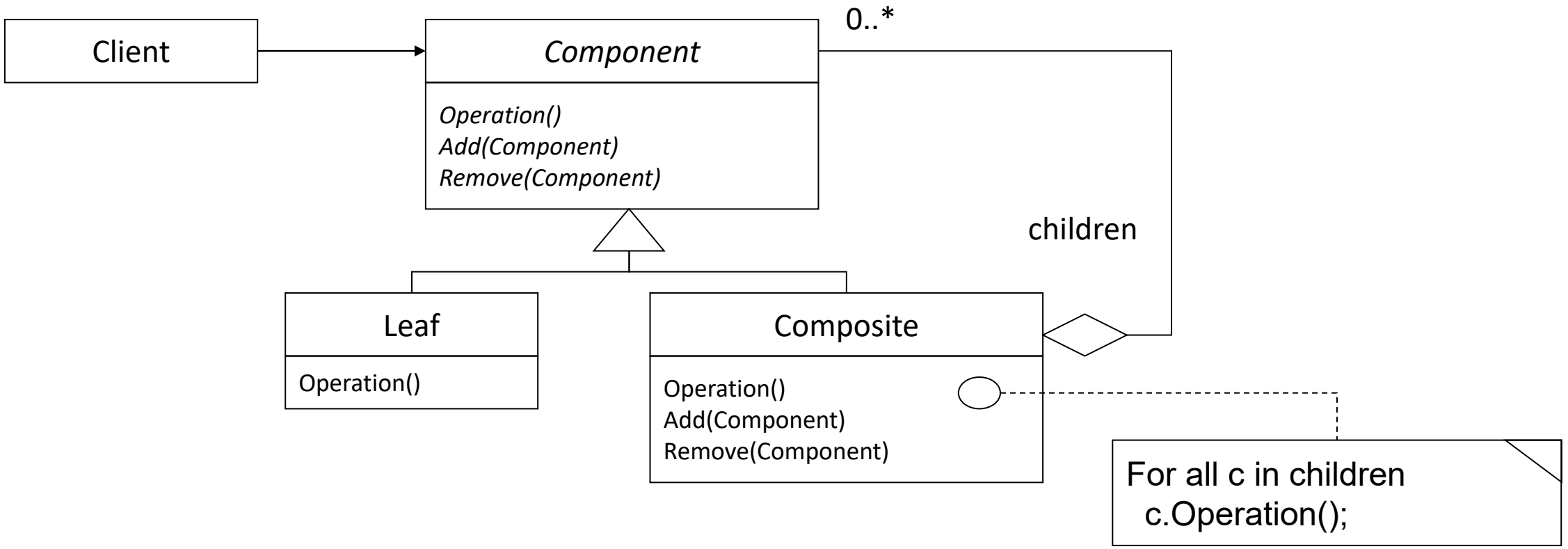


MDP: cont

- MDP is not limited to frame engines it covers several programming aspect including
  - DB via ORM and DAO
  - Presentation and views
  - Workflows using rules (ex Prolog)

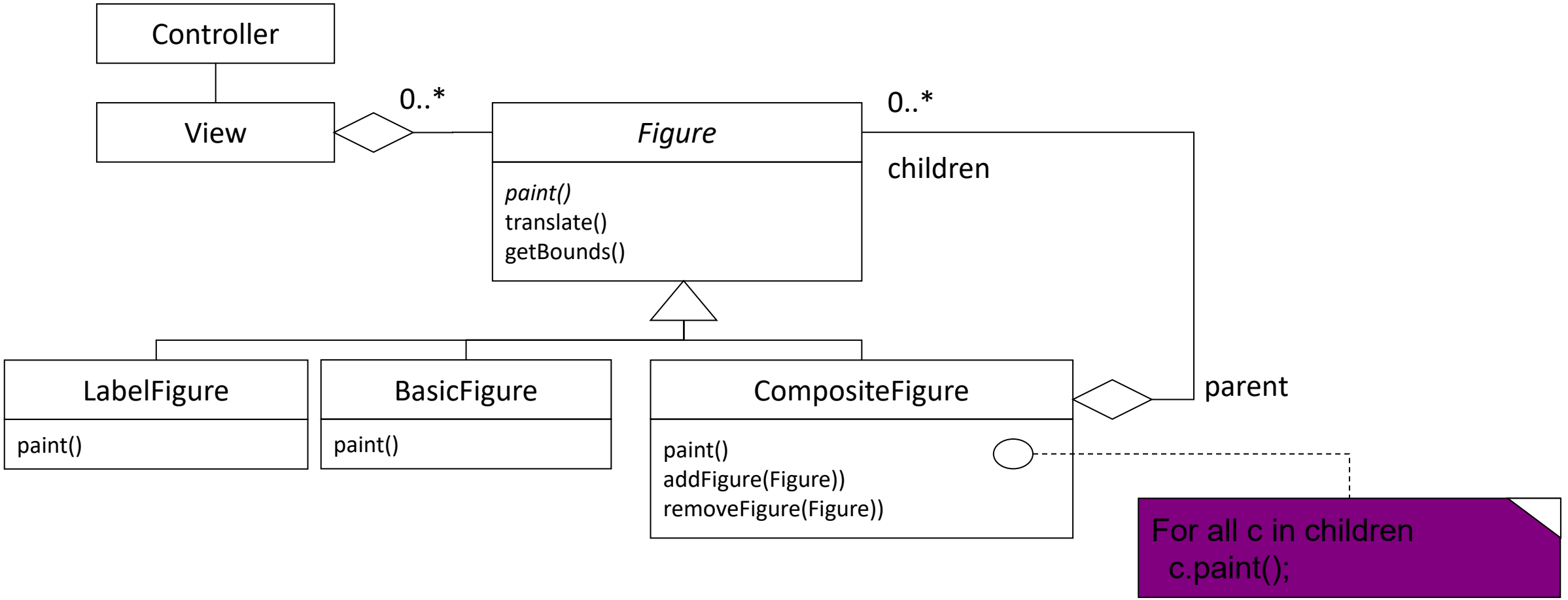
Composite

- Construct part-whole hierarchy
- Simplify client interface to leaves/composites
- Easier to add new kinds of components



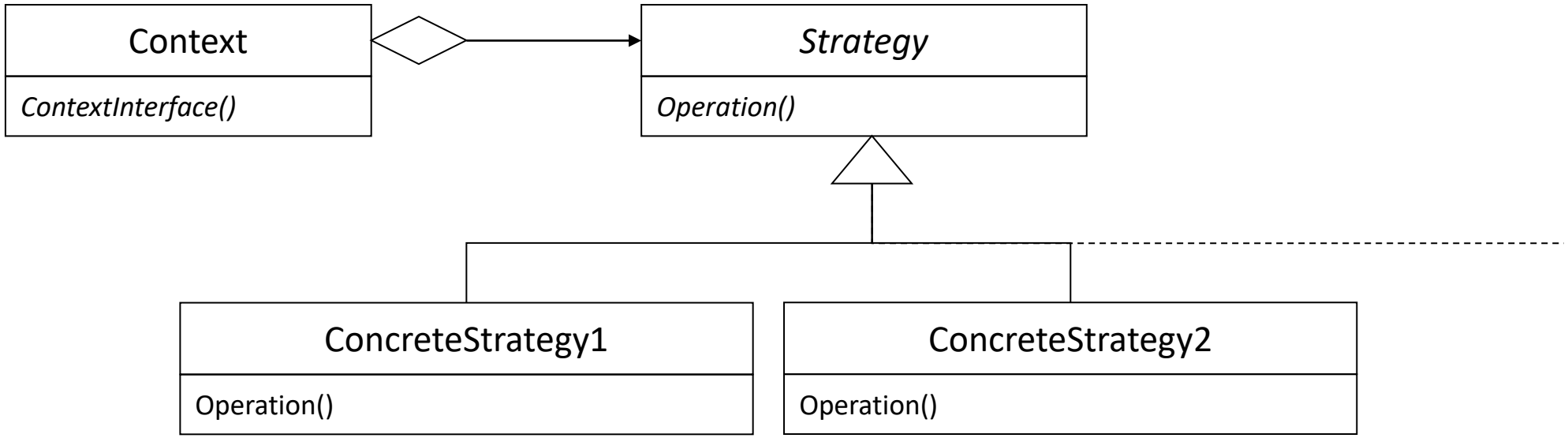
Composite (2)

- Example: figures in a structured graphics toolkit



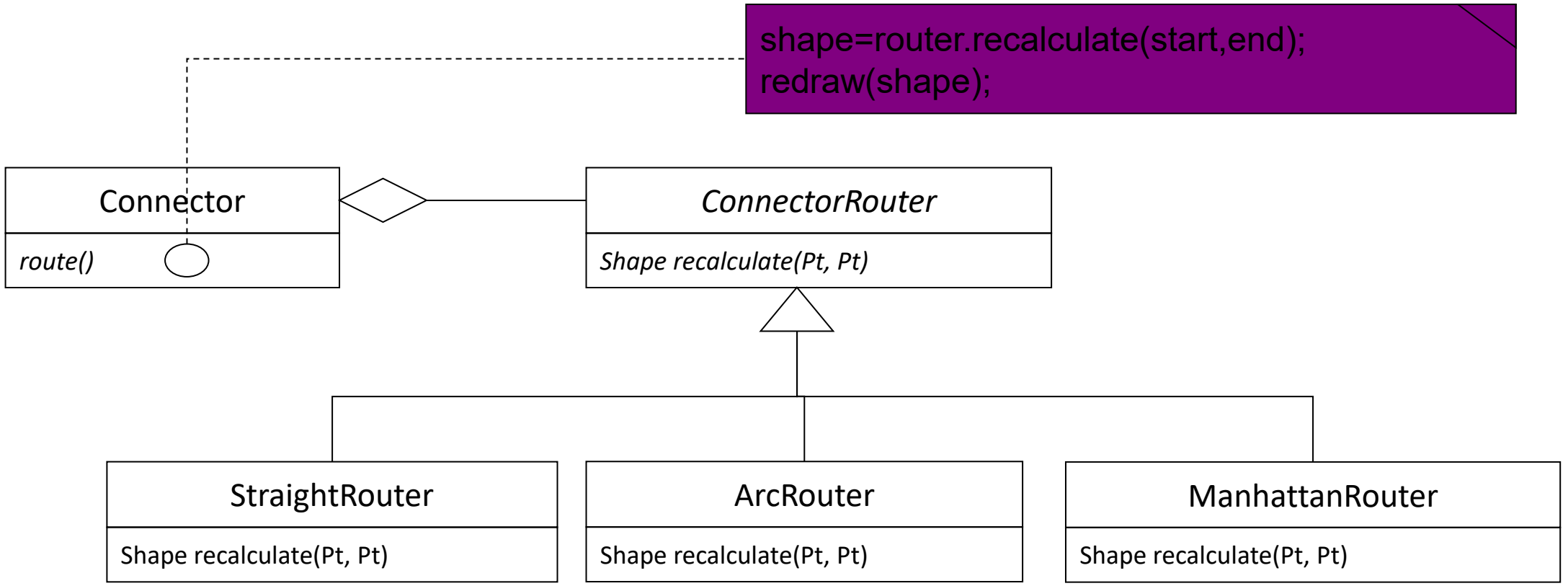
Strategy

- Make algorithms interchangeable---”changing the guts”
- Alternative to subclassing
- Choice of implementation at run-time
- Increases run-time complexity



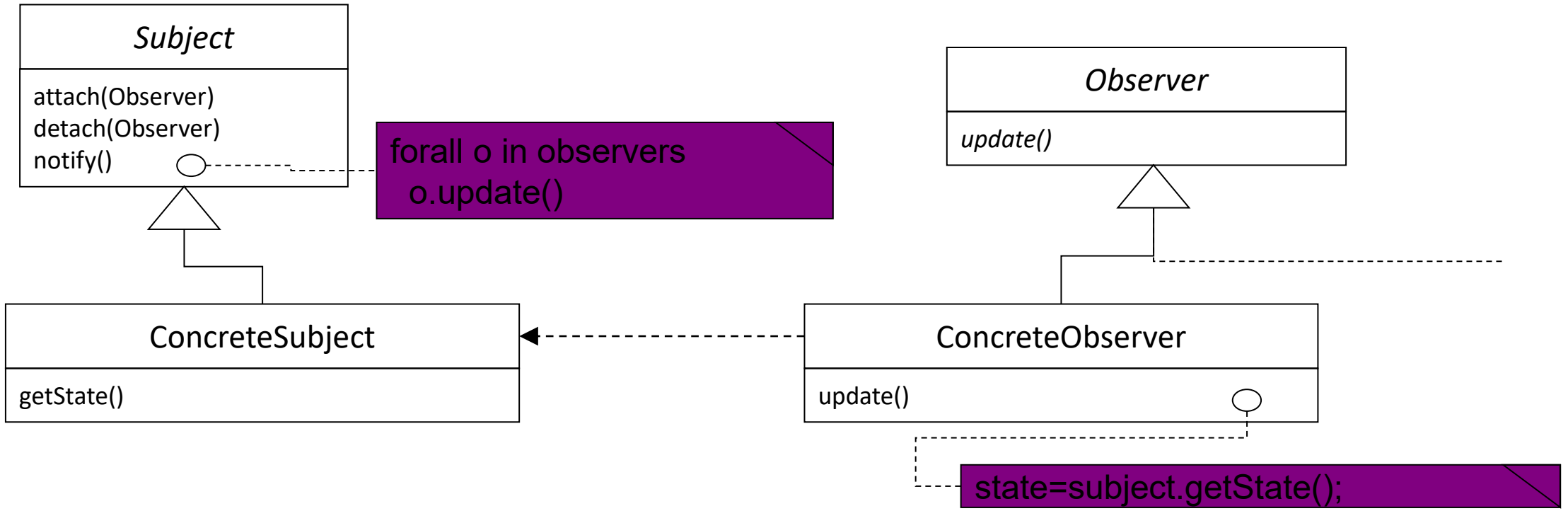
Strategy (2)

- Example: drawing different connector styles



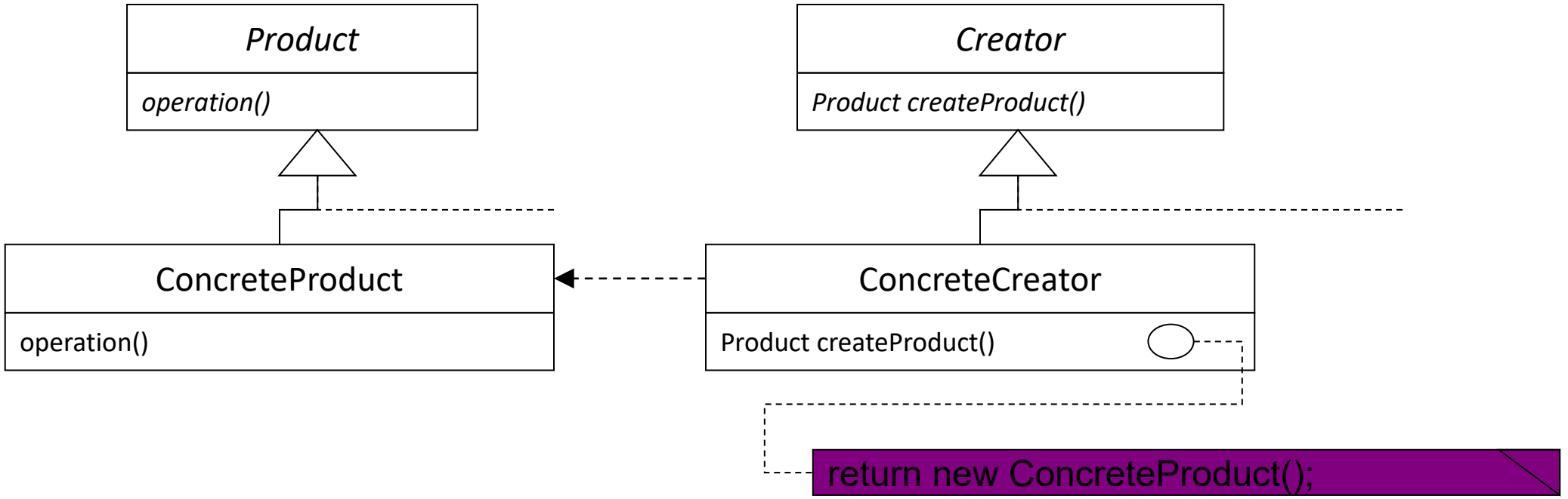
Observer

- Many-to-one dependency between objects
- Use when there are two or more views on the same “data”
- aka “Publish and subscribe” mechanism
- Choice of “push” or “pull” notification styles



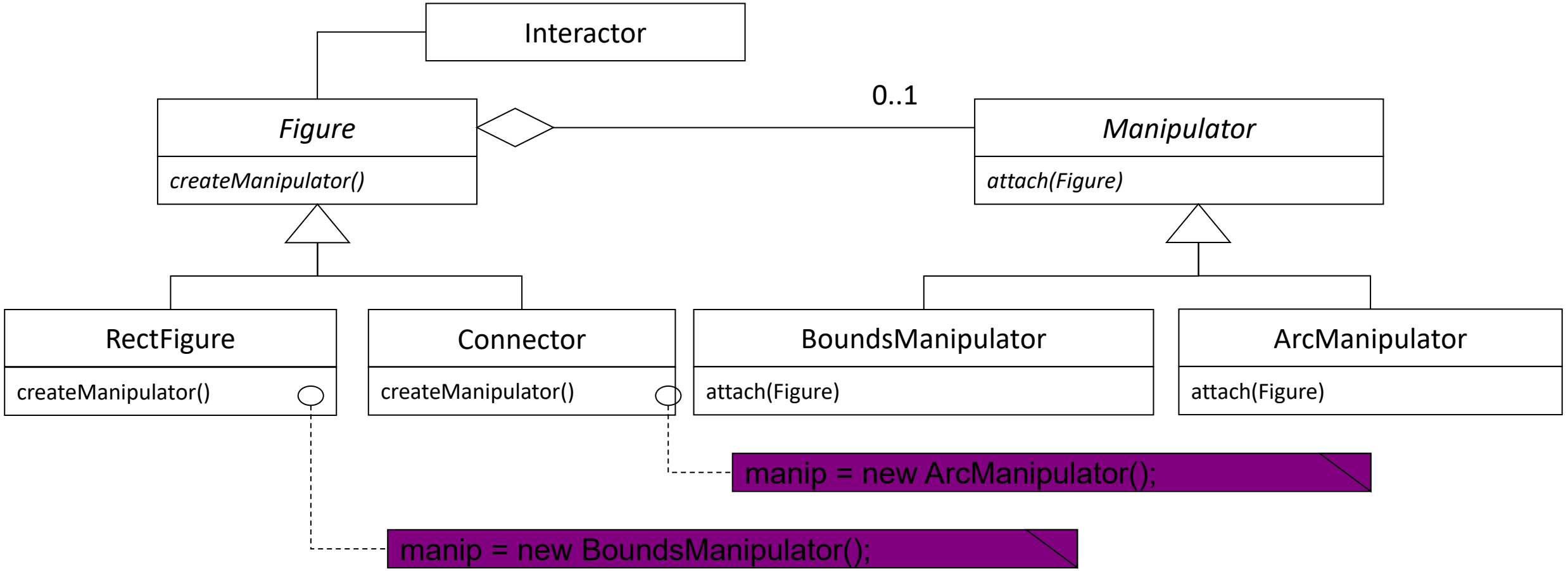
Factory Method

- Defer object instantiation to subclasses
- Eliminates binding of application-specific subclasses
- Connects parallel class hierarchies
- A related pattern is AbstractFactory

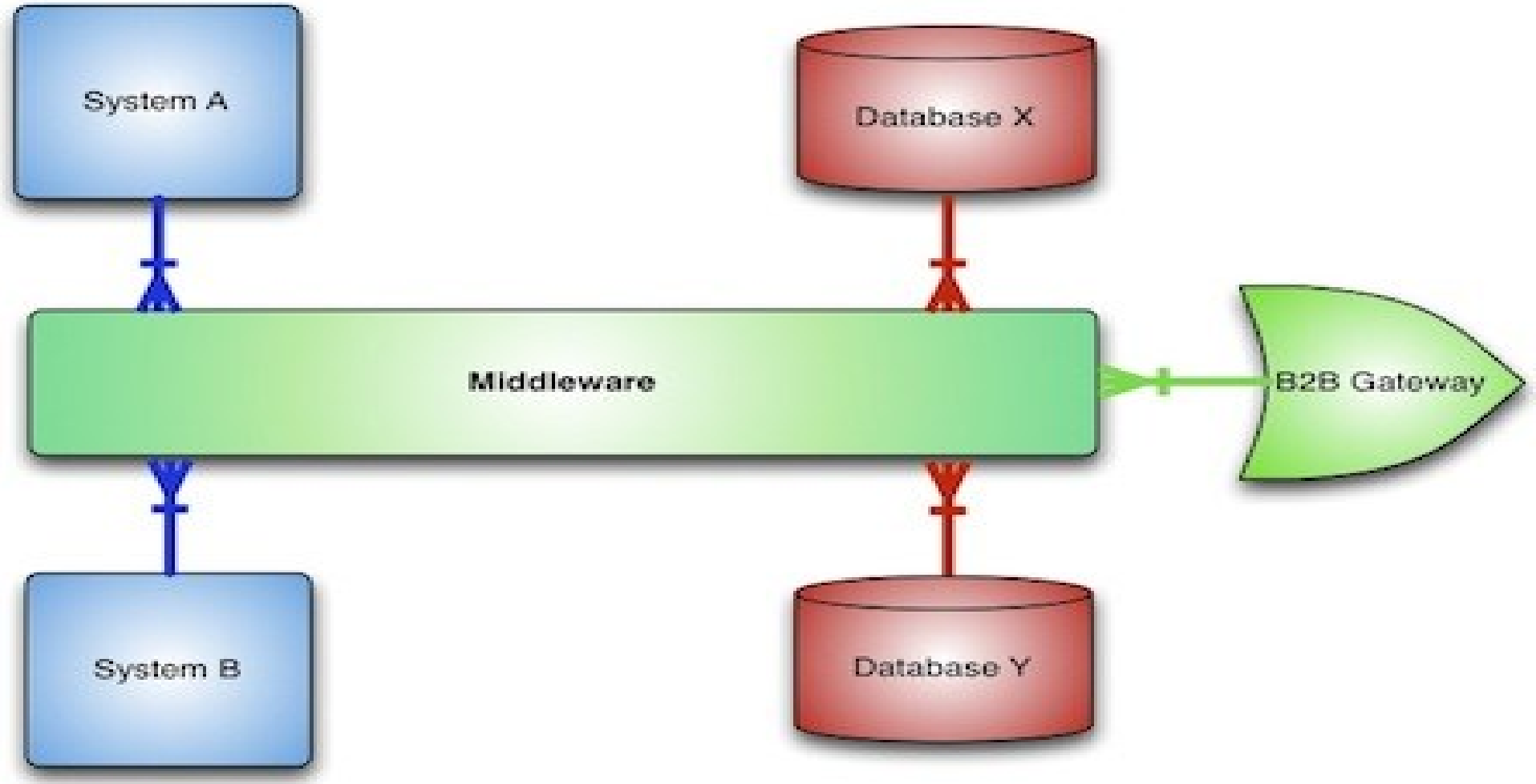


Factory Method (2)

- Example: creating manipulators on connectors



البرمجيات الوسيطة



ما هي البرمجيات الوسيطة
Middle Wares

- التطبيق هو عبارة عن برنامج أو عدة برامج وكل برنامج يتكون من مهمة أو عدة مهام Process و كل مهمة هي عبارة عن Thread واحد أو أكثر.
- لكل process لها stack و heap خاصة بها (انفصال بالذاكرة)
- الـ threads لا تشترك بالـ stack لكنها تشترك بالـ heap وهذا ما يجعل التخاطب بين الـ threads التي تنتمي لنفس الـ process ممكناً لأن الذاكرة مشتركة, بينما الـ processes تحتاج لـ MW كي تتخاطب مع بعضها البعض.
- لايمكن للـ Processes التخاطب إلا بوجود Middle Wares . مثال: لايستطيع المستخدم الاتصال مع سيرفر إلا عبر web service http

خصائص البرمجيات الوسيطة

- الاستقلال عن منصة العمل (نظام التشغيل ولغة البرمجة والعتاديات)
- دعم اتصال نقطة لنقطة point to point والإذاعة Broadcasting
- يدعم المناقلات Transactions.
- الوثوقية: أن تصل الرسالة مرة ومرة واحدة فقط
- التصعدية أو التلاؤم مع تزايد الحركة Traffic
- دعم التخاطب المتزامن وغير المتزامن.
- دعم بروتوكول تخاطب
- دعم الاستدعاء بالاسم
- تنوع بنى الاتصال :امكانٌة العمل عن طرٌق .etc,...wireless,sockets

ليس بالضرورة أن يدعم MW جميع الخصائص السابقة فهو يدعم الخصائص التي تلائم وتلبي حاجة النظام.

أنواع التواصل بين البرمجيات

- **Extraction , Transformation & Load : ETL**
  http://youtu.be/K_FCHYWGGug
- **Enterprise Application Integration : EAI**

	ETL	EAI
transactions	few	many
amount of data	enormous	small
integration	data	applications
purpose	operations & strategic	operations
business case	business intelligence	IT, e-business
	decision making	better workflow
	one view of customer	data entry once
common	transformations	
	meta data	
	connectors	
	service oriented architectures	

الأنواع المختلفة لـ Middleware

- Client/server
- 3 tiers client/server
- Queue
- RPC
- Object Oriented Middleware
  - RMI
  - CORBA
  - EJB
  - COM+
  - Web services

زبون/ مخدم Client/Server

- في هذه الحالة نستعمل Socket للتواصل بين التطبيقات ويقسم العمل كما يلي
- الزبون: يهتم بالواجهة التخاطبية وقد يهتم بمنطق العمل
- المخدم يهتم بالتخزين وقد يهتم بمنطق العمل وهو الأفضل كنموذج تصميم.
- مثال برمجي بلغة جافا

Client	Server	
GUI + BL	DB	المنهجية 1 Thick Client
GUI	DB + BL	المنهجية 2 Thin Client

Client/Server example

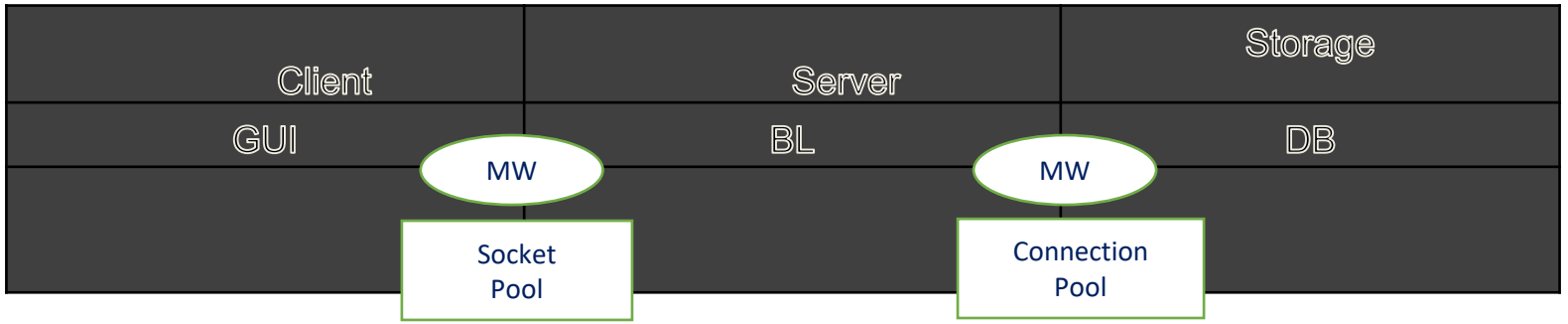
```
Class Server{

  Public static void main(){

ServerSocket  ss = new ServerSocket(4000);
ServerSocket  ss = new ServerSocket(4000);
      while (true){
          Socket  cs = ss.accept();
         InputStream  is = cs.getInputStream();
         OutputStream  os = cs.getOutputStream();
         dealWith(is,os);
}
}
```

```
Class client{

   Public static void main(){

ClientSocket  cs = new ClientSocket(ip,4000)

ls.os;

dealWith(os,s)



}
```

محدودية الـClient/server

الاستقلال عن منصة العمل	نعم
دعم Broadcasting/ point to point	Point to point only
دعم Transactions	لا
الوثوقية: أن تصل الرسالة مرة ومرة واحدة فقط	يعتمد على المبرمج
التصعدية أو التلاؤم مع تزايد الحركة Traffic	لا
دعم التخاطب المتزامن وغير المتزامن.	متزامن فقط
دعم بروتوكول تخاطب	لا
دعم الاستدعاء بالاسم	لا
تنوع بنى الاتصال :امكانُة العمل عن طرُق .etc,...wireless,sockets	socket

3 Tiers client/server

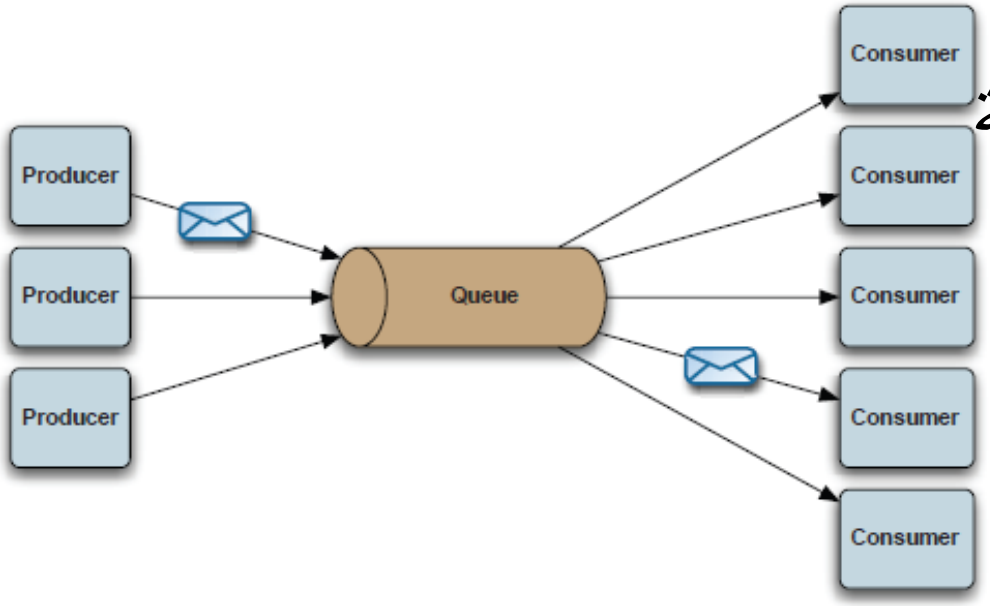


محدودية 3Tiers Client/Server

الاستقلال عن منصة العمل	نعم
دعم Broadcasting/ point to point	Point to point only
دعم Transactions	لا
الوثوقية: أن تصل الرسالة مرة ومرة واحدة فقط	يعتمد على المبرمج
التصعدية أو التلاؤم مع تزايد الحركة Traffic	نعم
دعم التخاطب المتزامن وغير المتزامن.	متزامن فقط
دعم بروتوكول تخاطب	لا
دعم الاستدعاء بالاسم	لا
تنوع بنى الاتصال :امكانيّة العمل عن طرُق .etc,...wireless,sockets	socket
أصبح الأمن من مسؤولية طبقة BL وبالتالي أصبح هناك صعوبة أكبر بكثير في البرمجة مقارنة مع client/server	

الطابور Queue

- برنامج وسيط بسيط ولايستعمل بروتوكولات النظام ولايستدعي تعليمات نظام ويؤمن التخاطب غير المتزامن
- تقنية ناضجة سبق واختبرت مراراً في مجالات تطبيقية
- لا تحتاج لآخر ما توصل إليه العلم في مجال البرمجة الغرضية التوجه وهذا هام لتتمكن التقنية من ربط برمجيات قديمة لا تعتمد هذه التقنية مع اخرى حديثة
- لا يوجد معيار ولا يمكن الخلط بين المنتجات
- أمثلة صناعية:
  - BEA MessageQ
  - IBM proposes MQSeries

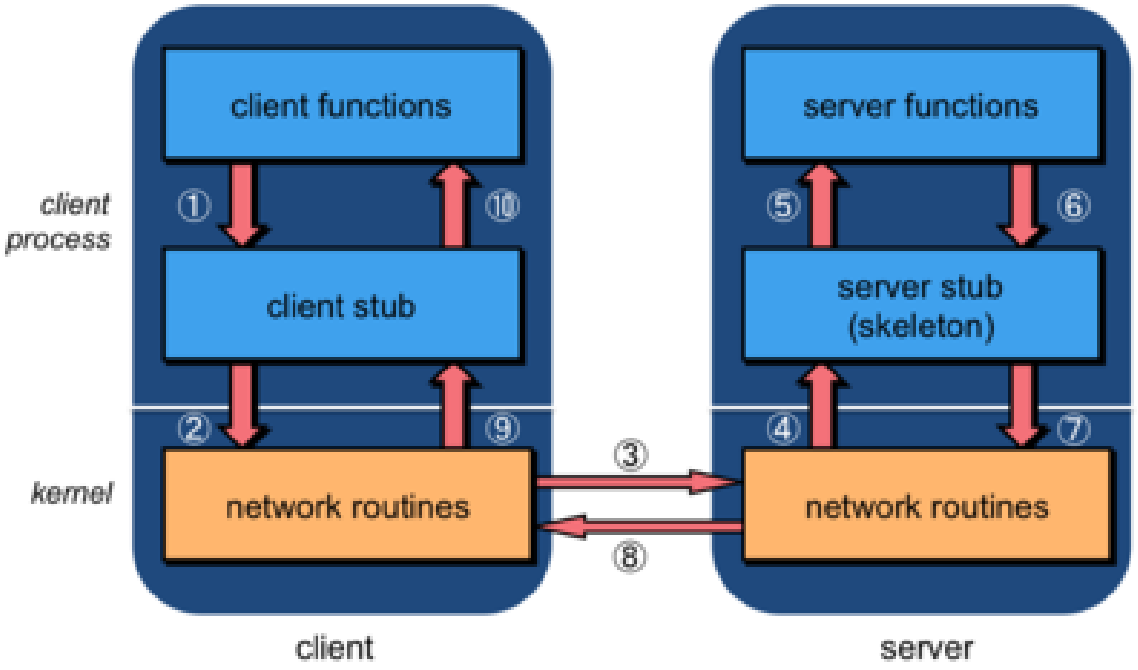


الطابور Queue

الاستقلال عن منصة العمل	يتوفر على منصات Unix, IBM/OS2, Windows NT
دعم Broadcasting/ point to point	نعم
دعم Transactions	نعم
الوثوقية: أن تصل الرسالة مرة ومرة واحدة فقط	يضمن وصول الرسائل وذلك بالمحافظة على نسخة منها على القرص الصلب لا تهدم ما لم تعالج الرسالة
التصعدية أو التلاؤم مع تزايد الحركة Traffic	نعم
دعم التخاطب المتزامن وغير المتزامن.	نعم
دعم بروتوكول تخاطب	لا
دعم الاستدعاء بالاسم	لا
تنوع بنى الاتصال :امكانُة العمل عن طرُق etc,...wireless,sockets.	نعم
يسمح بتحقيق ممتاز لنموذج Producer/Consumer بدون أي جهد	

RPC

- يسمح الـ Queue بمكاملة تطبيقات موجودة مسبقاً لأنه لا يضع أي قيود على بنية الاتصال.
- إذا أردنا تطوير برنامج جديد وتوزيعه على عدة آلات فسنحتاج لما هو أسرع.
- يتكون التطبيق من :
  - Client : main procedure
  - Server: procedure
- يمكن أن يكتب الـ Client بلغة مختلفة عن المخدم --> نحتاج للغة عامة وهي IDL (Interface Definition Language) ويسمى الوسيط بـ RPC (Remote Procedure Call).



RPC

- في حالة الطابور فإن code اللازم للتخاطب يشكل جزءاً من client code ويجب على المبرمج كتابته في حين أن client في RPC لا نفرق بين استدعاء محلي واستدعاء RPC فإن الـcode اللازم للتخاطب يولد آلياً اعتباراً من لغة IDL التي توصف واجهة المخدم المستعمل.
- نسمي الجزء من الرماز المرتبط بالزبون بـ client stub
- نسمي الجزء من الرماز المرتبط بالمخدم بـ server stub

خصائص أساسية للـ RPC

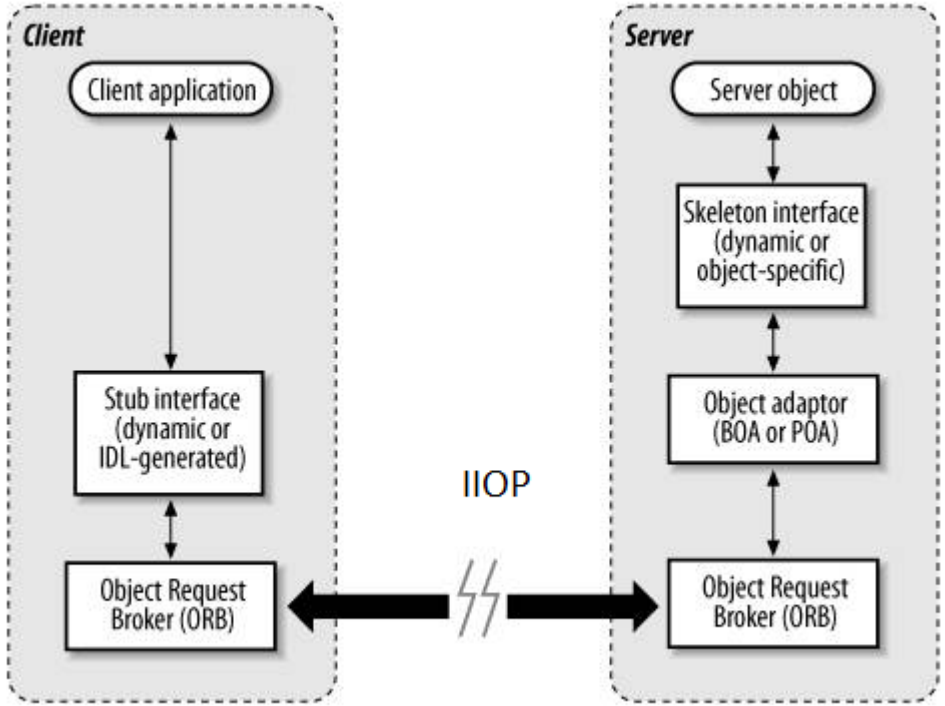
- رماز المخدم والزبون مستقل عن نظام التخاطب ولا يشعر الزبون بفرق بين الاستدعاء المحلي والاستدعاء عن بعد
- لا يحتاج الزبون لتحديد مكان المخدم أو تحضير الرسالة فهذا دور البرنامج الوسيط
- نظام التحادث مستقل عن المخدم والزبون ومكتوب بالـ IDL يُولد الرماز اللازم للتخاطب أثناء الترجمة
- تولد البنية الازمة للتخاطب (بروتوكول، ...) أثناء الترجمة
- التخاطب متزامن
- يوجد معيار تستخدمه الشركات المنتجة يسمى OSF/DCE أو Open Software Foundation/ Distributed Computing Envirenment -->
- يمكن مكاملة أكثر من RPC middleware

مساوئ RPC

الاستقلال عن منصة العمل	نعم
دعم Broadcasting/ point to point	لا
دعم Transactions	لا
الوثوقية: أن تصل الرسالة مرة ومرة واحدة فقط	نعم
التصعدية أو التلاؤم مع تزايد الحركة Traffic	لا
دعم التخاطب المتزامن وغير المتزامن.	متزامن فقط
دعم بروتوكول تخاطب	نعم
دعم الاستدعاء بالاسم	لا
تنوع بنى الاتصال :امكانُة العمل عن طرُق wireless,sockets...,etc.	لا
لم تحقق هذه التكنولوجيا نجاحاً حقيقياً حيث أن فكرة procedure هي فكرة متدنية المستوى لا تظهر في مرحلة التحليل ولا التصميم وإنما تظهر متأخراً في مرحلة البرمجة مما يضطرنا لتغيير الطرق الحالية في التحليل والتصميم لإظهارها في مراحل متقدمة	

(COM/ CORBA) Object Oriented Middleware

- رأينا محدودية RPC كونها تستعمل الـ Procedure كوحدة توزيع كونها متدنية المستوى. ما نحتاجه هو بنية عالية المستوى تظهر في مرحلة التحليل وتكون قريبة أيضاً من التصميم لنستطيع تحويلها إلى لغة برمجة. هذه البنية موجودة وتسمى بالغرض Object.
- للغرض ميزات عديدة منها Encapsulation، تحقيق العديد من Interfaces، والوراثة Inheritance .
- إذا افترضنا أن الغرض هو وحدة التوزيع فسيتم التخاطب عن طريق استدعاء عملية method من غرض ما نسميه المخدم. نسمي البرنامج الوسيط الذي يؤمن هذا الاستدعاء بـ Object Middleware أو ORB (Object Request Broker) أي وسيط الطلبات الخاصة بالأغراض.
- يوجد معياران:
  - الأول هو معيار عالمي وضعته OMG (Object Management Group) و هو موجود منذ عام 1990 ويسمى CORBA أي Common Objec Request Broker Architecture
  - الثاني منذ 1996 وهو لـ Microsoft ويسمى DCOM (Distributed Component Object Model)
- تطورت CORBA سريعاً بسبب تأثير جافا حيث يؤكد المعيار 3.0 على فكرة المكونات البرمجية التي جسدتها جافا من خلال JavaBeans لتسهيل عملية التطوير مما خفف من تعقيد CORBA.



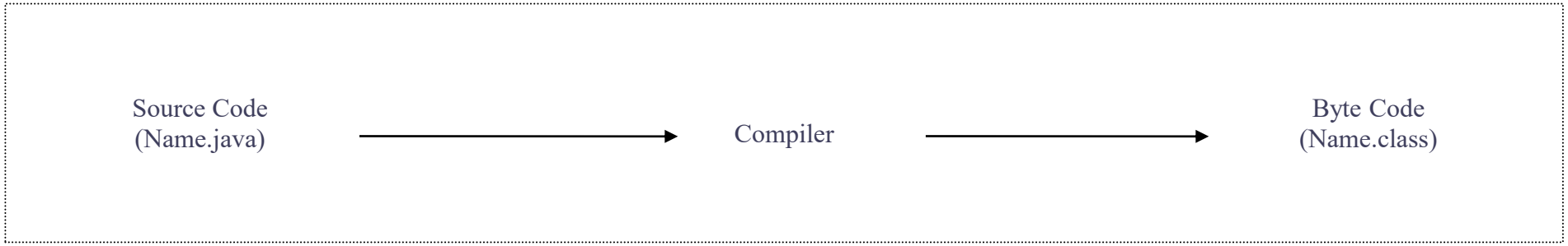
محدودية CORBA

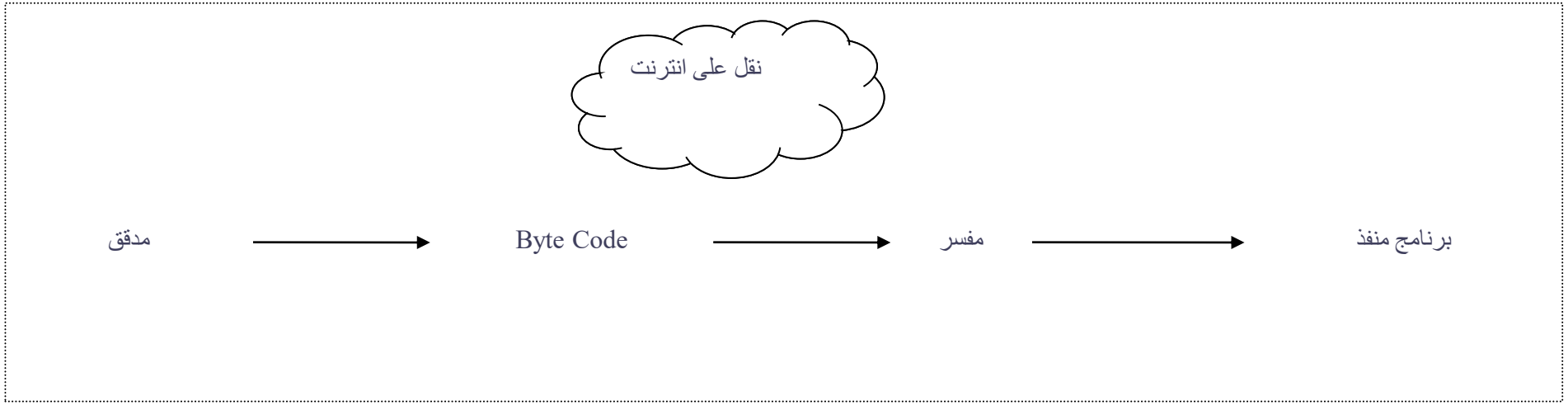
الاستقلال عن منصة العمل	نعم
دعم Broadcasting/ point to point	point to point
دعم Transactions	نعم
الوثوقية: أن تصل الرسالة مرة ومرة واحدة فقط	نعم
التصعدية أو التلاؤم مع تزايد الحركة Traffic	لا
دعم التخاطب المتزامن وغير المتزامن.	متزامن فقط
دعم بروتوكول تخاطب	نعم
دعم الاستدعاء بالاسم	لا
تنوع بنى الاتصال :امكانُة العمل عن طرُق wireless,sockets,...etc.	TCP/IP
لا تؤمن آليات لإدارة الـ Resources : load balancing، fail over كما أنها صعبة الاستعمال	

RMI & Java -

- نموذج جديد للـ Middlewareجدير بالدراسة صممت جافا لتكون قابلة للنقل على الشبكة ولهذا حققت خاصتين:
- حجم code صغير يمكن نقله على الشبكة دون ضغط (bytecode)
- قابل للتنفيذ على أية آلة Interpreter

RMI & Java -


Source Code
(Name.java)
Compiler
Byte Code
(Name.class)


نقل على انترنت
مدقق
Byte Code
مفسر
برنامج منفذ

RMI & Java -

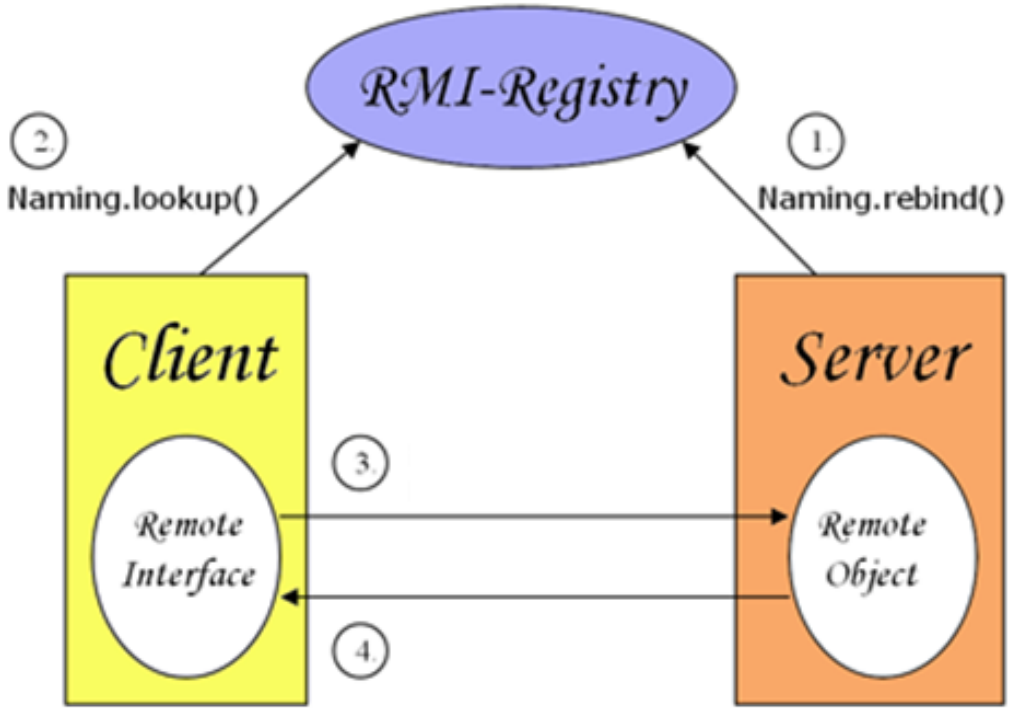
- يتكون البرنامج من code و Data
- توزيع البرنامج يعني وجود هذين المكونين على آلتين مختلفتين فإما أن ننقل المعطيات إلى آلة code أو ننقل الـ code إلى آلة المعطيات
- نقل المعطيات هي تكنولوجيا قديمة تعود إلى ODBC
- سمحت جافا بنقل الـ code وهذا ما يميزها عن بقية Middleware
- في حالة RPC يوزع البرنامج إلى إجرائيات
- في حالة Queue يوزع البرنامج إلى تطبيقات
- في حالة Object Middleware يوزع البرنامج إلى أغراض
- في جميع الحالات السابقة تُنقل المعطيات فقط في حين يمكن لغرض جافا بالتجوال على الانترنت وهو ما نراه عادة في حالة Applet :
  - يرسل الزبون request للـWeb server
  - يعيد الـ web server صفحة html تحتوي على applet

بنية JVM

- مدقق : يتحقق من أن البرنامج لا يحتوي على عمليات ممنوعة ولا يحاول الوصول إلى القرص الصلب
- مفسر
- المكتبات
- Native methods
- Garbage Collector
- Just In Time:JIT
- كما تستطيع الـ Applet عن طريق ارتباط URL شحن أي معطيات أو قيم على الإنترنت:
  newURL("www.sun.com/obj").getContent()

RMI

- آلية تسمح لغرضين جافا موجودين على JVMs مختلفتين بالتخاطب بطريقة استدعاء الطرائق.
- الفرق الأساسي بين CORBA و RMI هو إمكانية تمرير الأغراض كمعاملات في حين أن أغراض CORBAلا تتحرك وتتبادل أنماطاً بسيطة.
- تسمى آلية تمرير الأغراض هذه بـ Serialization.
- بنيان Java RMI
  - client: applet / application
  - server: Object
  - لكل مخدم Object عنوان مسجل ضمن بنية نسميها Registry تربط كل اسم بعنوار وذلك عبوراً بالطبقات التالية:
  - طبقة stub/Skeleton: الـ Stub هو صورة عن الغرض يمتلك نفس الواجهة وي -
  - طبقة العناوين البعيدة -
  - طبقة النقل -



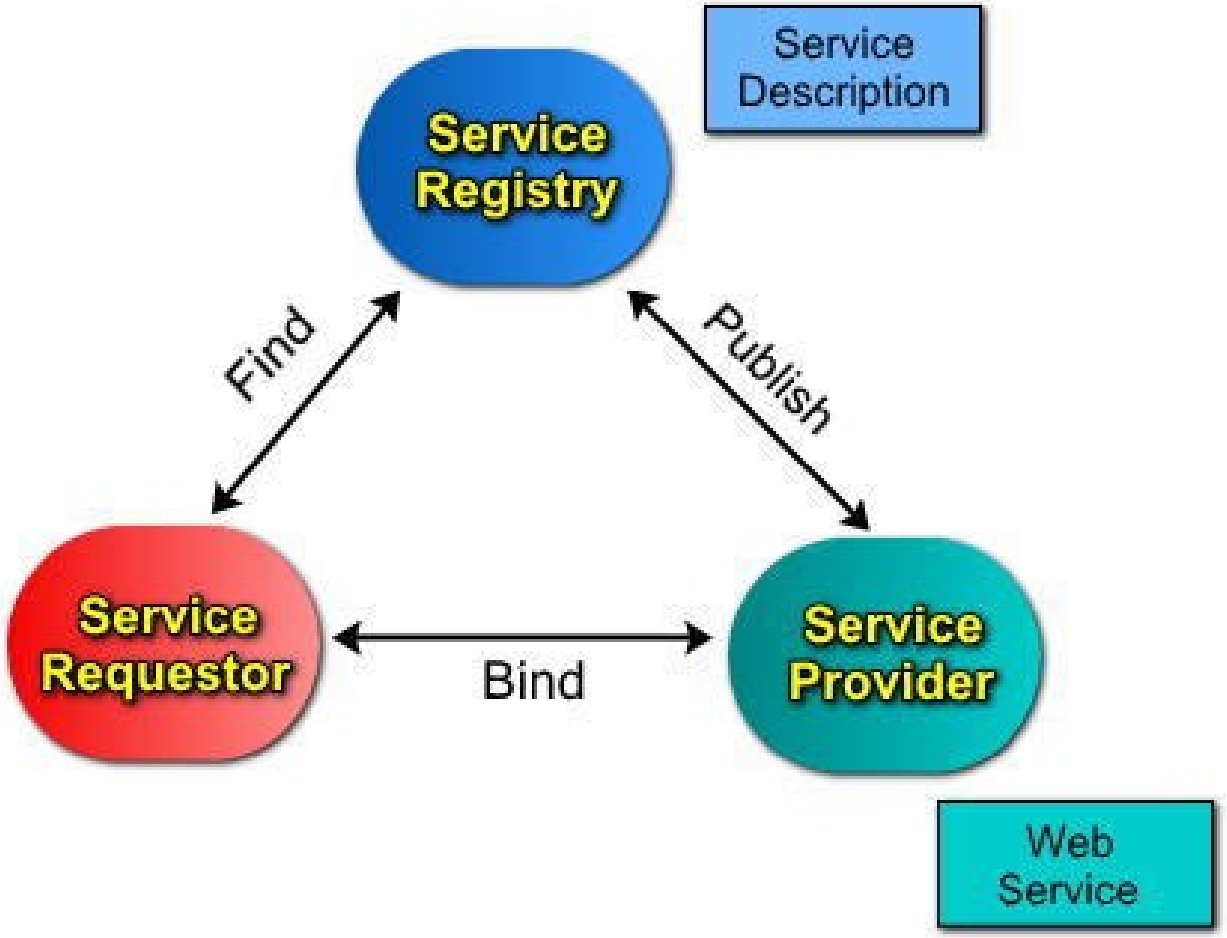
مراحل بناء تخاطب باستعمال RMI

- بناء واجهة الفرض المخدم: هو interface java يحتوي على ترويسات الطرائق ويُشتق من الواجهة Remote
- بناء الغرض المخدم الذي يحقق الواجهة السابقة
- نستعمل javac للحصول على bytecode و rmic للحصول على stub و skeleton
- بناء الغرض المخدم الذي يحقق الـinterface
- إنشاء نسخة من صف الغرض المخدم
- تسجيل هذه النسخة باسم ما تحت الـ Registery
- إنشاء وتوطين security manager لحماية المخدم
- تتشغيل مخدم الأسماء على آلة المخدم
- تشغيل المخدم الذي يقوم بتسجيل الغرض
- تشغيل التطبيق الزبون
- يستدعي الزبون الـ registry للحصول على مؤشر بعيد للغرض المراد من المخدم

RMI

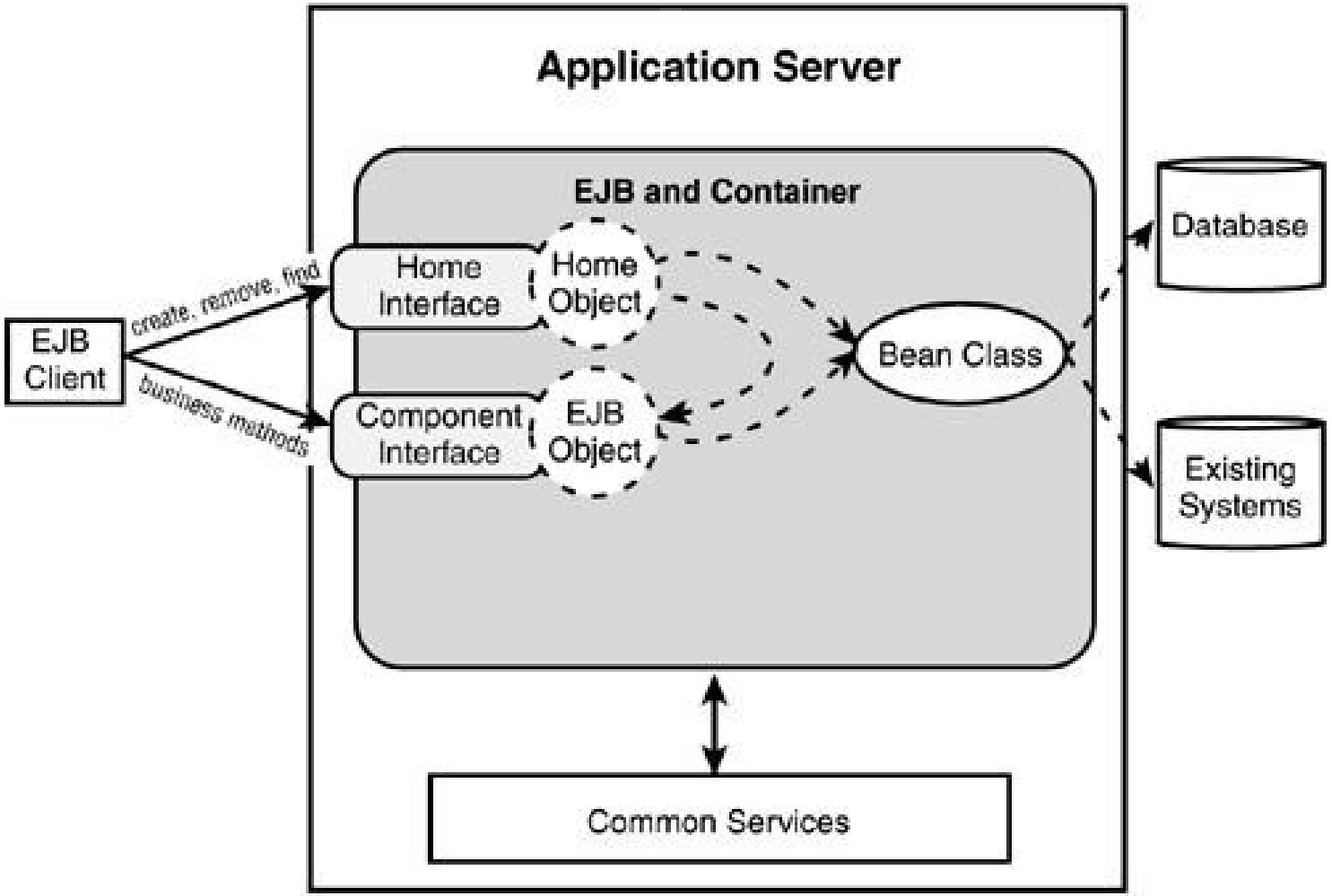
الاستقلال عن منصة العمل	نعم
دعم Broadcasting/ point to point	point to point
دعم Transactions	لا
الوثوقية: أن تصل الرسالة مرة ومرة واحدة فقط	نعم
التصعدية أو التلاؤم مع تزايد الحركة Traffic	لا
دعم التخاطب المتزامن وغير المتزامن.	متزامن فقط
دعم بروتوكول تخاطب	نعم
دعم الاستدعاء بالاسم	نعم
تنوع بنى الاتصال :امكانُة العمل عن طرُق etc,...wireless,sockets.	TCP/IP
مع أن RMI لا تدعم المناقلات إلا أنها انتشرت بسبب سهولة استعمالها	

Web Service



- XML over http

J2EE و EJB



EJB and Container

J2EE و EJB

الاستقلال عن منصة العمل	نعم
دعم Broadcasting/ point to point	نعم باستعمال MDB وSession Bean
دعم Transactions	نعم فهو يستعمل RMI over IIOP
الوثوقية: أن تصل الرسالة مرة ومرة واحدة فقط	نعم
التصعدية أو التلاؤم مع تزايد الحركة Traffic	نعم
دعم التخاطب المتزامن وغير المتزامن.	نعم
دعم بروتوكول تخاطب	نعم
دعم الاستدعاء بالاسم	نعم
تنوع بنى الاتصال :امكانُة العمل عن طرُق etc,...wireless,sockets.	نعم
IIOP هو البروتوكول الذي طورته CORBA والذي يدعم المناقلات حيث جمع EJB بين سهولة الـRMI ودعم CORBA للمناقلات	

J2EE وEJB

- للـEJB مميزات داعمة لهندسة البرمجيات حيث أنها وزعت العمل بين 6 أدوار
  - Bean Provider –
  - Bean Assembler –
  - Deployer –
  - Container Provider –
  - Application Server Provider –
  - Administrator –

ملخص

خصائص البرمجيات الوسيطة	Client/server	3 tiers Client/server	Queue	RPC	Corba	RMI	EJB/J2ee
الاستقلال عن منصة العمل	نعم	نعم	يتوفر على منصاتUnix,IBM/OS2, Windows NT	نعم	نعم	نعم	نعم
دعم point to point /broadcasting	Point to point only	Point to point only	نعم	لا	Point to point	Point to point	نعم يستعمل Session Beanو MDB
دعمTransaction	لا	لا	نعم	لا	نعم	لا	نعمRMI over يستعمل IIOP
الوثوقية	يعتمد على المبرمج	يعتمد على المبرمج	يضمن وصول الرسائل وذلك بالمحافظة على نسخة منها على القرص الصلب لا تهدم ما لم تعالج الرسالة	نعم	نعم	نعم	نعم
التصعدية أو التلائم مع تزايد الحركةTraffic	لا	نعم	نعم	لا	لا	لا	نعم
دعم التخاطب المتزامن وغير المتزامن	متزامن فقط	متزامن فقط	نعم	متزامن فقط	متزامن فقط	متزامن فقط	نعم
دعم بروتوكول التخاطب	لا	لا	لا	نعم	نعم	نعم	نعم
دعم الاستدعاء بالاسم	لا	لا	لا	لا	لا	نعم	نعم
تنوّع بنى الاتصال	socket	socket	نعم	لا	TCP/IP	TCP/IP	نعم

أمثلة وتطبيقات

- SAAS
- Tow phase commit
- Integration
  - DB level
  - Application level

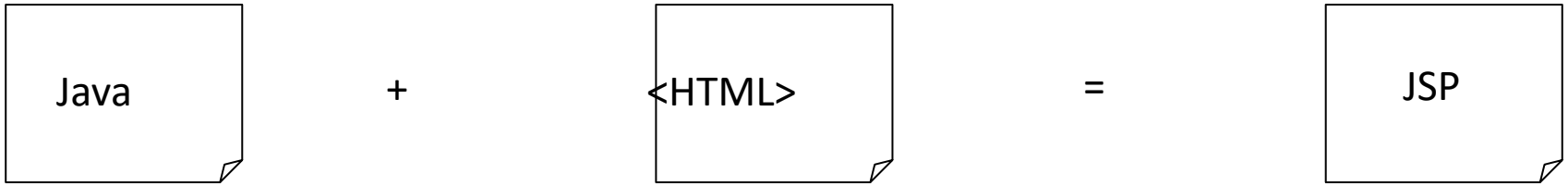
Introduction to the Java Server Pages (JSP) Technology

Presentation Overview:

- What are Java Server Pages?
- Structure of a JSP document.
  - Scriplet Tag
  - Expression Tag
  - Declaration Tag
  - Directive Tag
  - JSP Tags
- How JSPs Manage Sessions.
- Processing Request Parameters in JSPs.

What are Java Server Pages?

- The Java Server Pages technology combine Java code and HTML tags in the same document to produce a JSP file.



Why use JSP Technology?

- Convenient:
  - We already know Java and HTML!
  - Provides an extensive infrastructure for:
    - Tracking sessions.
    - Managing cookies.
    - Reading and sending HTML headers.
    - Parsing and decoding HTML form data.
- Efficient:
  - Every request for a JSP is handled by a simple Java thread. Hence, the time to execute a JSP document is not dominated by starting a process.

Why use JSP technology?

- Portable
  - JSP follow a well standardized API.
  - The Java VM which is used to execute a JSP file is supported on many architectures and operating systems.
- Inexpensive
  - There are a number of free or inexpensive Web Servers that are good for commercial-quality websites.
    - Helium uses Apache Tomcat.

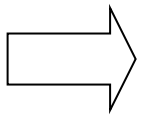
Structure of a JSP file.

- Similar to a HTML document.
- Four basic tags:
  - Scriplet
  - Expression
  - Declaration
  - Definition

Structure of a JSP document

- Scriptlet Tag
  - *<% code %>*
  - Imbeds Java code in the JSP document that will be executed each time the JSP page is processed.

```
<html>
  <body>
    <% for (int i = 0; i < 2; i++) { %>
      <p>Hello World!</p>
    <% } %>
  </body>
</html>
```

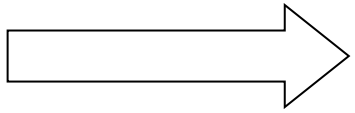


```
<html>
  <body>

      <p>Hello World!</p>
      <p>Hello World!</p>

  </body>
</html>
```

Structure of a JSP document

- Expression Tag
  - <%= *expression* %>
  - Imbeds a Java expression that will be evaluated every time the JSP is processed.
  - Note: no semi-colon ";" following expression.

```
<html>
  <body>
    <p><%= Integer.toString( 5 * 5 ) %></p>
  </body>
</html>
```



```
<html>
  <body>
    <p>25</p>
  </body>
</html>
```

Structure of a JSP document.

- Declaration Tag
  - <%! *declaration* %>
  - Imbeds Java declarations inside a JSP document.

```
<html>
  <body>
    <%! public boolean isPositive(int x)
            {
                if (x < 0) {
                    return false;
                }
                else {
                    return true;
                }
             } %>
  </body>
</html>
```

Structure of a JSP document.

- Directive Tag
  - <%@ *directive* %>
  - Directives are used to convey special processing information about the page to the JSP container.
  - Directive are defined in the web.xml file.

```
web.xml:
<taglib>
  <taglib-uri>/taglib/struts-logic</taglib-uri>
  <taglib-location>/WEB-INF/struts-logic.tld</taglib-location>
</taglib>
```

```
Example.jsp:
<%@ taglib uri="/taglib/struts-logic" prefix="logic" %>
```

Structure of a JSP document.

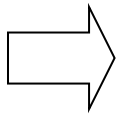
- JSP Tags.
  - <jsp:*property* />
  - *property* is defined in Tag Library Definition files.
  - JSP Tag reference: http://java.sun.com/products/jsp/syntax/1.2/syntaxref12.html

```
<html>
  <body>
    <jsp:useBean id="user" class="com.thirdbit.helium.model.User"/>
    <jsp:setProperty name="user" property="userName" value="Luigi" />
    <p>User Name: <jsp:getProperty name="user" property="userName" /></p>
  </body>
</html>
```

JSP Comments

- Regular Comment
  - <!-- *comment* -->
- Hidden Comment
  - <%-- *comment* --%>

```
Example.jsp:
<html>
  <!-- Regular Comment -->
  <%-- Hidden Comment --%>
</html>
```



```
<html>
  <!-- Regular Comment -->

</html>
```

Using JSPs to Manage Sessions

- Problem: HTTP is stateless. Each new request that the Web server receives starts with a blank slate.

- Solution: JSP provides an implicit object `session`.
  - Sessions store and retrieve session attributes.
  - Session attributes are remembered using either cookies or by passing request parameters.
  - The `session` object implements methods from the interface `javax.servlet.http.HttpSession`.

Managing Sessions

Method	Description
`getId()`	Returns the session ID.
`getCreationTime()`	Returns the time at which the session was created
`isNew()`	Returns `true` if user's browser has not yet confirmed the session id.
`invalidate()`	Discards the session, releasing any objects stored as attributes.
`getAttribute(String key)`	Returns an attribute that is stored in the session.
`setAttribute(String key, String value)`	Stores an attribute in the sessions.

Request Parameters

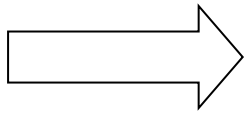
- JSP provides an implicit object `request` that stores attributes related to the request for the JSP page.

For example,

http://localhost/example.jsp?param1=hello&param2=world

example.jsp:

```
<html>
  <body>
    <p><%= request.getParameter("param1") %></p>
    <p><%= request.getParameter("param2") %></p>
  </body>
</html>
```



```
<html>
  <body>
    <p>Hello</p>
    <p>World</p>
  </body>
</html>
```

JSP Resources

- http://java.sun.com/products/jsp
- http://javaalmanac.com

- JSTL (JSP Standard Tag Library)
  - Provides “wrapper” tags for commonly used JSP elements.

- Jakarta Tag Libraries
  - Similar to JSTL. Defined in TLD (Tag Library Definition) files.