تطبيق UML

التحليل و التصميم بالمنحى للكائن باستخدام UML

ترجمة و إعداد

خالد الشقروني

2004-2006

تم تحميل هذا الكتاب من موقع كتب الحاسب العربية

www.cb4a.com

Computer Books for Arab

للمزيد من الكتب في جميع مجالات الحاسب والإلكترونيات ، تفضلوا بزيارتنا

يمكن اقتباس بعض الفقرات و لكن ليس كامل الكتاب.
عند الاقتباس يشار للمصدر و للمترجم كالتالي:

من كتاب تطبيق UML
ترجمة خالد عياد الشقروني
الرابط: www.nidam.net\sd\umlapl\index.html

لأغراض التضمين في كتاب لايجوز الاقتباس إلا بعد إذن مباشر من المترجم.

للحصول على آخر تحديث للكتاب أو للإطلاع على موضوعات ذات علاقة بتنشئة البرمجيات، راجع موقع الكتاب:

www.nidam.net\sd\

نظام للمعلومات
www.nidam.net

جدول المحتويات

مقدمة المترجم.....1

الفصل1: مقدمة إلى UML..... 3

ما هي UML ؟.....3
لغة مشتركة 3
ملخص 5

الفصل 2: UML داخل عملية التطوير..... 6

UML كترميز 6
النموذج الانحداري WATERFALL MODEL.....6
النموذج اللولبي.....8
إطار العمل التكراري التزايدي.....10
الاستهلال.....11
التفصيل 11
البناء.....12
التحوّل (الانتقال) TRANSITION.....13
كم عدد هذه التكرارات؟ و كم يجب أن تطول؟.....13
القيد الزمني TIME BOXING 14
التوقيتات النمطية للمشروع.....15
العملية الموّحدة من راشيونال 16
ملخص 17

الفصل 3: المنحى الكائني..... 18

البرمجة المهيكلة 18
أسلوب المنحى الكائني.....21
التغليف ENCAPSULATION 22
الكائنات OBJECTS 23
مصطلحات.....24

إستراتجية المنحى الكائني 25
ملخص 26

الفصل 4: نبذة عامة عن UML..... 27

مخطط واقعة استخدام THE USE CASE DIAGRAM 28
مخطط الصنفيات THE CLASS DIAGRAM 29
مخططات التعاون THE COLLABORATION DIAGRAMS 30
مخطط التتابع SEQUENCE DIAGRAM 31
مخططات الحالة STATE DIAGRAMS 32
مخططات التحزيم PACKAGE DIAGRAMS 33
مخططات المكونات COMPONENT DIAGRAMS 34
مخططات التجهيز DEPLOYMENT DIAGRAMS 35
ملخص 35

الفصل 5: طور الاستهلال..... 37

الفصل 6: طور التفصيل 39

المسودات PROTOTYPING 39
المخرجات DELIVERABLES 39
ملخص 41

الفصل 7: نمذجة وقائع الاستخدام 42

اللاعب ACTOR 42
الغرض من وقائع الاستخدام 44
مدى كثافة واقعة الاستخدام 45
توصيفات وقائع الاستخدام 48
وقائع الاستخدام في طور التفصيل 49
البحث عن وقائع الاستخدام 49
ورش عمل التخطيط المشترك للمتطلبات (JRP) 49
نصيحة حول العصف الذهني 51
ملخص 52

الفصل 8: نمذجة المفاهيم 53

إيجاد المفاهيم..... 55
استخلاص المفاهيم من المتطلبات 55
النموذج المفاهيمي في UML 56
إيجاد السمات..... 57
إرشادات لإيجاد السمات 58
الروابط ASSOCIATIONS 58
الإلزاميات المحتملة..... 60
بناء النموذج بالكامل 61
ملخص 62

الفصل 9: ترتيب وقائع الاستخدام 63

ملخص 64

الفصل 10: طور البناء 65

البناء..... 65
ملخص 66

الفصل 11: طور البناء: التحليل 67

عودة لوقائع الاستخدام 68
شروط مسبقة..... 68
شروط لاحقة 69
التدفق الرئيسي 69
التدفقات البديلة 70
التدفقات الاستثنائية..... 71
واقعة الاستخدام بعد اكتمالها..... 72
مخطط التتابع في UML 73
ملخص 75

الفصل 12: طور البناء: التصميم 76

التصميم – مقدمة 76
تعاون الكائنات في واقع الحياة 77
مخططات التعاون 79
صيغ التعاون: الأساسيات 79
مخططات التعاون: التوالي 81
مخططات التعاون: خلق كائنات جديدة 82
ترقيم الرسائل 83
مخططات التعاون: مثال عملي 83
بعض الإرشادات لمخططات التعاون 89
ملخص 90

الفصل 13: مخططات صنفيات تصميم 91

مديونية و دائنية الحسابات 91
خطوة 1: إضافة العمليات 92
خطوة 2: إضافة الاتجاهات 92
خطوة 3: تحسين السمات 93
خطوة 4: تحديد المنظورية 93
التجمع 94
التكون 94
إيجاد التجمع و التكون 95
ملخص 95

الفصل 14: أنماط توزيع المسؤولية 96

ما هو النمط؟ 96
(GRASP 1): الخبير EXPERT 97
(GRASP 2): المنشئ CREATOR 100
(GRASP 3): اتساق عال HIGH COHESION 101
(GRASP 4): اقتران منخفض LOW COUPLING 103
(GRASP 5): الموجه CONTROLLER 108
ملخص 110

الفصل 15: الوراثة 111

الوراثة – الأساسيات..... 111
الوراثة هي استخدام لصندوق أبيض 114
قاعدة 100%..... 115
الاحلالية..... 116
قاعدة هي_نوع_من..... 116
المشاكل عند الوراثة 119
منظورية السمات..... 120
التشكل POLYMORPHISM 121
الصنفيات المجردة 122
قوة التشكل 124
ملخص 125

الفصل 16: معمار النظام – الأنظمة الكبيرة و المركبة..... 126

مخطط التحزيم في UML..... 126
العناصر داخل الحزمة..... 127
لماذا التحزيم؟ 128
بعض الاستكشافات في التحزيم..... 128
الخبير EXPERT 129
الاتساق العالي HIGH COHESION 129
ضعف الاقتران LOOSE COUPLING..... 129
معالجة الاتصالات عبر الحزم..... 129
نمط الواجهة..... 131
المرتكز المعماري للتنشئة 132
مثال 133
معالجة وقائع الاستخدام الضخمة..... 134
طور البناء..... 135
ملخص 135

الفصل 17: نمذجة الحالات 136

مثال رسم حالة 137
صيغة مخطط الحالة 138
الحالات الفرعية..... 139
أحداث دخول/خروج..... 140

أحداث الإرسال 141
دفاعات 142
حالات التاريخ..... 142
استخدامات أخرى لمخططات الحالة 143
ملخص 143

الفصل 18: التحول للتوليف 144

تحديث (مزامنة) المشغولات SYNCHRONISING ARTIFACTS..... 144
ترجمة التصاميم إلى توليف 145
تحديد المناهج METHODS..... 148
خطوة 1..... 149
خطوة 2..... 149
خطوة 3..... 150
خطوة 4..... 150
ترجمة الحزم لتوليف 150
بلغة جافا..... 150
بلغة سي++ 151
نموذج المكونات في UML 151
ملخص 152

قائمة المراجع 153

مقدمة المترجم

أصبحت UML (لغة النمذجة الموحّدة) اللغة المعتمدة لترميز العمليات البرمجية لدى الوسط الصناعي. لقد خرجت من تحت عباءة ثلاثة يعدون من أهم أصحاب المنهجيات ولقت قبولا واسعا لدي المهتمين ببناء البرمجيات على اختلاف مشاربهم و منهجياتهم.

هي تقدم وسيلة رموزية مبسطة للتعبير عن مختلف نماذج العمل البرمجي يسهل بواسطتها على ذوي العلاقة – من محللين و مصممين و مبرمجين بل و حتى المستفيدين – التخاطب فيما بينهم و تمرير المعلومات في صيغة نمطية موحدة و موجزة، تغنيهم عن الوصف اللغوي المعتاد. إنها مثل مخططات البناء التي يتبادلها المساحون والمعماريون ومهندسو التشييد، أو مخططات الدوائر الكهربائية و الالكترونية التي يمكن لأي كان في هذا المجال أن يفهمها و يتعامل معها.

هنا يجب التنويه إلى نقطتين شكلتا سوء فهم ارتبط ب UML لدى الكثيرين:

UML ليست منهجية لبناء البرمجيات. بمعنى أنها لن ترشدك إلى أفضل الطرق لتصميم البرمجيات و تطويرها.

UML لا ترتبط بمنهجية محددة لتنشئة البرمجيات. بالرغم من أنها استلهمت رموزها من منهجيات سابقة و بالرغم من أنها صدرت من نفس الأفراد الذين صاغوا منهجية العملية الموحدة RUP لاحقا. يمكن توظيف عناصر لغة UML على مختلف العمليات البرمجية بغض النظر عن المنهجية المتبعة و بل بغض النظر عن وجود منهجية أصلا.

الدروس في هذا الكتاب لا تشرح فقط عناصر UML؛ و لكن أيضا تضعها ضمن سياقها داخل مختلف العمليات في منهجية واحدة لتطوير البرمجيات. غير هذا فإنها تتعرض بالشرح لكثير من المفاهيم السائدة في الوسط البرمجي مثل المنحى للكائن (Object Orientation) والتوريث وأنماط التصميم (Design Patterns), و المرتكز المعماري للتنشئة (Architecture-Centric Development) كلها ضمن مسار واحد داخل الدورة الحياتية للمشروع البرمجي.

أيضا أود أن أشير إلى أن هذه الدروس تبنّت منهجية العملية الموحّدة من راشيونال (RUP) ، إحدى أهم المنهجيات "الثقيلة". و بالرغم من تفضيلي شخصيا للمنهجيات الخفيفة الأقلّ وزنا* ، إلا أني أجدها فرصة للكثيرين للتعرّف عن قرب على كيفيّة تنشئة البرامج ضمن منهجية معينة.

* www.shagrouni.com/sd/agilemanpre.html

الفصل1: مقدمة إلى UML

ما هي UML ؟

لغة النمذجة الموحّدة (Unified Modelling Language)، أو (UML) ، هي لغة نمذجة رسومية تقدم لنا صيغة لوصف العناصر الرئيسية للنظم البرمجية. (هذه العناصر تسمّى artifacts مشغولات في UML). في هذه الفصول سوف نستكشف النواحي الرئيسية في UML، و نصف كيف يمكن تطبيق UML في مشروعات تطوير البرمجيات.

تتّجه UML بطبيعتها نحو بناء البرمجيات كائنية المنحى object oriented ، لذلك سوف نستكشف بعض أهم مبادئ المنحى الكائني.

في هذا الفصل القصير، سوف نلقي نظرة على أصول UML، و سنناقش الاحتياج إلى لغة مشتركة في صناعة البرمجيات. بعدها نرى كيف يتم تطبيق UML على مشروع برمجي.

لغة مشتركة

الصناعات الأخرى لديها لغات و رموز خاصة بها، و يفهمها كل من له علاقة في حقل اختصاص معين.

$$\int_0^{\infty} 1/x^2 dx$$

شكل 1 ــ معادلة رياضية للتكامل

بالرغم من أن الصورة أعلاه هي رسم بسيط جدا ، فإن الرياضيين في كل العالم يدركون و من أول وهلة بأنها تمثل معادلة تكامل . و بالرغم من بساطة الرمز، إلا أنه يشير إلى موضوع بالغ العمق و التعقيد. الرمز بسيط، و لكن بالمقابل، كل الرياضيين في العالم يمكنهم و بكل وضوح تبادل الآراء فيما بينهم باستخدامه مع مجموعة بسيطة

أخرى من الرموز. الرياضيون هنا لديهم *لغة مشتركة*. كذلك الموسيقيون، و مهندسو الالكترونيات، و الكثير من الفروع والمهن الأخرى.

لمدة، كان مهندسو البرمجيات يفتقرون لمثل هذه الرموز. بين عامي 1989 و 1994، و هي الفترة التي يشار إليها بـ "حروب المناهج"، كان يوجد ما يزيد عن 50 لغة نمذجة برمجية قيد الاستعمال – كل منها تملك رموزها الخاصة! كل لغة تحتوي على قواعد تميزها، بينما في نفس الوقت، كل لغة لديها عناصر تتشابه مع تلك التي في اللغات الأخرى.

و لمزيد من الفوضى، لا توجد لغة متكاملة، بحيث نادرا ما يجد القائمون على البرمجيات ما يرضي كامل حاجتهم في لغة واحدة!

في منتصف التسعينيات، برزت ثلاث منهجيات لكي تكون الأقوى. بدأت هذه المنهجيات الثلاث في التقارب، كل واحدة منها تحوي على عناصر من الأخريين. كل منهجية تملك نقاط قوة خاصة بها:

- **بوك Booch** كانت ممتازة فيما يخص التصميم و التنفيذ. لقد عمل "قرادي بوك" Grady Booch بكثافة على لغة آدا Ada، و كان له دور رئيسي في تطوير تقنيات المنحى الكائني (object oriented) للغة. وبالرغم من قوة منهجية بوك إلا أن الرموز فيها لم تأخذ القبول الحسن (الكثير من الأشكال السحابية تغزو نماذجه – ليست بالجميلة!)
- **OMT** (تقنية النمذجة الكائنية Object Modelling Technique) كانت الأفضل في التحليل و في أنظمة المعلومات ذات البيانات الكثيفة.
- **OOSE** (Object Oriented Software Engineering هندسة البرمجيات كائنية المنحى) و تتميز بنموذج يسمى وقائع الاستخدام (**Use Cases**). تعد وقائع الاستخدام أسلوب قوي من أجل فهم سلوك كامل النظام (و هو المجال الذي كان فيه المنحى الكائني ضعيفا).

في عام 1994، قام جيم رامبخ Jim Rumbaugh، مؤسس OMT، بمفاجأة عالم البرمجيات حين ترك العمل بشركة جنرال الكتريك General Electric و انضمّ الى قرادي بوك للعمل في شركة راشيونال (Rational Corp). الغرض من المشاركة كانت

من أجل دمج أفكارهما و صبّها في منهجية موحدة (و كان بالطبع عنوان العمل لهذه المنهجية هي "المنهجية الموحدة" Unified Method).مع عام 1995، انضم أيضا مبدع OOSE ايفار جاكوبسون Ivar Jacobson، إلى راشيونال، و تم ضم أفكاره (خاصة مفهوم "وقائع الاستخدام" Use Cases) في المنهجية الموحدة – الآن تدعى لغة النمذجة الموحدة (Unified Modelling Language).* وعُرف الفريق الذي يتكون من رامبخ و بوك و جاكوبسون بـ "الأصدقاء الثلاثة" (Three Amigos).

بغض النظر عن بعض الحروب و المشاحنات البسيطة، بدأت المنهجية الجديدة تجد استحبابا لدى أوساط صناعة البرمجيات، فتم تكوين لجنة مشتركة consortium خاصة بـ UML، شاركت فيها عدد من المؤسسات ثقيلة الوزن مثل هيولت–باكارد (Hewlett-Packard) و ميكروسوفت (Microsoft) و أوراكل (Oracle).

كما تم تبنّي UML من قبل منظمة (OMG) ** في 1979، و من حينها امتلكت (OMG) اللغة و دأبت على صيانتها. لذلك عمليا أصبحت لغة UML عامة وليست ملكية خاصة.

ملخص

UML هي لغة رسومية للتعبير عن مشغولات (artifacts) التطوير البرمجي.

تقدم لنا اللغة رموزا ننتج بها النماذج.

تلقى UML تبنيا واسعا في الوسط الصناعي كلغة موحدة.

اللغة غنيّة جدا، و تحمل في طيّاتها العديد من جوانب أفضل الممارسات في هندسة البرمجيات.

---

* التسمية الرسمية هي modeling، وما ورد هو التهجئة الانكليزية للكلمة.

** OMG: مجموعة الإدارة الكائنية (Object Management Group)، و هي جهة غير ربحية لوضع المواصفات. انظر www.omg.org لمزيد من التفاصيل.

الفصل 2: UML داخل عملية التطوير

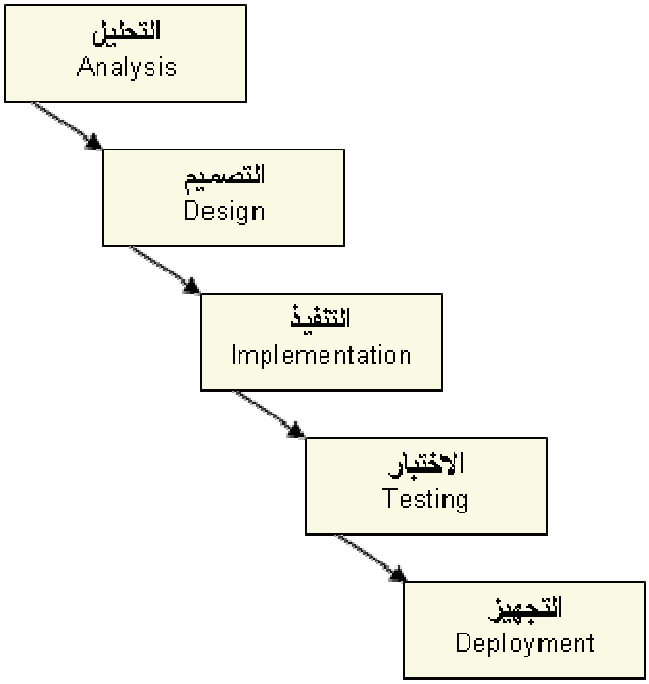
UML كترميز

عند تطويرهم لUML ، اتخذ الأصدقاء الثلاثة قرارا واضحا بعدم تضمين اللغة أية قضايا تتعلق بالعمليات (process). ذلك لأن العمليات تثير الكثير من الجدل – فما يسري على شركة ما قد يشكل كارثة بالنسبة لشركة أخرى. فشركة مختصة بمجالات الدفاع تتطلب عمليات توثيق و جودة و اختبارات أعقد بكثير من شركة مختصة بالتجارة الإلكترونية. لذلك فإن لغة UML عمومية، لغة عامة تسمح بالتقاط المفاهيم الأساسية لتطوير البرمجيات و وضعها على "ورقة".

بعبارة أخرى، لغة UML هي ببساطة لغة أو أداة ترميز أو قواعد نحوية ، سمّها ما شئت. المهم، أنها لن ترشدك إلى **كيف** يتم تطوير البرمجيات.

لكي نتعلم كيفية استخدام UML بكفاءة، سوف نتعامل مع عملية process مبسّطة خلال الدروس القادمة، و نحاول فهم كيف تساعدنا UML في كل مرحلة. و لكن كبداية، لنلقى نظرة أولا على بعض العمليات البرمجية الشائعة.

النموذج الانحداري Waterfall Model

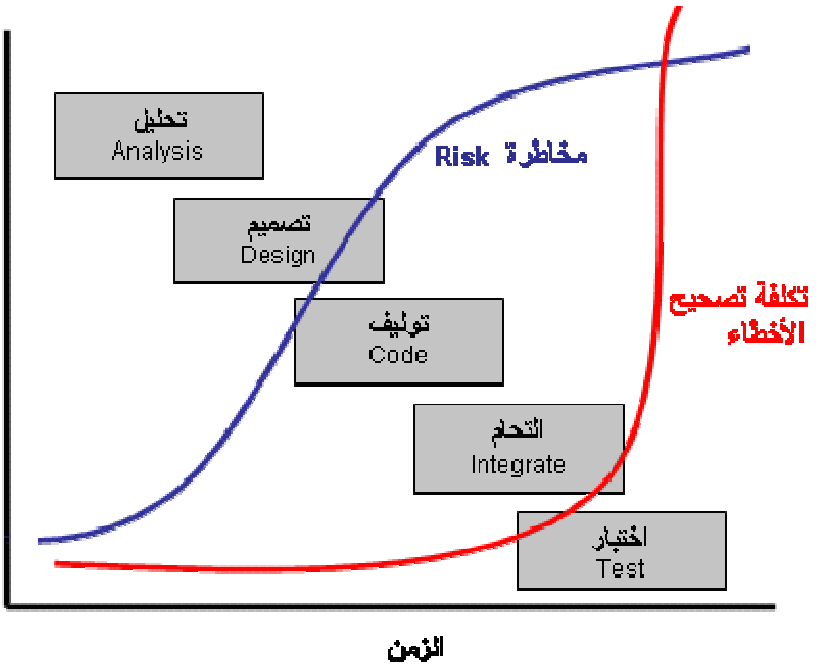


شكل 2: النموذج "الانحداري" التقليدي.

بحسب النموذج الانحداري –التدفقي– كل مرحلة يجب إنهائها قبل الشروع في المرحلة التي تليها.

هذه العملية المبسطة (و التي يسهل إدارتها) تبدأ بالتداعي بمجرد أن يزداد تعقيد و حجم المشروع. أهم مشاكلها هي:

- أنه حتى في الأنظمة الضخمة يجب أن تكون مفهومة وأن يكون قد سبق تحليلها بالكامل قبل مباشرة مرحلة التصميم. فيزداد بذلك التعقيد و يصبح عبئا على المطوّرين.
- المخاطر (Risks) تتجمع لاحقا. المشاكل عادة ما تظهر في المراحل الأخيرة من العملية – خاصة خلال التحام النظام. و للأسف؛ تزداد تكلفة تصحيح الأخطاء بصورة مضاعفة مع مرور الزمن.
- في المشاريع الكبيرة، كل مرحلة تستغرق فترات طويلة مبالغ فيها، إن مرحلة اختبار بطول سنتين ليست بالتأكيد وصفة جيدة للاحتفاظ بفريق العمل!



شكل 3: في النموذج الانحداري، تتزايد المخاطر وتكلفة تصحيح الأخطاء مع مرور الزمن.

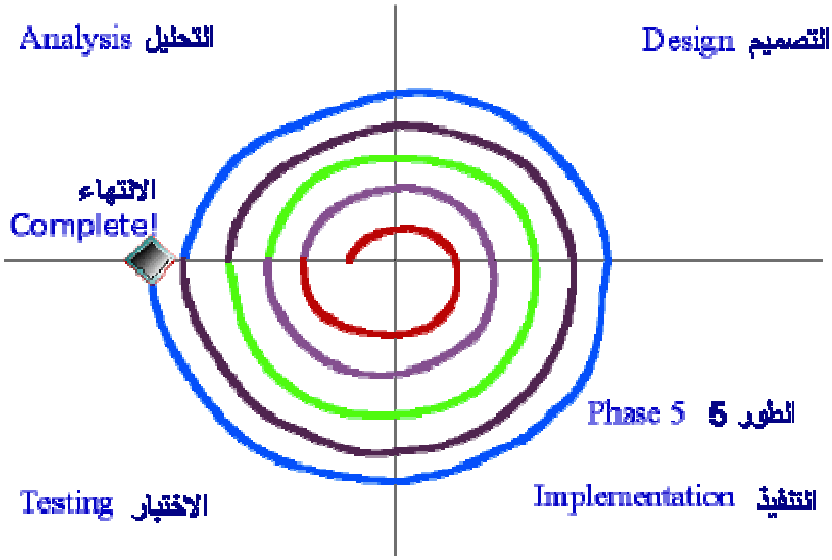
أيضا، حيث أن مرحلة التحليل تنجز في مخاض قصير مع تدشين المشروع، سنواجه مخاطرة جدية تتمثل في القصور في فهم متطلبات الزبون. حتى لو التزمنا بإجراءات صارمة لإدارة المتطلبات و قمنا بصياغتها تعاقديا مع الزبون. هناك دائما احتمال بأنه مع استكمال التوليف (coding) و الدمج (integration) و الاختبار لن يكون المنتج النهائي بالضرورة هو ما يتوقعه الزبون.

برغم كل ما ذكرنا، لا يوجد عيب في النموذج الانحداري، بشرط أن يكون المشروع صغيرا بما يكفي. صحيح أن تعريف "صغير بما يكفي" قابل للنقاش، و لكنه أساسي، فإذا أمكن لمشروع أن يتصدى له فريق صغير من الأفراد، كل فرد على دراية بجميع جوانب المشروع، و إذا كانت فترة المشروع قصيرة (بضعة أشهر)، عندها يكون النموذج الانحداري عملية لها قيمتها. فهو أفضل بكثير من الخبط العشوائي!

بإيجاز، النموذج الانحداري سهل الفهم و بسيط في إدارته. لكن مميزات هذا النموذج تبدأ في التداعي بمجرد أن يزداد تعقيد المشروع.

النموذج اللولبي

الأسلوب البديل هو النموذج اللولبي (spiral model) ، حيث نقوم بالتصدّي للمشروع عن طريق تقسيمه إلى سلسلة من الدورات الحياتية lifecycles القصيرة ، كل دورة تنتهي بإصدار لبرنامج قابل للتنفيذ.



شكل 4: العملية اللولبية، هنا يتم تقسيم المشروع إلى خمسة أطوار، كل طور يُبنى فوق سابقه، و كل طور ينتهي بإنتاج إصدارة لبرنامج جاهز للتشغيل.

مع هذا الأسلوب:

- يستطيع فريق العمل أن يشتغل على كامل الدورة الحياتية (تحليل، تصميم، توليف، اختبار) بدلا من صرف سنوات على نشاط واحد.
- يمكننا الحصول على ملاحظات وتقييم الزبون مبكرا و بصورة منتظمة، ورصد الصعوبات المحتملة قبل التمادي بعيدا في عمليات التطوير.
- يمكننا التصدّي لنقاط المخاطرة مقدما، بالأخص التكرارات ذات المجازفة العالية (مثلا: التكرار الذي يتطلب تنفيذ بعض التقنيات الجديدة غير المجرّبة) يمكن تطويرها أولا.
- يمكن اكتشاف مدى حجم و تعقيد العمل مبكرا.
- الإصدار المنتظم للبرنامج يعزّز من الثقة.
- الوضع الحالي للمشروع (مثل: مقدار ما تم انجازه) يمكن تحديده بدقة أكبر.

عيوب العملية اللولبية هي:

- عادة ما تقترن هذه العملية بما يعرف بالتنشئة السريعة للتطبيقات (RapidApplication Development)، و التي تعتبر من قبل كثيرين مجرد عمل هواة (hacker's charter).
- العملية أكثر صعوبة عند إدارتها. ففي النموذج الانحداري يمكن الاستعانة بالتقنيات التقليدية لإدارة المشروعات مثل مخططات غانط (Gantt Charts)، لكن العمليات اللولبية تتطلب أساليب مختلفة.

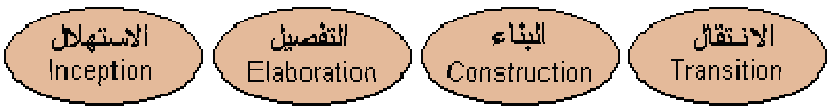
من أجل تدارك عيوب العملية اللولبية، لنلقي نظرة على أسلوب مشابه لكن أكثر تقنينا و يدعى **إطار العمل التكراري التزايدي** (Iterative, Incremental Framework).

في ورقة لفيليب كروشتن قام بتعداد الفخاخ التي غالبا ما يقع فيها المدراء عند تطوير أول تكرار (المرجع*[5]* ، متوفر في موقع راشيونال.

إطار العمل التكراري التزايدي

إطار العمل التكراري التزايدي (Itreative, Incremental Framework) هو امتداد منطقي للنموذج اللولبي، لكنه أكثر تقنينا و صرامة. سوف نقوم بتبنّي إطار العمل التكراري التزايدي خلال بقية هذه الدروس.

ينقسم إطار العمل إلى أربعة أطوار رئيسية: (Inception)، التفصيل (Elaboration)، البناء (Construction) والانتقال (Transition). يتم انجاز هذه الأطوار على التوالي، لكن يجب أن لا نخلط بين هذه الأطوار و المراحل في الدورة الحياتية للنموذج الانحداري. في هذا القسم سوف نشرح هذه الأطوار و نستعرض النشاطات التي يتم أداؤها في كل طور.



شكل 5: الأطوار الأربعة لإطار العمل التكراري التزايدي

الاستهلال

يتعلق طور الاستهلال بوضع نطاق المشروع و تحديد التصوّر العام له. بالنسبة للمشاريع الصغيرة يمكن لهذا الطور أن يكون مجرد دردشة بسيطة على فنجان قهوة، يعقبها اتفاق على البدء في المشروع. في المشاريع الكبيرة يتطلب الأمر المزيد من التحرّي. المخرجات (deliverables) المحتملة من هذا الطور هي:

- وثيقة التصوّر.
- استكشاف مبدئي لاحتياجات الزبون.
- التحديد الابتدائي لمفردات (glossary) المشروع (المزيد حول هذا لاحقا).
- دراسة جدوى (تتضمن محددات النجاح، التنبؤات المالية، تقديرات العائد على الاستثمار، الخ).
- التحديد المبدئي لنقاط المخاطرة.
- خطة المشروع.

سوف نناقش طور الاستهلال بشيء من التفصيل عندما نتعرض لحالة دراسية case study في الفصل الرابع.

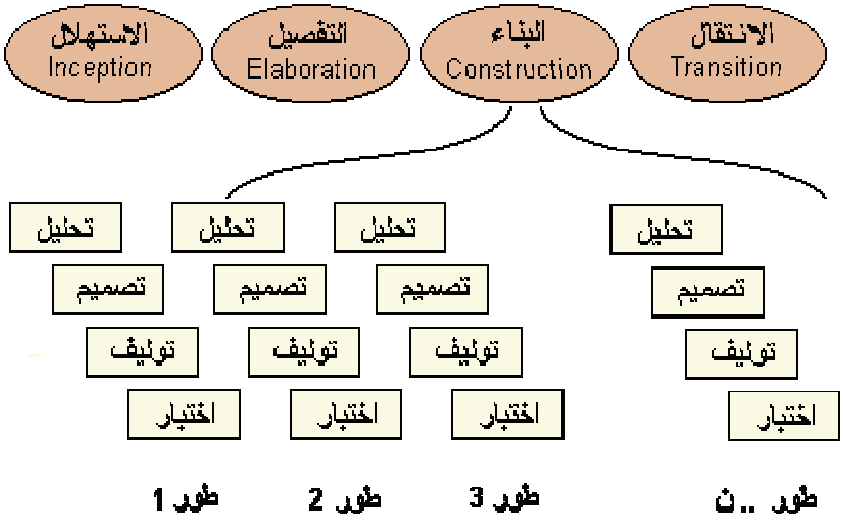
التفصيل

الغرض من التفصيل هو تحليل المشكلة، و المضي خطوة ابعد في إعداد خطة المشروع، و استبعاد المناطق الأكثر مخاطرة فيه. مع نهاية طور التفصيل؛ نتأمل حصولنا على فهم عام لكامل المشروع، و لكن ليس بالضرورة فهما متعمقا (لاحقا سيتم التصدي له بصورة أجزاء صغيرة يسهل مناولتها).

نموذجان من UML يكون لهما قيمة كبيرة في هذه المرحلة. نموذج حالة الاستخدام Case Use سيساعدنا في متطلبات المستفيد (أي الزبون)، و مخطط الصنفية Class Diagram و الذي يمكن أيضا استخدامه لاستكشاف المفاهيم العامة التي يتصورها المستفيد. المزيد حول هذا الموضوع قريبا.

البناء

في طور البناء، نقوم ببناء المنتج. هذا الطور لن يتحقق بأسلوب خطي Linear ؛ بل يتم بناؤه بنفس أسلوب النموذج اللولبي، من خلال سلسلة من التكرارات. كل تكرار هو نفسه الأسلوب القديم: نموذج انحداري 3 بسيط. و من خلال الحرص على أن يكون كل تكرار أقصر ما يمكن، نأمل أن نتجنب المشاكل المزعجة التي ترافق الانحداريات.



شكل 6: طور البناء و يحتوي على سلسلة من الانحدارات المصغرة.

مع نهاية أكبر عدد من التكرارات سوف نطمح للحصول على منظومة تعمل (مبدئيا بالطبع، ستكون منظومة محدودة جدا في المراحل المبكرة). هذه التكرارات تسمّى تزايدات Increments، ومن هنا أتت تسمية إطار العمل هذا!

[3] لاحظ أنه في أطوار الاستهلال و التفصيل، يمكن بناء مجسمات (برمجية) أولية. هذه المجسمات يمكن تطويرها تماما بنفس الطريقة ؛ أي سلسلة من التكرارات الانحدارية المصغرة. عموما و بقصد التوضيح في هذه الدروس سوف نبقي على أطوار الاستهلال و التفصيل بسيطة قدر الإمكان، و نستعمل الانحداريات عند البناء فقط.

التحوّل (الانتقال) Transition

الطور النهائي يتعلق بنقل المنتج النهائي إلى الزبائن. النشاطات المعتادة في هذا الطور تتضمن:

- الإصدارات المبدئية لأغراض الاختيار من قبل بيئة المستخدم.
- الاختبارات في الموقع، أو تشغيل المنتج بالتوازي مع النظام القديم الذي سيستبدل.
- تجهيز البيانات (مثل تحويل قواعد البيانات و صبّها في قوالبها الجديدة، توريد البيانات ، الخ..)
- تدريب المستخدمين الجدد.
- التسويق والتوزيع و المبيعات.

يجب أن لا نخلط بين طور التحوّل هذا و طور الاختبار التقليدي في نهاية النموذج الانحداري، فمنذ بداية التحوّل يجب أن يكون لدينا منتجا قابلا للتشغيل بعد أن تم اختباره بالكامل، و أن يكون جاهزا و متوفرا للمستخدمين. و كما أوضحنا سابقا، بعض المشاريع قد تتطلب خطوة اختبار مبدئية ، و لكن المنتج يجب أن يكون مكتملا قدر الإمكان قبل الدخول في هذا الطور.

كم عدد هذه التكرارات؟ و كم يجب أن تطول؟

التكرار الواحد يمتد عادة من أسبوعين إلى شهرين، أية زيادة على شهرين سوف تؤدي إلى زيادة في التعقيد و الوصول إلى النقطة التي لامناص منها: "الانفجار الكبير" ، دوّامة ضم الأجزاء إلى بعضها البعض، حيث العديد من المكونات البرمجية ستحتاج إلى أن تنتظم و تلتحم لأول مرة.

إذا كبر المشروع و زاد تعقيده فلا يعني هذا أن تكون التكرارات أطول – لأن هذا سوف يزيد من مستوى التعقيد الذي سيكون على المطوّرين التعامل معه في المرّة الواحدة. بدلا من ذلك يجب أن يخصص للمشروع الأكبر تكرارات أكثر.

فيما يلي بعض العوامل التي يجب أن توثر في طول مدّة التكرار:

- دورات التطوير المبكرة قد تحتاج لأن تكون أطول. هذا يعطي المطوّرين فرصة لأداء أعمال استكشافية على تقنيات جديدة أو غير مختبرة ، أو لتحديد البنية التحتية للمشروع.
- الأفراد حديثو الخبرة.
- فرق العمل المتعددة والتي تعمل على التوازي.
- فرق العمل الموزعة (في أكثر من موقع).

أيضا ، نريد أن نضم إلى هذه القائمة المشروع ذو *المراسمية العالية* (الشكليات التعاقدية) الذي سيحتاج عادة إلى تكرارات أطول. و نقصد بالمشروع ذو المراسمية العالية ذلك الذي يتطلّب إصدار و توفير الكثير من الوثائق الخاصة بالمشروع إلى الزبون، أو ربما مشروع يجب أن يلبّي العديد من المتطلبات القانونية. مثال جيد على هذا المشاريع ذات العلاقة بالأمور العسكرية، ففي هذه الحالة فان أعمال التوثيق سوف تزيد من طول فترة التكرار – و لكن كمية التطوير البرمجي الذي يتم التصدي لها في هذا التكرار يجب أن تكون في حدودها الدنيا لتجنّب عدّونا الرئيسي و هو عبء التعقيد.

القيد الزمني Time Boxing

الأسلوب الأمثل لإدارة عملية تكرارية تزايدية هو فرض *قيد زمني (صندقة زمنية)*. بهذا الأسلوب الحازم يتم تحديد فترة زمنية ثابتة يجب خلالها إتمام تكرارية معينة.

إذا لم تكتمل التكرارية مع نهاية القيد الزمني، يتم إنهاء التكرارية على أية حال. النشاط الأهم في التقييد الزمني هو المراجعة في نهاية التكرارية. يجب أن تبحث المراجعة في أسباب التأخير، و أن تعيد جدولة الأعمال غير المنتهية و تضمينها في التكرارات القادمة.

إحدى النصائح لكيفية تطبيق القيد الزمني، أن يكون المطورون هم المسؤولين (أو على الأقل من لهم الكلمة العليا) عن تحديد ما هي المتطلبات التي يتم تغطيتها في كل تكرار ، باعتبارهم هم الذين سوف يلتزمون بهذه الآجال.

يعد الالتزام بالقيد الزمني أمرا صعبا، فهو يتطلّب حسّا عاليا بالانضباطية خلال كامل المشروع. من المغري جدا التخلّي عن المراجعة و تخطّي القيد الزمني إذا حان أجل التكرار و كانت نسبة اكتماله "99%". حالما يرضخ المشروع لمثل هذا الإغراء و يتم

تجاهل مراجعة واحدة ، فإن المفهوم بكامله يبدأ في التداعي. إذا تم تجاهل عدة مراجعات، فان التخطيط للتكرارات القادمة سوف تكون مائعة و تبدأ الفوضى في أخذ مكانها.

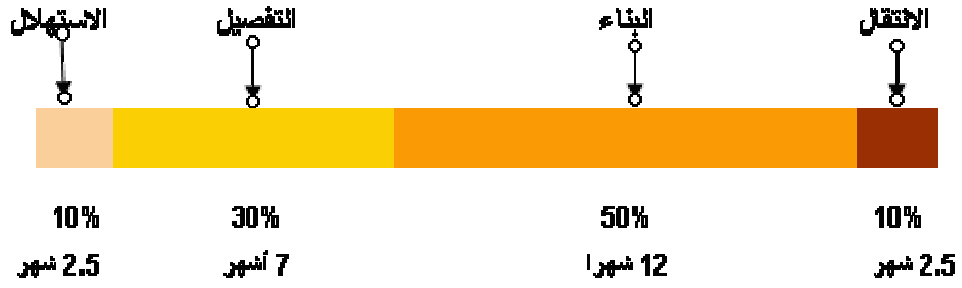
يفترض بعض المديرون أن التقييد الزمني يعيق مرونة الإرجاء، هذا غير صحيح، فإذا لم تكتمل التكرارية وقت انتهاء أجل القيد الزمني فإن العمل غير المكتمل يجب نقله إلى التكرارات التالية، و يتم إعادة جدولة خطط التكرارات – يمكن أن يتضمن هذا إرجاء تاريخ التسليم أو إضافة تكرارات أكثر. عموما للتقييد الزمني فوائد هي:

- تفرض الهيكلية الصارمة عملية التخطيط و معاودتها. فلا يتم التخلّي عن الخطط إذا بدأ المشروع في التمطيط.
- إذا تم فرض التقييد الزمني، تتضاءل فرص أن يغص المشروع في الفوضى إذا ما ظهرت مشاكل، حيث أن هناك دائما مراجعة رسمية منتظمة تلوح في الأفق.
- إذا ما تسرّب الخوف و بدأ المطوّرون في التخبّط بصورة عشوائية، سيتم وقف هذا التخبّط حالما تتم المراجعة.

بصورة أساسية، يسمح القيد الزمني للمشروع بكامله أن يتريث مرة بعد الأخرى ليحزم أمره من جديد. انه لا يعرقل إمكانيات الإرجاء، و يحتاج إلى إدارة مشروعات قوية كي يعمل بصورة صحيحة.

التوقيتات النمطية للمشروع.

كم يجب أن يستغرق كل طور من الأطوار الأربعة؟ يتباين ذلك من مشروع لآخر، و لكن كمؤشّر عام 10% للاستهلال، 30% للتفصيل، 50% للبناء و 10% للانتقال.



شكل7: التوقيتات المحتملة لكل طور. هذا المثال يوضح طول كل طور لمشروع يستغرق سنتين.

العملية الموّحدة من راشيونال

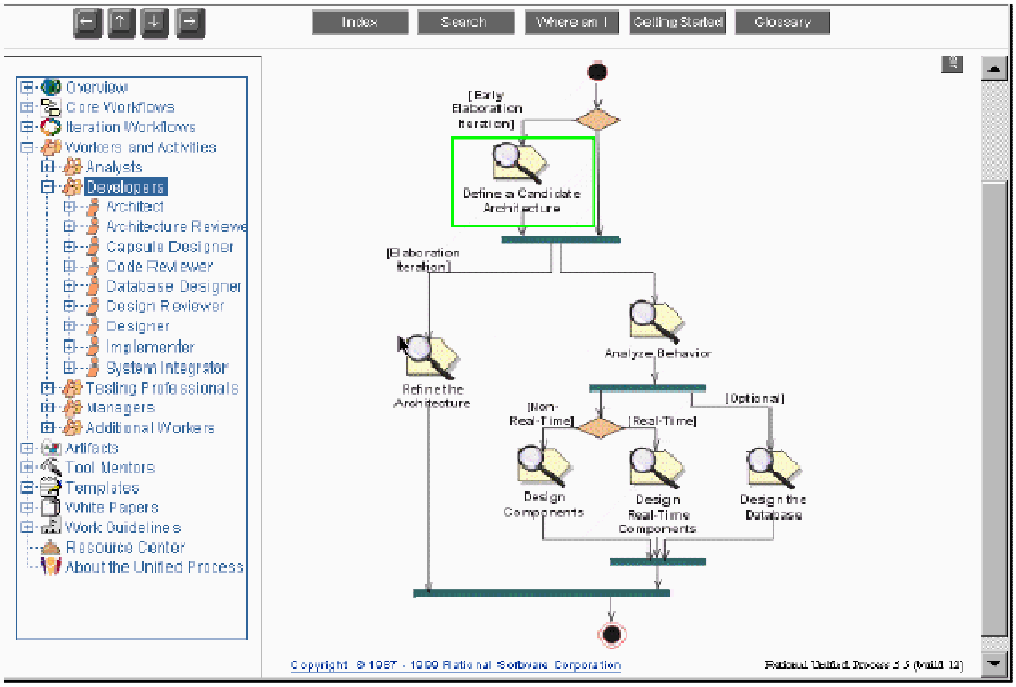
العملية الموحّدة من أي بي ام The Rational Unified Process (the RUP) هي إحدى أكثر الأمثلة شهرة للدورة الحياتية التكرارية قيد الاستعمال حاليا. تم تطوير هذه المنهجية من قبل نفس "الأصدقاء الثلاثة" الذين قاموا بتطوير UML ، و لذلك فان RUP متكاملة جدا مع UML.

بصورة أساسية، يقدّر القائمون على هذه المنهجية بأن كل مشروع يختلف عن الآخر، و ذو احتياجات مختلفة، مثلا، بعض المشاريع لا تتطلب إلا طورا قصيرا للاستهلال، بينما المشاريع ذات العلاقة بأمور عسكرية فإن طور الاستهلال فيها قد يمتد لسنوات.

حتى هذه النقطة، تعد RUP مرنة و تسمح بإعادة تكييف كل طور في العملية. أيضا تحدد RUP و بكل عناية قواعد لكل فرد في المشروع و بحسب الاحتياج.

و قد أصدرت وقتها شركة راشيونال منتوجا لمساعدة المشاريع التي تتبنى RUP ، يمكن إيجاد المزيد من التفاصيل في www.rational.com . [*] المنتوج بالأساس عبارة عن دليل على شبكة الانترنت أو برنامجا يضم جميع ملامح RUP. و تقدم الشركة إمكانية استعماله على سبيل التجربة. (تم بيعها لاحقا لشركة IBM في فبراير 2003)

[*] http://www-306.ibm.com/software/awdtools/rup/index.html



شكل 8: لقطة من RUP (مؤسسة IBM).

تفصيل مزايا و عيوب RUP خارج نطاق درسنا هذا، و لكن بصفة عامة فان أســاس RUP هي الدورة الحياتية التكرارية التزايدية و التي سيتم تبنّيها خلال دروسنا هذه.

ملخص

يقدم إطار العمل التكراري التزايدي العديد من الفوائد مقارنة بالعمليات التقليدية.

ينقسم إطار العمل هذا إلى أربعة أطوار الاستهلال، التفصيل، البناء، الانتقال.

يعني التطوير التصاعدي استهداف الحصول على توليف قابل للتشغيل في نهايــة كــل تكرار (بأكبر عدد ممكن منها).

التكرارات يمكن تقييدها زمنيا كأسلوب صارم لجدولة و مراجعة التكرارات.

بقية هذه الدروس سوف تركز على إطار العمل Framework ، و كيف تقــوم UML بدعم مخرجات كل طور في إطار العمل.

الفصل 3: المنحى الكائني

في هذا الفصل سوف نلقي نظرة على مفهوم المنحى الكـــائني[4] Orientation Object (OO) . لقد تمّ تصميم لغة النمذجة الموحّدة UML بحيث تدعم المنحى الكـــائني ، لـــذلك سنقوم بتعريف مفاهيمه قبل التعمق في لغة UML ، و استكشاف المزايا التي قد يوفرّها.

البرمجة المهيكلة

أولا، لنختبر بصورة سريعة كيف يتم تصميم الأنظمة البرمجية باستخدام الاتجاه المهيكل (أحيانا يُسمّى وظائفي Functional).

في البرمجة المهيكلة Structured Programming، الطريقة العامة المتّبعة هي النظر إلى المسألة، ثم تصميم مجموعة من الوظائف functions التي يمكنهـــا انجـــاز المهـــام المطلوبة لحلها. إذا تضخّمت هذه الوظائف ، يتم تجزئتها حتى تصير صغيرة بالحدّ الذي يتيسّر فيه مناولتها و فهمها. هـــذه العمليـــة تـــدعى التفكيـــك الوظـــائفي functional decomposition.

ستحتاج معظم الوظائف إلى بيانات من نوع ما لتعمل عليها. البيانـــات فـــي الأنظمـــة الوظائفية عادة ما يحتفظ بها في قاعدة بيانات من نوع ما (أو قد يحتفظ بها في الـــذاكرة كمتغيّرات عامة global variables).

لنأخذ مثالا بسيطا، تخيّل منظومة لإدارة معهد، هذه المنظومة تحتفظ ببيانات الطلبـــة و المدرّبين في المعهد إضافة للمعلومات حول الدورات المتوفرة، كذلك تقوم المنظومة بتتبع كل طالب و المقررات التي التحق بها.

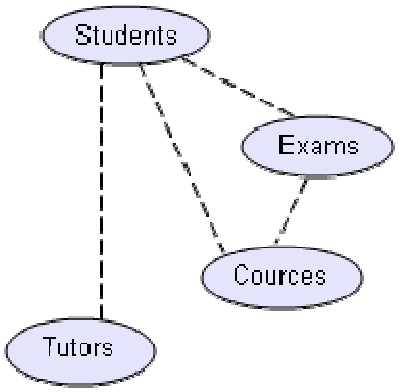
التصميم الوظائفي المحتمل سيتضمّن كتابة الوظائف functions التالية:

---

[4] سوف نستعمل جملة المنحى الكائني للتعبير عن المنحى الكائني للتـــصميم أو/ و المنحـــى الكـــائني للبرمجة.

```
add_student [5] إضافة طالب
enter_for_exam دخول امتحان
check_exam_marks فحص علامات امتحان
issue_certificate إصدار شهادة
expel_student طرد طالب
```

سوف نحتاج أيضا إلى نموذج بيانات data model ليمثّل هذه الوظائف. نحتاج لتخزين معلومات عن الطلبة، و المدربين و الامتحانات و المقررات، لذا يجب علينا تصميم مخطط قاعدة بيانات database schema للاحتفاظ بهذه البيانات.[6]



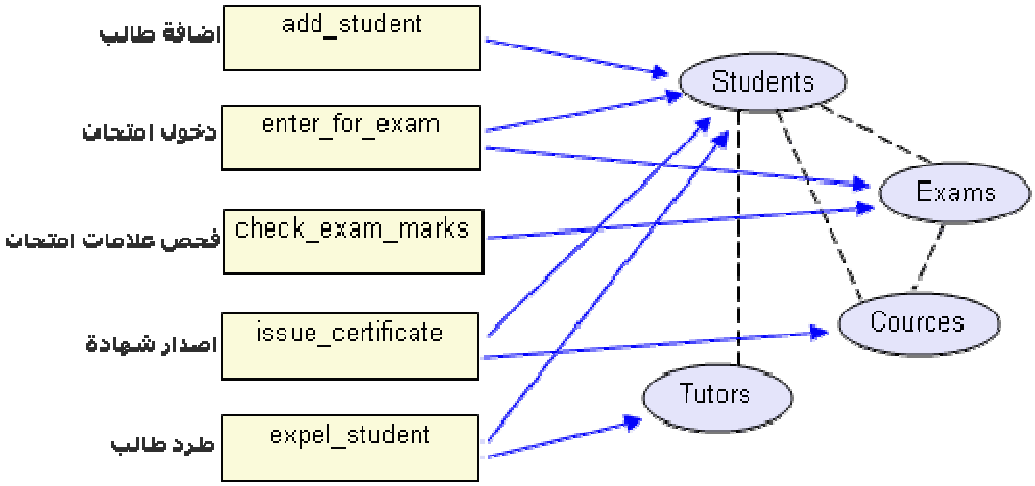
شكل 9: مخطط قاعدة بيانات بسيط. الخطوط المنقطّة تشير إلى اعتمادية مصفوفة بيانات على أخرى، مثلا، كل طالب يتم تدريبه بواسطة عدة مدرّبين.

الآن بدأ واضحا أن الوظائف functions التي حدّدناها سابقا سوف تعتمد على هذه المصفوفة من البيانات. مثلا، وظيفة "add_student" (إضافة طالب) ستقوم بتغيير محتويات "Students" (طلبة)، و وظيفة "issue_certificate" (إصدار شهادة) تحتاج إلى الوصول إلى بيانات طالب "Student" (لمعرفة تفاصيل الطالب الذي يحتاج للشهادة) و ستحتاج الوظيفة أيضا إلي بيانات الامتحانات "Exam" .

[5] نستخدم الشرطة السفلية "_" للدلالة على أن هذه الوظائف مكتوبة داخل توليف code.

[6] لاحظ أنه خلال هذه الفصل، سوف لن نستعمل صيغة الترميز notation الرسمية للتعبير عن المفاهيم.

المخطط diagram التالي عبارة عن رسم لكل الوظائف، مجتمعة رفق البيانات، و رسمت الخطوط فيها حيثما وجدت اعتمادية dependency .



شكل 10: خريطة الوظائف، و البيانات و الاعتماديات.

المشكلة مع هذه المقاربة؛ أن المسألة التي نتعامل معها إذا ما تعقدت أكثر فإن صعوبة المحافظة على المنظومة و صيانتها ستزداد. فإذا أخذنا المثال أعلاه، ماذا سيحدث لو تغيّرت المتطلبات requirements بطريقة تؤدّي إلى تغيير أسلوب مناولة بيانات الطالب Student.

كمثال، لنتخيّل أن منظومتنا تعمل على أكمل ما يكون، لكننا اكتشفنا أن تخزين تاريخ ميلاد الطالب على شكل عدد ذو خانتين كي يمثل السنة كانت فكرة سيّئة، الحل المبدئي هنا هو أن نقوم بتغيير حقل تاريخ الميلاد في جدول الطلبة Students من خانتين إلى أربع خانات لرقم السنة.

المشكلة الجدية لهذا التغيير تنبع من أنه قد يسبّب في ظهور آثار جانبية غير متوقعة. فبيانات جدول الامتحانات Exam و جدول المقررات Courses و جدول المدرّبين Tutors تعتمد كلها (بطريقة أو بأخرى) على بيانات جدول الطالب Students، لذا قد نتسبب في كسر بعض العمليات بتغييرنا البسيط هذا، و إعاقة وظائف add_student و enter_for_exams و issue_certificate و expel_student ، فوظيفة

add_student لن تعمل بالتأكيد لأنها تتوقع أن تكون المعلومة الخاصة بسنة الميلاد على شكل رقم بخانتين بدلا من أربع.

إذا، لدينا معدل كبير من المشاكل المحتملة، و الأسوأ أننا لن نستطيع بسهولة تعيين أماكن الاعتمادية في التوليف code التي ستتأثر بهذا التغيير.

كم من مرّة قمت بتعديل سطر في التوليف و بكل براءة و دون أن تعي أنك سبّبت عن غير قصد في كسر عمليات أخرى قد تبدو لا علاقة لها في الظاهر؟

إشكالية عام 2000 (ثغرة الألفية) ذات التكلفة العالية كان سببها بالضبط هذه المشكلة. فحتى لو أن حلها يفترض فيه أن يكون بسيطا (تغيير كل حقل سنة من خانتين إلى أربع) فإن التداعيات المحتملة لهذا التغيير البسيط يجب التحقق منها و فحصها بدقة.[7]

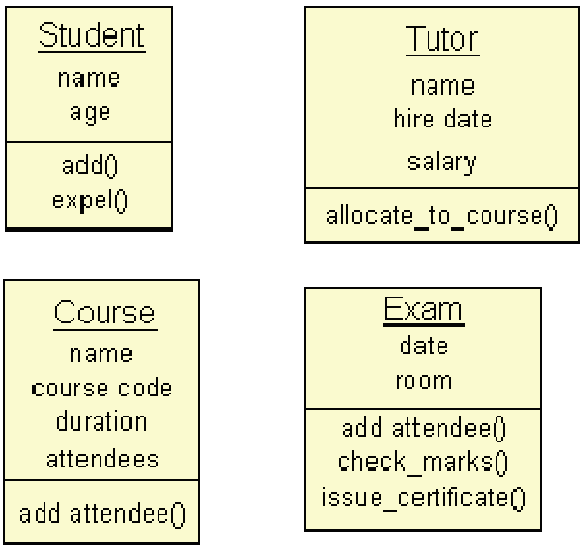
أسلوب المنحى الكائني

يحاول المنحى الكائني "OO" التقليل من تأثير هذه المشكلة عن طريق الجمع بين البيانات data و الوظائف functions ذات العلاقة في قالب module واحد.

بالنظر إلى الشكل 10 أعلاه، يبدو واضحا وجود علاقة بين البيانات و الوظائف؛ فمثلا وظيفتا: add_student و expel_student مرتبطتان بقوة ببيانات Student الطالب.

الشكل التالي يبيّن التجميعات الكلّية للبيانات و الوظائف ذات العلاقة على شكل قوالب:

[7] هذا لا يعني أن المنظومات بلغة كوبول و التي هي ليست ذات منحى كائني كلها منظومات عقيمة. ما نود قوله هو انه لا يوجد أي عيب في البرمجة المهيكلة النمطية، و لكن و جهة النظر هنا في هذا الفصل؛ أن المنحى الكائني يوّفر سبيلا لبناء برمجيات أكثر متانة كلما كبرت و تشابكت.



شكل 11: البيانات و الوظائف المرتبطة موضوعة في قوالب.

بعض النقاط الجديرة بالملاحظة حول نظام القولبة الجديد في البرمجة:

- أكثر من تمثل لنفس القالب يمكن استحضاره عند تشغيل البرنامج. في منظومة المعهد، سيكون هناك تمثل لقالب Student لكل طالب يتبع المعهد يجرى التعامل معه في المنظومة. و كل تمثل سيكون له بياناته الخاصة. (بالطبع لكل تمثل اسما مختلفا).
- القوالب يمكنها "التخاطب" مع قوالب أخرى عن طريق استدعاء وظائفها. مثلا، عندما يتم استدعاء وظيفية "add" (إضافة) في Student (الطالب) ، سيتم خلق تمثلا جديدا لقالب Student ، و بعدها سيتم استدعاء وظيفة "add_attendee" (إضافة مشترك) من التمثل المناسب لقالب "Course" دورة.

التغليف Encapsulation

الأمر الأساسي هنا، أنه لا يتم السماح بقراءة أو تغيير أي عنصر في البيانات إلا من قبل التمثّل الذي تتبعه هذه البيانات . فمثلا، التمثل الخاص بقالب المدرّب Tutor لا يمكنه تحديث أو قراءة بيانات "age" العمر داخل قالب Student الطالب. هذا المفهوم يسمّى بالتغليف Encapsulation ، الذي يسهم في جعل هيكل المنظومة أكثر متانة، و يجنّبها

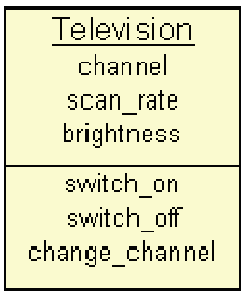
الحالة التي تعرّضنا لها سابقا، حيث التغيير البسيط في جزء من البيانات قد يؤدّي إلى تغييرات متتابعة و أكثر عمقا.

بواسطة هذا التغليف، يمكن للمبرمج الذي يتعامل مع قالب Student مثلا أن يقوم بتغيير بيانات القالب بكل أمان و دون الخشية من وجود قوالب أخرى مرتبطة أو تعتمد على هذه البيانات. قد يحتاج المبرمج إلى تحديث الاجرائيات داخل القالب، و لكن التأثير سيبقى محصورا داخل قالب واحد معزول عن القوالب الأخرى.

الكائنات Objects

خلال هذا الفصل، أشرنا إلى هذه التجميعات من البيانات و الوظائف المرتبطة بأنها قوالب "modules". عموما إذا نظرنا إلى خصائص هذه القوالب سنجد مرادفات لها في العالم الحقيقي.**الكائنات** Objects في عالم الواقع يمكن تمييزها بشيئين : كل كائن في عالم الواقع لديه **بيانات** data و **سلوك** behaviour . فمثلا جهاز التلفاز هو كائن و يعالج البيانات بطريقة تجعلها تنضبط في قناة محددة، يتم تحديد معدّل المسح إلى قيمة معيّنة، كذلك معدّل التباين و شدّة الإضاءة و هكذا. التلفاز أيضا يمكنه أن "يقوم" بأشياء، التلفاز يمكنه التشغيل أو الإقفال، القنوات يمكن تغييرها، و هكذا.

يمكننا تمثيل هذه المعلومات بنفس أسلوب القوالب البرمجية السابقة:



شكل 12: بيانات و سلوك جهاز التلفاز

على المنوال نفسه إذا، فإن "كائنات" العالم الحقيقي بالإمكان قولبتها بطريقة مشابهة للقوالب البرمجية التي سبق مناقشتها.

لهذا السبب، نسمّي هذه القوالب بالكائنات Objects و منها جاء مصطلح البرمجة / التصميم بالمنحى الكائني Design/Programming Object Oriented.

حيث أن نظمنا البرمجية تقدّم حلولا لمشاكل حقيقية في واقعنا (سواء كان ذلك نظام تسجيل في معهد، أو نظام إدارة مخازن، أو نظام توجيه صواريخ)، يمكننا تحديد الكائنات في العالم الواقعي و بسهولة نقوم بتحويلها إلى كائنات برمجية.

بكلمات أخرى، *يقدم المنحى الكائني تجريدا أو تمثيلا أفضل للعالم الحقيقي أو الواقع*. نظريا، هذا يعني أنه إذا تغيّر الواقع (كتغيّر المتطلبات) فان تغيير الحلول أو تعديلها سيكون أسهل، ما دامت الخطوط الرابطة بين مشاكل الواقع و الحلول الموضوعة لها واضحة.

مصطلحات

البيانات data الخاصة بالكائن object تسمّى عادة بخصائص أو سمات Attributes الكائن. و تسمّى التصرّفات المختلفة التي يقوم بها الكائن طرق أو نهجيات Methods الكائن. النهجيات هي مرادف لما يعرف في لغات البرمجة بالوظائف functions أو الإجرائيات procedures .

المصطلح الآخر المشهور في هذا السياق هو Class الصنفية. الصنفية أو الصنف هي ببساطة أرضية (template) يقوم عليها الكائن. يتم في الصنفية وصف السمات attributes و النهجيات methods التي ستكون حاضرة لكل تمثلات أو تجسّدات الصنفية. في منظومة المعهد التي تكلّمنا عنها في هذا الفصل، كان لدينا صنفية تسمّى Student.

سمات attributes صنفية الطالب Student Class كانت الاسم و العمر و ما إلى ذلك، أما النهجيات methods فكانت add و expel. في التوليف code سنحتاج لتحديد هذه الصنفية لمرة واحدة فقط. و حالما يتم تشغيل التوليف ، يمكننا خلق create تجسّدات/تمثلات instances لهذه الصنفية أو بعبارة أخرى: إنشاء كائنات objects الصنفية.

كل كائن منها سيمثل طالبا محددا، و كل منها أيضا سيحوي ما يخصّه من قيم البيانات.

إستراتجية المنحى الكائني

بالرغم من أن هذا الفصل قد لمس باختصار فوائد المنحى للكائن (مثل: منظومات أكثر ثباتا، تمثيل أفضل للواقع)، إلا أننا تركنا بعض الأسئلة بدون إجابة. كيف نميّز الكائنات التي نحتاجها عند تصميمنا لمنظومة ما؟ ما هي النهجيات methods و السمات attributes المفترض وجودها؟ ما هو الحجم المناسب للصنفية؟ و غيرها من الأسئلة. ستنتقل بنا هذه الدروس خلال عمليات تنشئة البرمجيات باستخدام المنحى للكائن (و UML) ، و سنقوم بالإجابة عن كل هذه الأسئلة.

أحد أهم نقاط ضعف المنحى الكائني في الماضي هو أنها في الوقت الذي تتميّز فيه بأنها قوية على مستوى الصنفية/الكائن، إلا أنها ضعيفة عند التعبير عن سلوك المنظومة ككلّ. النظر من خلال الصنفيات شيء جيّد، لكن الصنفيات في حد ذاتها هي مجموعات لكينونات على مستوى منخفض و لا يمكن لها أن تصف ما تقوم به المنظومة **ككل**. باستخدام الصنفيات فقط فإن الأمر يشبه محاولة فهم كيفية عمل الحاسوب من خلال فحص مكونات اللوحة الأم!

الاتجاه الحديث و المدعوم بقوة من قبل UML هو **نسيان** كلّ ما يتعلّق بالكائنات و الصنفيات في المراحل المبكّرة للمشروع، و التركيز بدل ذلك على ما يجب أن تكون المنظومة قادرة على القيام به. بعد ذلك، و مع تقدّم العمل في المشروع يتم **تدريجيا** بناء الصنفيات لتجسيد النواحي الوظيفية للمنظومة المطلوبة. في هذه الدروس سنتتّبع هذه الخطوات بدءاً من التحاليل الأوّلية و حتى تصميم الصنفية.

ملخص

- المنحى الكائني طريقة تفكير تختلف عن الاتجاه المهيكل.
- نقوم بالجمع بين البيانات و التصرفات ذات العلاقة داخل صنفيات.
- ثم يقوم برنامجنا بخلق تجسّدات/ تمثلات للصّنفية، بشكل كائنات.
- الكائنات يمكنها التعاون مع بعضها البعض، من خلال مخاطبة نهجياتها.
- البيانات في الكائن مغلّفة و لا يقوم بتعديلها إلا الكائن نفسه.

الفصل 4: نبذة عامة عن UML

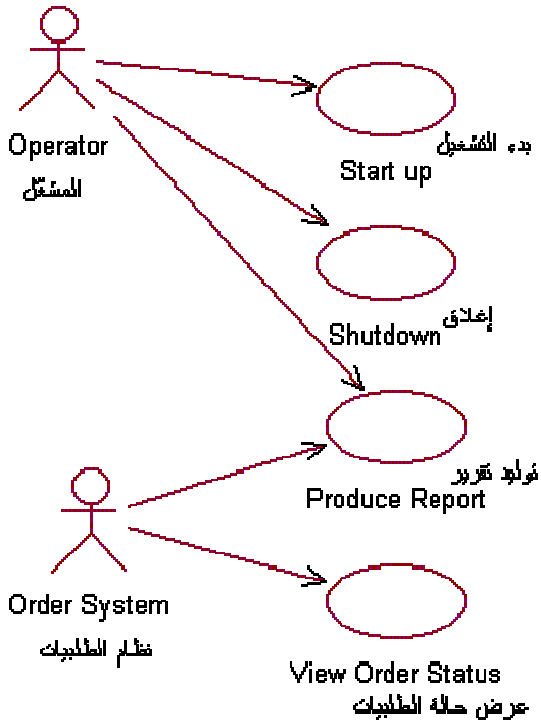
قبل أن نخوض في نظرية UML ، سنقوم بجولة مختصرة جدا للإطلاع على بعض المفاهيم الأساسية في UML.

أوّل ما يتم ملاحظته عن UML هو أنه يوجد العديد من المخطّطات المختلفة (نماذج) و التي يجب التعوّد عليها. السبب في هذا التنوّع يعود إلى أن المنظومة يُحتمل أن يُنظر إليها من زوايا مختلفة بحسب المشاركين فيها. تطوير البرمجيات يشترك فيه عدد من الأفراد، و كل واحد له دور – مثلا:

- المحلّلون
- المصمّمون
- المبرمجون
- القائمون بالاختبار
- مراقبو الجودة
- المستفيدون
- الكتّاب التقنيون

كلّ هؤلاء الأفراد يهتمون بجوانب مختلفة من المنظومة، و كلّ واحد منهم يحتاج إلى مستوى مختلف من التفاصيل. على سبيل المثال، المبرمج يحتاج إلى أن يفهم التصميم الموضوع للمنظومة من أجل تحويله إلى تعليمات برمجية في مستواها الأدنى. بالمقابل الكاتب التقني (الموثّق) ينصبّ اهتمامه على سلوك المنظومة ككلّ، فيحتاج لفهم كيف يعمل المنتوج. تحاول UML أن تقدّم لغة قويّة التعبير بحيث يمكن للمشاركين الاستفادة و لو من مخطط واحد على الأقل من مخططات UML. فيما يلي نظرة سريعة لبعض أهم هذه المخططات. بالطبع، سوف نتعرّض لها بمزيد من التفاصيل مع تقدّم هذه الدروس:

مخطط واقعة استخدام The Use Case Diagram



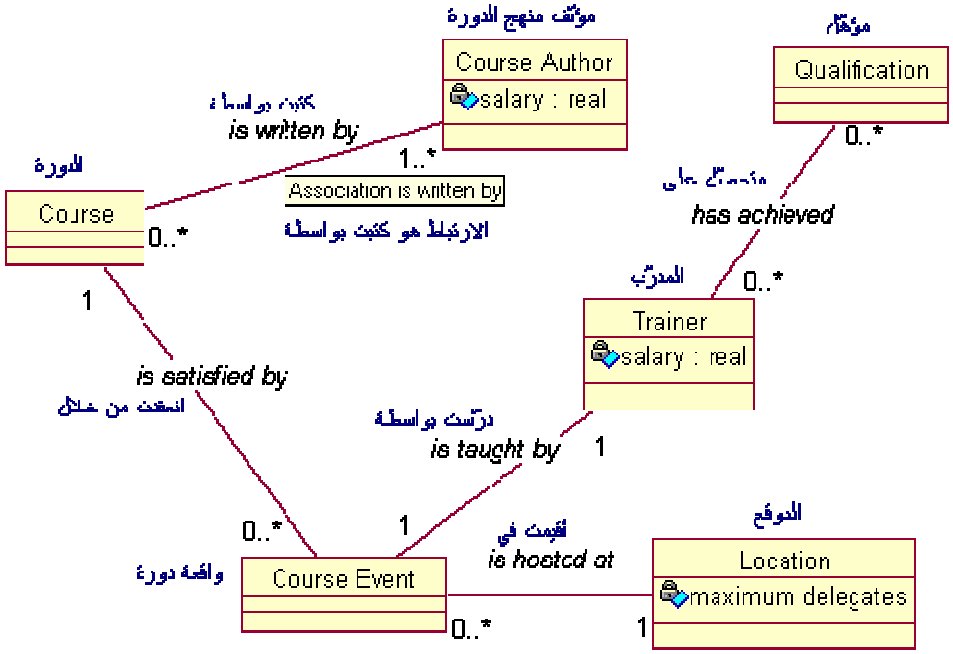
شكل 13: مخطّط واقعة استخدام

واقعة الاستخدام Use Case هي وصف لسلوك النظام من وجهة نظر المستخدم. فهي ذات فائدة خلال مراحل التحليل و التطوير، و تساعد في فهم المتطلبات.

يكون المخطط سهلا للاستيعاب. مما يمكّن كلا من المطوّرين (محلّلون، مصمّمون، مبرمجون، مختبرون) و المستفيدين (الزبون) من الاشتغال عليه.

لكن هذه السهولة يجب أن لا تجعلنا نقلل من شأن مخططات واقعة الاستخدام. فهي بإمكانها أن تحتمل كامل عمليات التطوير ، بدءا من الاستهلال و حتى التسليم.

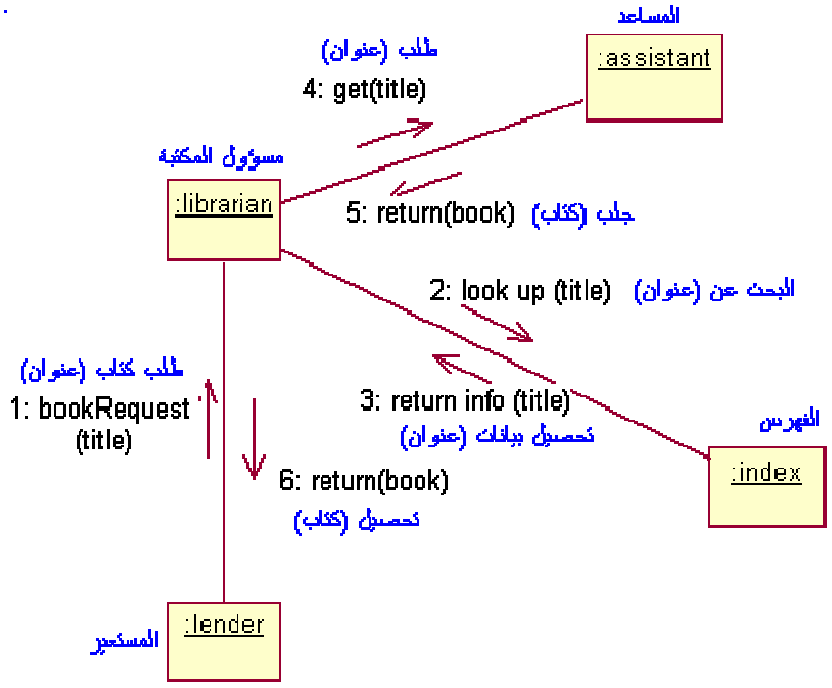
مخطط الصنفيات The Class Diagram



شكل 14: مخطط الصنفيات

رسم مخططات الصنفيات جانب أساسي لأي منهج للتصميم بالمنحى للكائن، لذلك ليس بالغريب أن تقدّم لنا UML الصيغة المناسبة لها. سوف نرى أنه بإمكاننا استخدام مخطط الصنفيات في مرحلة التحليل و كذلك في مرحلة التصميم – سوف نقوم باستعمال صيغ مخططات الصنفيات لرسم خريطة للمفاهيم العامة التي يمكن للمستفيد = الزبون أن يستوعبها (و سوف نسمّيها النموذج المفاهيمي Conceptual Model). وهي بالإضافة إلى مخطط وقائع الاستخدام، تجعل من النموذج المفاهيمي أداة قوية لتحليل المتطلبات.

مخططات التعاون The Collaboration Diagrams

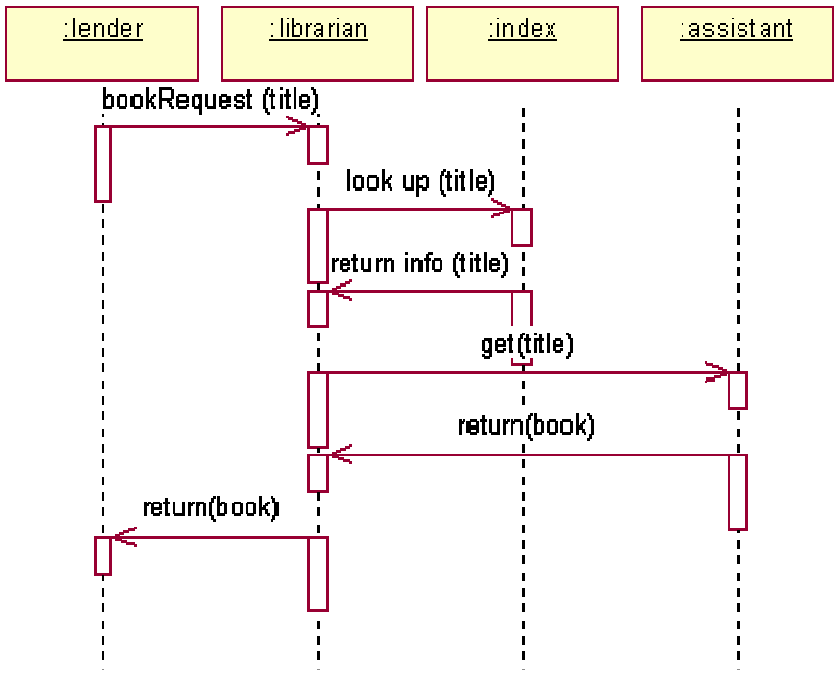


شكل 15: مخططات التعاون.

و نحن نقوم بتطوير برامج المنحى الكائني؛ إذا احتاج برنامجنا لأن يقوم بـــأي شـــيء فسيكون ذلك بواسطة تعاون الكائنات. يمكننا رسم مخططات التعاون لوصف الكيفية التي تتعاون بها الكائنات فيما بينها بالطريقة التي نريدها.

هنا مثال جيد عن لماذا UML هي مجرد صيغة أكثر من كونها عملية حقيقية لتطـــوير البرمجيات. سوف نرى أن ترميز UML للمخطط بسيط جدا، و لكن عملية تصميم تعاون فعّال، (لنقل "تصميم برنامج راسخ و يسهل صيانته") ، يعدّ صعبا بالتأكيد. ربمـــا علينـــا تخصيص فصل كامل يتناول الخطوط العريضة لمبادئ التصميم الجيّد، مع أن الكثير من مهارات التصميم تأتي من الخبرة.

مخطط التتابع Sequence Diagram

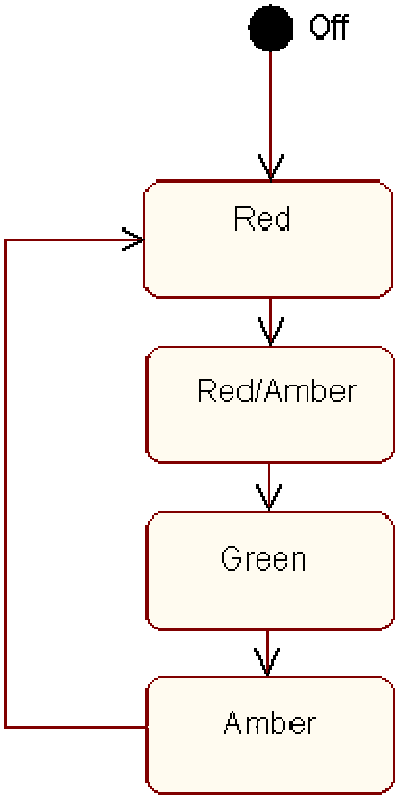


شكل 16: مخطط التتابع في UML

مخطط التتابع في حقيقته له علاقة مباشرة بمخطط التعاون و يقوم بعرض نفس المعلومات، و لكن بشكل يختلف قليلا. الخطوط المنقطة إلى أسفل المخطط تشير إلى الزمن، لذلك فما نشاهده هنا هو وصف لكيفية تفاعل الكائنات في نظامنا عبر الزمن.

بعض أدواة نمذجة UML ، مثل برنامج ناشيونال روز Rational Rose، يمكنها توليد مخطط التتابع آليا من مخطط التعاون، و هذا ما حدث تماما عندما تم رسم المخطط أعلاه – مباشرة من المخطط الذي في الشكل 15.

مخططات الحالة State Diagrams



شكل 17: مخطط حالة التحول

بعض الكائنات يمكنها في أي وقت محدد أن تكون في حالة ما. مثلا، يمكن للإشارة الضوئية أن تكون في إحدى الحالات التالية:

مطفأة، حمراء، صفراء، خضراء

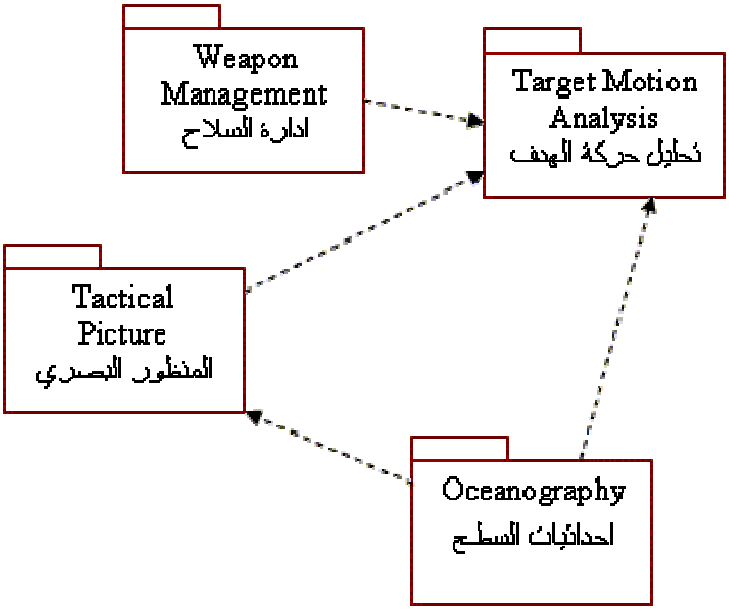
أحيانا، يكون تعاقب التحوّلات بين الحالات معقّد جدا – في المثال أعلاه، لا يمكننا أن ننتقل من حالة "خضراء" إلى حالة "حمراء" (و إلا تسبّبنا في حادث!).

و برغم أن الإشارة الضوئية قد تبدو مثالا بسيطا، إلا أن التهاون في التعامل مع الحالات يمكن أن يؤدّي إلى وقوع أعطال جدية و محرجة في برامجنا.

خذ مثلا – فاتورة غاز مرسلة إلى مستهلك توفّي منذ أربع سنوات – هذا يحـــدث فـــي الواقع و سبب ذلك أن المبرمج في نقطة ما لم يأخذ في اعتباره تحوّلات الحالة.

و كما يمكن لتحوّلات الحالة أن تكون معقّدة، فإن UML تقدّم صيغة تسمح لنا بتصويرها و نمذجتها.

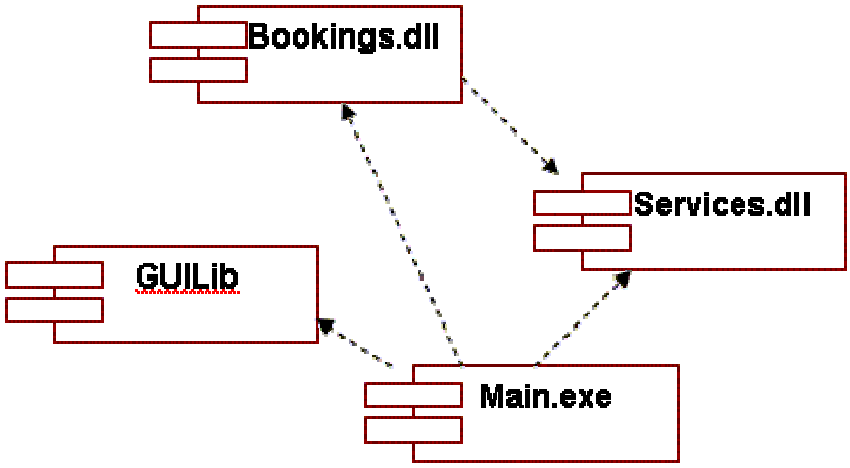
مخططات التحزيم Package Diagrams



شكل 18: مخططات التحزيم في UML

أي نظام= منظومة لا يكون صغيرا يحتاج إلى أن يقسّم إلى أجزاء "chunks" أصـــغر حجما و أسهل للفهم، و تتيح لنا مخططات التحزيم في UML نمذجة هذه الأجزاء بطريقة بسيطة و فعّالة. سوف نتعرّف بكثير من التفصيل على هذا النمـــوذج عنـــد استكـــشافنا للأنظمة الضخمة في فصل "معمار النظام".

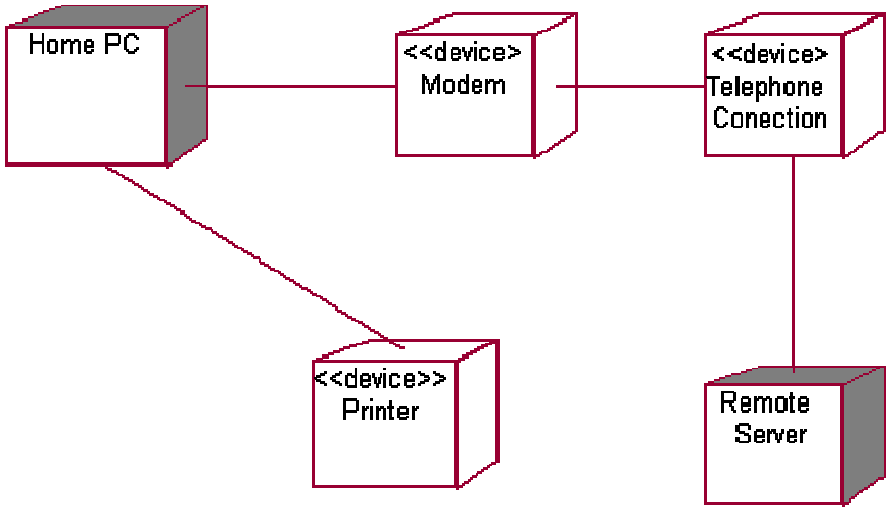
مخططات المكونات Component Diagrams



شكل 19: مخطط المكونات في UML.

يتشابه مخطط المكونات مع مخطط التحزيم – فهو يسمح لنا بترميز كيفية فصل أو تقسيم نظامنا، و كيف يعتمد كل قالب على آخر فيه. عموما، يركّز مخطط المكونات على المكونات الفعلية للبرنامج (الملفات، الترويسات headers، مكتبات الربط، الملفات التنفيذية، الحزم packages) و ليس بالفصل المنطقي أو الفكري كما في مخطط التحزيم. مرة أخرى، سوف نتعمق في دراسة هذا المخطط في فصل معماريات النظام.

مخططات التجهيز Deployment Diagrams



شكل 20: مخطط التجهيز في UML.

تقدم لنا UML نموذجا يمكننا من خلاله التخطيط لكيف سيتم تجهيز برنامجنا. مثلا، المخطط أعلاه يعرض توصيفا مبسطا لجهاز حاسوب شخصي.

ملخص

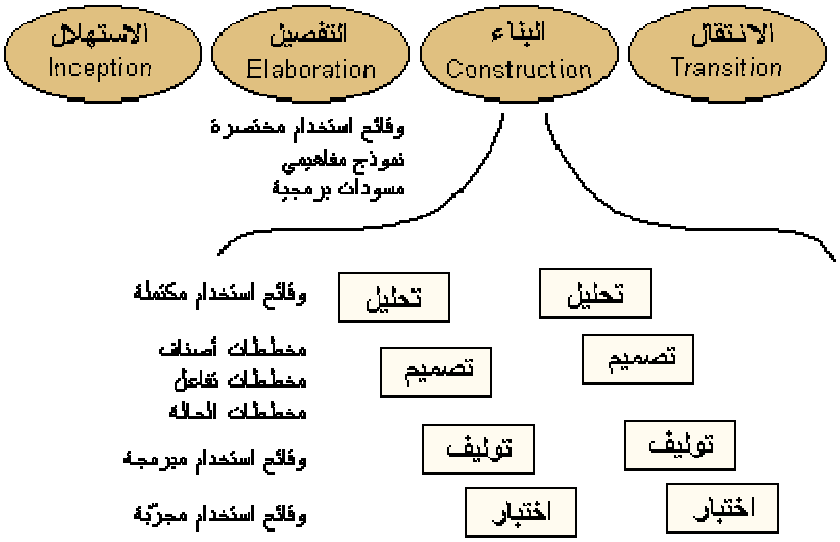
يوفّر UML عدة نماذج مختلفة لوصف النظام. القائمة التالية تعرضها كلها مع جملة واحدة توجز الغرض من كل نموذج:

- وقائع الاستخدامUse Cases – "كيف سيتفاعل نظامنا مع العالم الخارجي؟"
- مخطط الصنفيات Class Diagram – "ما هي الكائنات التي نحتاجها؟ و ما علاقتها؟"
- مخطط التعاون Collaboration Diagram – "كيف تتعامل الكائنات مع بعض؟"
- مخطط التتابع Sequence Diagram – "كيف تتعامل الكائنات مع بعض؟"

- مخطط الحالة State Diagram – "ما الحالات التي يجب أن تكون عليها الكائنات؟"
- مخطط التحزيم Package Diagram– "كيف سنقوم بقولبة عملنا؟"
- مخطط المكونات Component Diagram – "كيف سترتبط مكونات برنامجنا؟"
- مخطط التجهيز Deployment Diagram – "كيف سيتم تجهيز البرنامج؟"

الفصل 5: طور الاستهلال

بدءا من هنا في هذه الدروس، سنقوم بالتركيز على حالة دراسية case study لنصف كيف يتم تطبيق UML على مشروعات حقيقية. يجب أن تستخدم العملية التي تم عرضها في الفصل الأول، كما هو مبين في المخطط التالي:



شكل 21: العملية الخاصة بالحالة الدراسية.

في المخطط ، قمنا بوضع اسم كل نموذج model سيتم إنتاجه في كل طور. مثلا، في طور التصميم سنقوم بإعداد مخططات الصنفيات Class Diagrams ، مخططات التفاعل Interaction Diagrams و مخططات الحالة State Diagrams . و بالطبع سنقوم باستكشاف هذه المخططات خلال هذه الدروس.

للتذكير بطور الاستهلال Inception Phase، فان النشاطات الأساسية في هذا الطور هي:

- وضع رؤية للمنتوج.

- توليد واقعة عمل business case.
- تحديد نطاق المشروع.
- توقع التكلفة العامة للمشروع.

حجم الطور يعتمد على المشروع. فمشروع للتجارة الالكترونية قد يحتاج لأن نقتحم السوق بأسرع ما يمكن، و النشاطات الوحيدة في طور الاستهلال قد تكون تحديد الرؤية و الحصول على التمويل من مصرف بمساعدة مخطط العمل.

بالمقابل، مشروع له علاقة بالأمور العسكرية قد يحتاج إلى تحليل المتطلبات، و تعريف المشروع، دراسات مسبقة، دعوة لتقديم العطاءات، إلى آخر ذلك. كل هذا يعتمد على نوع المشروع.

في هذه الدروس، سنفترض أن طور الاستهلال مكتمل فعلا. و أنه قد تم إعداد دراسة أعمال تتناول بالتفصيل الاحتياجات المبدئية للزبون و وصف لنموذج الأعمال الخاص به.

الفصل 6: طور التفصيل

في طور التفصيل، ينصب اهتمامنا على استكشاف المسألة بالتفصيل و فهم احتياجات الزبون و طبيعة عمله، و تطوير الخطة بتفصيل أكبر.

يجب أن نضع أنفسنا في الإطار الذهني الصحيح لكي نتصدى لهذه المرحلة بشكل سليم. يجب علينا أن لا ننغمس كثيرا في التفاصيل – بخاصة تفاصيل التنفيذ.

يجب أن تكون لدينا نظرة واسعة جدا للنظام و أن نتفهم الخطوط العريضة فيه. يسمّي كروشتن Kruchten هذا الأمر : *نظرة بعرض ميل و عمق بوصة.*

المسودات Prototyping

النشاط الرئيسي في طور التفصيل هو تسهيل الصعاب و تيسير المخاطر. كلما تم تحديد المخاطر و قضي عليها مبكرا، كلما كان تأثيرها أقل على المشروع.

إعداد مسودات برمجية (برامج أولية) للأجزاء الصعبة و المناطق الإشكالية في المشروع سوف يساعد كثيرا في تيسير المخاطر. و مع أخذنا في الاعتبار إننا لا نريد الخوض في تفاصيل التنفيذ و التصميم في هذه المرحلة، فان المسودات prototypes يجب أن تكون مركزة جدا و لا تتناول إلا النواحي التي تعنينا.

هذه المسودات ، و بعد الانتهاء منها يمكن طرحها جانبا، أو يستفاد منها و يعاود استخدامها في طور البناء.

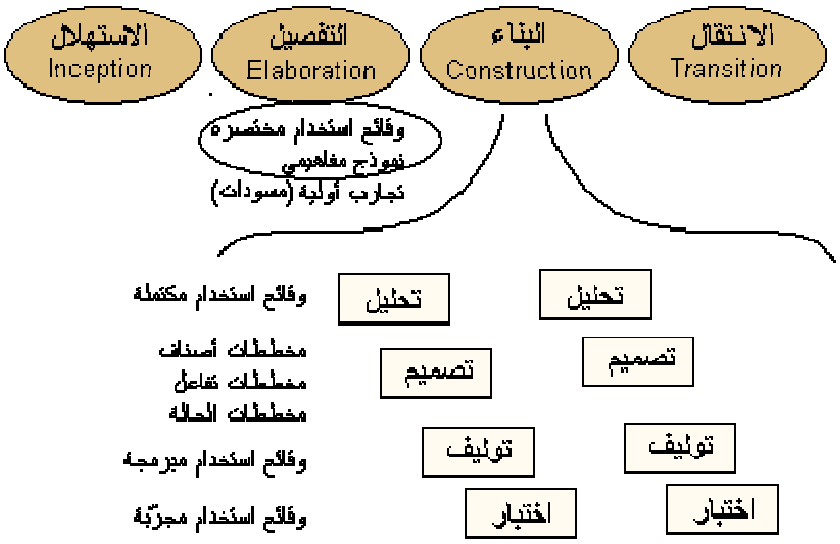
المخرجات Deliverables

بجانب المسودّات، سنقوم بتطوير اثنين من نماذج UML لمساعدتنا في تفهّم المسألة بشكلها العام.

النمـــوذج الأول هـــو واقعـــة الاســـتخدام Use Case Model. هـــذا النمـــوذج سيساعدنا على فهم ما الذي سيقوم به النظام ، و كيف سيبدو من وجهـــة نظـــر "العـــالم الخارجي" (مثل: المستخدمون، أو ربما نظام آخر سيرتبط به).

النموذج الثاني هو النموذج المفاهيمي Conceptual Model. هذا النموذج يسمح لنا ، عن طريق UML، بطبع صيغة رسومية لمعطيات الزبون. سوف تصف المفاهيم العامـــة لمعطيات الزبون، و كيف هي العلاقة فيما بينها. لبناء هذا النموذج سوف نستعمل مخطط الصنفيات Class Diagram من UML. وسوف نستخدم النموذج المفاهيمي هذا في طور البناء Construction Phase لبناء الصنفيات و الكائنات البرمجية.

سنقوم بتغطية هذين النموذجين ، بتعمق في الفصلين القادمين.



شكل 22: نموذجان ل UML بنيا خلال طور التفصيل

ملخص

يهتم طور التفصيل بتطوير آلية لفهم المشكلة بدون القلق بشأن تفاصيل التصميم المتعمقة (فيما عدا تلك المتعلقة بالمخاطر التي يتم تحديدها و المسودات البرمجية اللازمة).

نموذجان سوف يساعداننا في هذا الطور: نموذج وقائع الاستخدام والنموذج المفاهيمي.

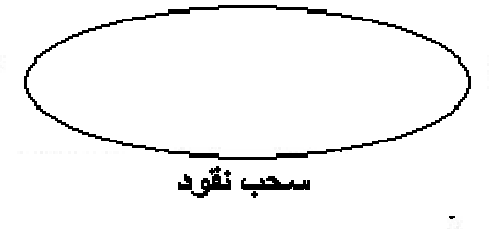
الفصل 7: نمذجة وقائع الاستخدام

واقعة الاستخدام هي من أدوات UML القوية، هي ببساطة وصف لمجموعة من التفاعلات بين المستخدم و النظام. و من خلال بناء مجموعة من وقائع الاستخدام، يمكننا وصف كامل النظام الذي نخطط لإنشائه، بصورة واضحة و موجزة.

عادة ما يتم وصف وقائع الاستخدام باستعمال توليفات من (الفعل / الاسم) – على سبيل المثال: "دفع الفواتير" ، "تحديث المرتبات" أو "إنشاء حساب".

مثلا، إذا كنا نكتب برنامجا لنظام التحكم بالصواريخ، فإن وقائع الاستخدام المعتادة قد تكون: "إطلاق الصاروخ"، أو "بدء العدّ التنازلي".

بجانب الاسم الذي سنعطيه لواقعة الاستخدام، سوف نقدم شرحا نصيا كاملا للتفاعلات التي ستنشأ بين المستخدم و النظام. هذه الشروح النصية سوف تنتهي في الغالب لتكون أكثر تعقيدا، لكن UML تقدم لنا ترميزا بسيطا و مدهشا لتمثيل واقعة الاستخدام، كالتالي:

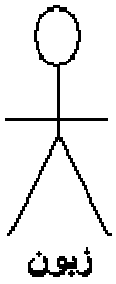


شكل 23: ترميز لواقعة استخدام

اللاعب Actor

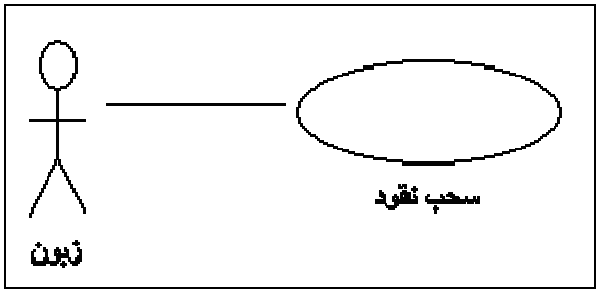
واقعة الاستخدام لا يمكنها بدء الأحداث أو التفاعلات من تلقاء نفسها. **اللاعب** هو شخص ما الذي **يمكنه** بدء أو تفعيل واقعة الاستخدام. مثلا، إذا كنا نقوم بتطوير نظام مصرفي، و كان لدينا واقعة استخدام تسمى "سحب النقود"، فيمكننا الإقرار بأننا نحتاج

لزبائن للتمكن من سحب هذه النقود، على ذلك سيكون **الزبون** أحد اللاعبين لدينا. مـــرة أخرى، الترميز لهذا اللاعب سيكون بسيطا:



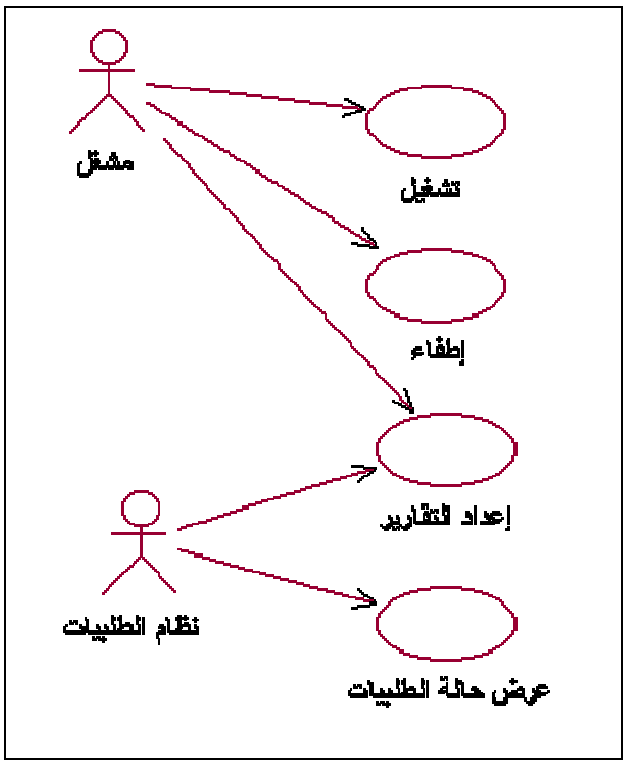
شكل 24: ترميز اللاعب في UML

لو تعمقنا أكثر، فإن اللاعبين يمكن أن يكونوا أكثر من مجرد أناس. اللاعب قد يكون أي شيء خارج النظام يقوم بتفعيل واقعة الاستخدام، مثل جهاز حاسوب آخر. أكثر من ذلك قد يكون اللاعب مفهوما أكثر تجريدا مثل **الوقت**، أو **تاريخا** معينا. مثلا، قد يكون لـــدينا واقعة استخدام اسمها "حذف الطلبيات القديمة" في منظومة لمناولة الطلبيات، و اللاعـــب الذي سيقوم بتفعيل هذه الواقعة قد يكون تاريخ "آخر يوم عمل". كما لاحظنا، اللاعبـــون مرتبطون بوقائع الاستخدام، حيث أن اللاعب هو الذي ســـيقوم بتفعيـــل أو بـــدء واقعـــة استخدام معينة. يمكننا تمثيل ذلك عل مخطط واقعة استخدام من خلال وصـــل اللاعـــب بواقعة الاستخدام:



شكل 25:علاقة لاعب بواقعة استخدام

من الواضح أنه بالنسبة لمعظم الأنظمة، يمكن للاعب الواحد التفاعل مع مجموعة من وقائع الاستخدام، كما أن واقعة الاستخدام الواحدة يمكن تفعيلها من قبل أكثر من لاعب مختلف. هذا يقودنا إلى مخطط واقعة استخدام متكامل، كما هو في المثال التالي:



شكل 26: نظام كامل تم وصفه باستخدام اللاعبين و وقائع الاستخدام

الغرض من وقائع الاستخدام

إن بساطة تعريف "واقعة الاستخدام" و "اللاعب"، مع بساطة تخيّل وقائع الاستخدام من خلال نموذج UML، سوف تجعلنا معذورين إذا اعتقدنا أن وقائع الاستخدام أمرها هيّن و سهل، و أنها أبسط من أن نقلق بشأنها. هذا خطأ. إن وقائع الاستخدام قوية بصورة كبيرة.

- وقائع الاستخدام تحدد النطاق العام للنظام. إنها تمكننا من تصوّر حجم و نطاق عملية التطوير بالكامل.

- وقائع الاستخدام شبيهة جدا بالمتطلبات، و لكن بينما المتطلبات تميل لأن تكون مبهمة و مربكة و ملتبسة و مكتوبة بشكل سيء، نجد أن البناء المحكم لوقائع الاستخدام يجعلها أكثر تركيزا.
- مجموع وقائع الاستخدام تشكل النظام بالكامل. مما يعني أن أي شيء لم يتم تغطيته في وقائع الاستخدام هو خارج حدود النظام المراد تطويره. لذا فإن مخطط وقائع الاستخدام متكامل بدون فجوات.
- إنها تمكن من التواصل بين العملاء و المطورين (ما دام المخطط بهذه السهولة، فالكل يستطيع فهمه).
- وقائع الاستخدام ترشد فرق التطوير خلال عمليات البناء التطوير– سوف نرى أن وقائع الاستخدام بمثابة العمود الفقري لعمليات التطوير، و ستكون المرجع لنا في كل ما نصنعه.
- سوف نرى كيف أن وقائع الاستخدام تقدم منهجية لتخطيط عملنا في التطوير ، و تسمح لنا بتقدير الوقت اللازم لانجاز العمل.
- وقائع الاستخدام تقدم الأساس لبناء اختبارات النظام .
- أخيرا، فإن وقائع الاستخدام تساعد في إعداد أدلة التشغيل!

غالبا ما تصدر ادعاءات بأن وقائع الاستخدام هي ببساطة أسلوب للتعبير عن متطلبات النظام. واضح أن كل من يقول بهذا قد غاب عنه الغاية من وقائع الاستخدام![8]

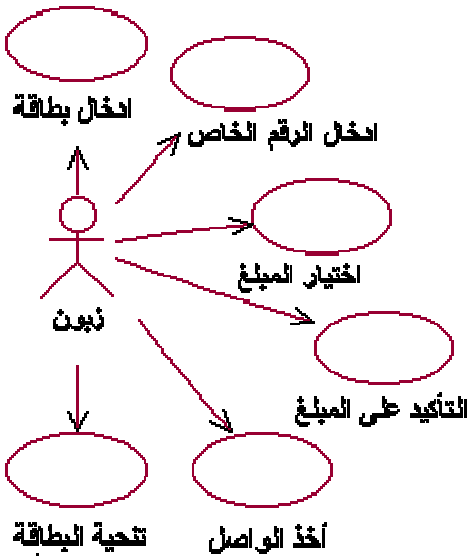
مدى كثافة واقعة الاستخدام

قد يكون من الصعب تقرير إلى أي مدى يجب أن تكون عليه كثافة وقائع الاستخدام، مثلا، هل على كل تفاعل بين النظام و المستخدم أن يكون واقعة استخدام، أو يجب على واقعة الاستخدام الواحدة أن تحوي كل التفاعلات؟ مثلا، لنأخذ آلة صرف النقود الآليATM، نحتاج لبناء نظام ATM يسمح للمستخدم أن يسحب النقود. ربما سيكون لدينا المجموعة التالية من التفاعلات الشائعة في هذا التصور:

[8] برغم هذا، فإن وقائع الاستخدام لها علاقة كبيرة بالمتطلبات. انظر المرجع [9] والمعالجة الممتازة لموضوع المتطلبات من خلال وقائع الاستخدام.

- إدخال البطاقة
- إدخال الرقم الخاص
- اختيار المبلغ المطلوب
- التأكيد على المبلغ المطلوب
- تنحية البطاقة
- أخذ الواصل

هل يجب أن يكون لكل من هذه الخطوات واقعة استخدام مثل "إدخال الرقم الخاص"؟



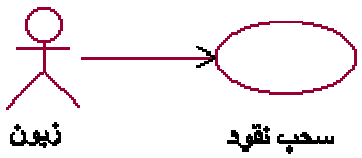
شكل 27: مخطط واقعة استخدام مفيد؟

هذا خطأ تقليدي عند بناء وقائع الاستخدام. هنا، قمنا بتوليد عدد كبير من وقائع الاستخدام الصغيرة، و الغير مهمة في أغلبها. في أي نظام لا يكون صغيرا، سوف نجد أنفسنا و قد تكوّن لدينا عددا ضخما من وقائع الاستخدام، و سيكون التعقيد عندها جارفا. لمعالجة التعقيد حتى في الأنظمة الكبيرة جدا، نحتاج لأن نبقي وقائع الاستخدام على "مستوى أعلى" ما أمكن. أفضل طريقة للتقرب من واقعة الاستخدام هي أن نبقي القاعدة التالية محفورة في ذهننا:

واقعة الاستخدام يجب أن تحقق هدفا ينشده اللاعب

بتطبيق هذه القاعدة البسيطة على مثالنا السابق، يمكن أن نطرح سؤالا: "**هل الحصول على الواصل**" هو هدف الزبون؟ حسنا، ليس تماما. سوف لن ينتهي العالم إذا لم يتم إصدار هذا الواصل.

بتطبيق القاعدة على وقائع الاستخدام الأخرى، سوف نجد أنه في الحقيقة لا أحد منها تصف الهدف الذي ينشده المستخدم. هدف المستخدم هو **سحب النقود**، وهذا ما يجب أن تكون عليه واقعة الاستخدام!



شكل 28: واقعة استخدام أكثر تركيزا

هذه المقاربة قد تكون مؤلمة في البداية، حيث أننا قد تعودنا على "التفكيك الوظائفي"، حيث يتم تحليل و كسر المهام المعقدة و تحويلها إلى مهام أصغر و أصغر. سوف نرى لاحقا أن وقائع الاستخدام يمكن تفكيكها، لكننا الآن يجب أن نترك هذه الخطوة إلى الوقت الذي نبدأ فيه بعملية البناء.

توصيفات وقائع الاستخدام

كل واقعة استخدام تحوي مجموعة كاملة من التفاصيل النصية عن التفاعلات و التصورات التي تشملها الواقعة.

نلاحظ أنUML لا تحدد ما يجب أن يكون عليه شكل أو محتويات هذه الوثيقة – هذا يرجع للمشروعات أو الشركات لتحدده كيفما يناسبها[9]. بالنسبة لنا سوف تستعمل النموذج التالي:

واقعة الاستخدام	اسم واقعة الاستخدام
وصف مختصر	وصف موجز لواقعة الاستخدام
شروط سابقة	وصف للشروط التي يجب أن تتوفر قبل تفعيل واقعة الاستخدام
شروط لاحقة	وصف لما سيحدث عند انتهاء واقعة الاستخدام
المجريات الأساسية	قائمة بتفاعلات النظام التي ستأخذ مكانها وفق أكثر التصورات شيوعا. مثلا، بالنسبة لواقعة "سحب النقود"، ستكون "إدخال البطاقة"، "إدخال الرقم الخاص"، و هكذا ..
مجريات بديلة	وصف للتفاعلات البديلة المحتملة.
مجريات استثنائية	و صف للتصورات المحتملة عندما تقع أحداث غير متوقعة، أو لا يمكن التنبؤ بها.

شكل 29: نموذج لتوصيف واقعة استخدام

[9] هذا مثال جيد على أن UML تقدم صيغ و تركيبات، لكنها و بصورة متعمدة ، لا تحدد كيف يتم استعمال هذه الصيغ.

وقائع الاستخدام في طور التفصيل

واجبنا الأساسي في طور التفصيل هو تحديد أكبر عدد ممكن من وقائع الاستخدام الممكنة. مع أخذنا في الاعتبار مبدأ: " عرض ميل و عمق بوصة"، هدفنا هو تقديم ملامح عامة لأكبر عدد ممكن من وقائع الاستخدام – و لكن بدون الحاجة لتقديم تفاصيل كاملة لكل واقعة استخدام. هذا سيساعدنا على تجنب الحمل الثقيل للتعقيدات الزائدة.

في هذه المرحلة، سيكون كافيا، تقديم مخطط واقعة استخدام (اللاعبون مع وقائع الاستخدام)، إضافة إلى وصف مختصر لكل واقعة استخدام. و سيكون بمقدورنا مراجعة التفاصيل الدقيقة لوقائع الاستخدام أثناء طور البناء. حالما ننتهي من تحديد وقائع الاستخدام، يمكننا ربطها بالمتطلبات للتأكد من أننا تتبعنا كافة هذه المتطلبات.

في هذه المرحلة، إذا قمنا بتشخيص بعض وقائع الاستخدام ذات المخاطرة العالية، فسيكون من المهم استكشاف تفاصيلها. إن إنتاج نماذج أولية لها في هذه المرحلة سوف يساعد على تخفيف حدة المخاطر.

البحث عن وقائع الاستخدام

إحدى طرق العثور على وقائع الاستخدام هي من خلال المقابلات التي يتم إجراؤها مع المستخدمين المحتملين للنظام. هذه مهمة صعبة، لأن شخصين مختلفين سيعطيان في الأغلب صورتين مختلفتين بالكامل لما يجب أن يكون عليه النظام (حتى لو كانا يعملان في نفس الشركة)!

بالتأكيد، إن معظم عمليات التطوير سوف تتضمن بدرجة ما اتصالات مباشرة مع المستخدمين وجها لوجه. لكن حيث أنه من الصعب الحصول منهم على رؤية موحدة لما يجب على النظام القيام به، فقد وجد أسلوب آخر أصبح أكثر شعبية وهو: ورشة العمل.

ورش عمل التخطيط المشترك للمتطلبات (JRP)

Joint Requirements Planning Workshops (JRP)

أسلوب ورشة العمل تجمع جماعة من الأفراد معا ممن لهم اهتمام أو علاقة بالنظام الجاري تطويره و يسمون مجازا : (بحاملي الأسهم stakeholders) كل فرد في هذه الجماعة مدعو لإعطاء وجهة نظره في ما يجب على النظام أن يقوم به.

مفتاح النجاح لورش العمل هذه هم المنسقون. الذين يقودون الجماعة من خلال التأكد من أن النقاشات لا تخرج عن الموضوع الأساسي، و أن كل المشاركين يتم تشجيعهم لطرح آرائهم ، و أن هذه الآراء يتم رصدها بالكامل. المنسقون الجيدون لا يقدرون بثمن!

سيكون حاضرا أيضا في ورش العمل هذه مقرر الجلسات، الذي يتولى مهمة توثيق كافة المناقشات. المقرر قد يقوم بعمله من خلال الورقة و القلم ، لكن الأسلوب الأفضل هو أن يتم ربط إحدى أدوات CASE (هندسة البرمجيات بمساندة الحاسوب) أو برنامج رسم بآلة عرض و رسم المخططات مباشرة.

بساطة مخطط وقائع الاستخدام أمر حيوي هنا– كل المشاركين، حتى من لا دراية له بالحاسوب، يجب أن يكون قادرا على استيعاب مفهوم المخطط بكل بسهولة.

الطريقة السهلة لتسيير ورش العمل هي:

1. أن يتم أولا "عصف ذهني"* لاستعراض كافة اللاعبين المحتملين
2. بعدها، عصف ذهني آخر لاستعراض كافة وقائع الاستخدام المحتملة
3. حال الانتهاء من عملية العصف الذهني وطرح الأفكار، من قبل المجموعة، يتم تبرير كل واقعة استخدام من خلال صياغة وصف مبسط في سطر أو فقرة واحدة.
4. وضعها في أنموذج.

الخطوات 1 و 2 يمكن عكسهما حسب الرغبة.

فيما يلي بعض النصائح النافعة حول ورشة العمل:

- **لا تجهد نفسك كثيرا في محاولة إيجاد كل واقعة استخدام أو لاعب.** إنه من الطبيعي نشوء المزيد من وقائع الاستخدام لاحقا أثناء عملية التطوير.

- **إذا لم تستطع تبرير واقعة استخدام في الخطوة 3، فهذا قد يعني أنها ليست واقعة استخدام.** لا تتردد في إزالة أية واقعة استخدام تشعر بأنها غير صحيحة أو مكررة (هذه الوقائع سوف تعود ثانية إذا وجد حاجة إليها!)

النصائح أعلاه ليست ترخيصا لنكون متساهلين، لنتذكر أن فائدة العمليات التكرارية iterative هي أن كل شيء لا يشترط أن يكون صحيحا 100% في كل خطوة!

نصيحة حول العصف الذهني*

العصف الذهني ليس أمرا سهلا كما يبدو؛ في الواقع من النادر أن نجد عملية عصف ذهني تم إعدادها بطريقة جيدة. المفاتيح الأساسية التي علينا تذكرها عند المشاركة في جلسة عصف ذهني هي:

- توثيق كل الأفكار، لا يهم مدى ما تبدو عليه من غرابة أو غباء، الأفكار الغبية قد تتحول إلى أفكار معقولة جدا بعد فترة
- أيضا، الأفكار السخيفة قد تستدعي أفكارا منطقية في ذهن شخص آخر– هذا ما يسمى أفكار الوثب للأمام leapfrogging ideas
- لا يتم أبدا تقييم أو نقد الآراء. هذه قاعدة يصعب التقيّد بها – نحن نحاول كسر إحدى طبائع البشر هنا.

"**مممم... لا، هذا لن يصلح ، لن نزعج أنفسنا بتوثيق ذلك!**"

المنسق يجب أن يبقي عينيه على أصابع المشتركين و أن يحرص على أن كل الآراء قد تم رصدها، و أن كل المجموعة قد قامت بالمشاركة.

في هذا السياق، سيتم القيام بورشة عمل وقائع الاستخدام رفقة الزبون الذي سنتعامل معه.

* العصف الذهني Brainstorming تعني عملية طرح الأفكار على علاتها كما ترد إلى الذهن دون تريث أو تفكير في مدى معقوليتها. (المترجم)

ملخص

وقائع الاستخدام أسلوب فعال لنمذجة ما يحتاج النظام لعمله.

هي طريقة ممتازة للتعبير عن نطاق عمل النظام (ما بالداخل = مجموع وقائع الاستخدام؛ ما بالخارج = اللاعبون).

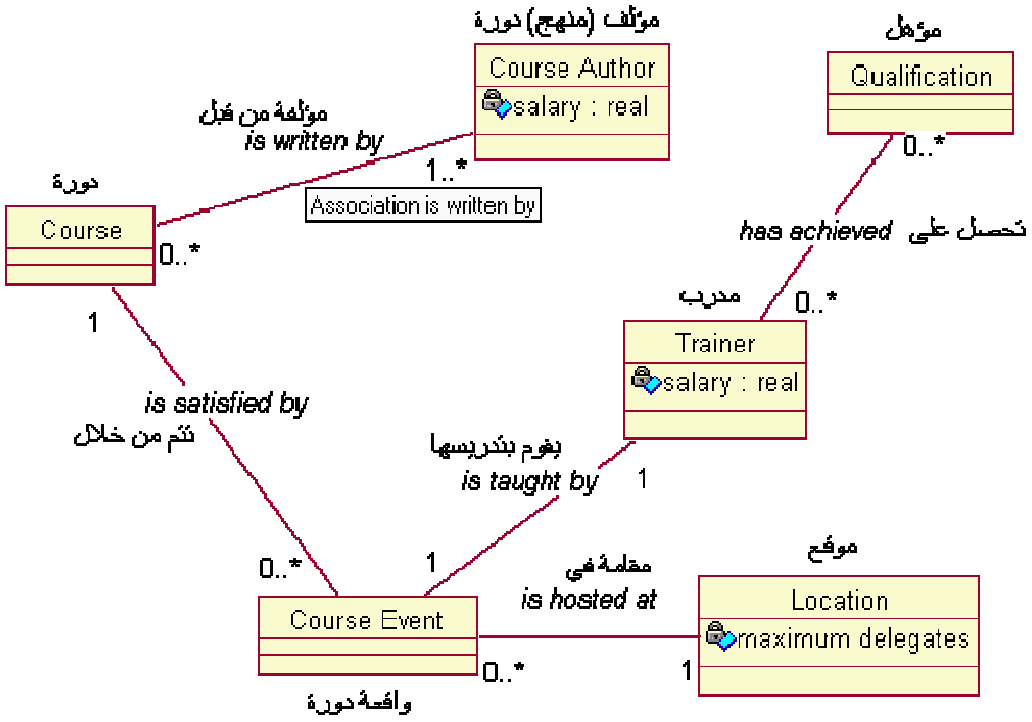
نحتاج إلى أن ننتبه لمدى كثافة وقائع الاستخدام التي تحوي تعقيدات.

أفضل وسيلة لبناء وقائع الاستخدام هي مع الزبون في ورشة عمل.

الفصل 8: نمذجة المفاهيم

نمذجة المفاهيم Conceptual Modeling (أحيانا تدعى : نمذجة النطاق العام Domain Modeling) هو النشاط الذي يهتم بإيجاد المفاهيم ذات الأهمية في نظامنا. هذه العملية تساعدنا على فهم المشكلة أكثر، و تدعم إدراكنا لمجال عمل الزبون الذي نخدمه.

مرة أخرى، UML لا تخبرنا كيف أو متى نقوم بعملية نمذجة المفاهيم، لكنها تقدم لنا الصياغة التي نعبر بها عن النموذج. النموذج الذي نحن بصدد استخدامه هو مخطط الصنفيات class diagram.



شكل 30: مخطط صنفيات في UML

بناء مخطط الصنفيات هو المفتاح لأية عملية تصميم بالمنحى للكائن object oriented design. سيوضح مخطط الصنفيات بشكل أساسي الهيكل العام للتوليف code النهائي الذي سننتجه.

عموما، في هذه المرحلة، لا يهمنا كثيرا تصميم النظام (ما زلنا نقوم بالتحليل)، لذا فإن مخطط الصنفيات الذي سنقوم بإعداده في هذه المرحلة سيكون أقرب لتخطيط مبدئي، و لن يحوي على أي تصميم نهائي.

مثلا، لن نقوم بإضافة صنفية لقائمة مرتبطة "LinkList" في هذه المرحلة، لأن هذا سيجرنا إلى اتخاذ حل معين في فترة مبكرة جدا .

في النموذج المفاهيمي، نهدف إلى رصد كل المفاهيم و الأفكار بطريقة يمكن للزبون أن يتعرف عليها. فيما يلي بعض الأمثلة الجيدة لمثل هذه المفاهيم:

- **مصعد** في نظام مراقبة الصعود
- **طلبية** في نظام التسوق المنزلي
- **لاعب كرة القدم** في نظام تحويل كرة القدم (أو لعبة كرة قدم في جهاز "بلاي ستيشن!")
- **حذاء رياضي** في نظام إدارة المخزون لمحل أحذية
- **غرفة** في نظام حجز الغرف

بعض الأمثلة السيئة جدا للمفاهيم هي:

- **معالج_تطهير_الطلبيات** العملية التي تقوم دوريا بإزالة الطلبيات القديمة من النظام
- **صمام_حدث** العملية الخاصة التي تنتظر 5 دقائق ثم تقوم بإيقاظ النظام ليفعل شيء ما
- **نموذج_بيانات_زبون** الشاشة التي يتم فيها تعبئة بيانات زبون جديد في نظام محل تسوّق
- **جدول_أرشفة** جدول قاعدة بيانات تحوي قائمة بكل الطلبيات القديمة

أفضل قاعدة هنا هي:

إذا كان المفهوم غير واضح بالنسبة للزبون، فمن المحتمل أنه ليس بمفهوم!

المصممون يكرهون المرحلة المفاهيمية – لا يستطيعون الانتظار، ينطلقون مباشرة داخل ضجيج التصميم. سوف نرى لاحقا بأن النموذج المفاهيمي سيتحول ببطء إلى مخطط صنفيات تصميم متكامل مع دخولنا لطور البناء .

إيجاد المفاهيم

ينصح باستخدام نفس الأسلوب لإيجاد وقائع الاستخدام – الأفضلية لورش العمـــل – و مرة أخرى، بحضور أكبر عدد ممكن من المهتمين.

المقترحات الناتجة عن العصف الذهني Brainstorm و رصد **كل** هذه المقترحات. حال الانتهاء من طرح كل الآراء، يتم كمجموعة مناقشة و تبرير كل اقتراح. تطبيق القاعـــدة المجربة بأن: المفهوم يجب أن يكون واضحا للزبون، و إبعاد أي مفهوم يكـــون خـــارج نطاق المسألة، و إبعاد كل ما يؤثر سلبا على التصميم.

استخلاص المفاهيم من المتطلبات

تعد وثيقة المتطلبات مصدر جيد لبناء المفاهيم. يقترح كريك لارمان Craig Larman (المصدر[2]) ترشيح المفاهيم التالية من المتطلبات:

- الكائنات المادية و المحسوسة
- الأماكن
- الحركات/ المعاملات
- أدوار الأفراد (الزبون، موظف المبيعات)
- الحاويات لمفاهيم أخرى
- الأنظمة الخارجية الأخرى (مثل قواعد البيانات النائية)
- الأسماء المجردة (مثل عطش)
- التقسيمات التنظيمية
- الأحداث (مثل: الحالات الطارئة)
- القواعد/ اللوائح والسياسات
- التسجيلات / ملفات رصد الحركة

هنا تجدر الإشارة إلى بعض النقاط ، قبل كل شيء، جمع المفاهيم بطريقـــة ميكانيكيـــة يعتبر أسلوبا ضعيفا. القائمة أعلاه هي اقتراحات جيدة، لكنه من الخطأ اعتبار أنه يكفي أن

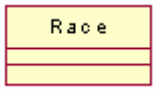
ننطلق من وثيقة المتطلبات مع قلم لتضليل بعض العبارات الواردة فيها وجعلها كمفاهيم. يجب الحصول على مشاركة الزبون و رأيه هنا.

ثانيا، الكثير من المتمرسين يقترحون أن يتم استخلاص **عبارات الأسماء** من الوثيقة. هذه الطريقة شاع استخدامها لما يقرب من 20 عاما، و بالرغم من أنه لا يوجد ما يعيبها في حد ذاتها، إلا أنه ليس صحيحا الإيحاء بأن البحث الآلي عن هذه الأسماء سينتج عنه قائمة جيدة من المفاهيم، و التصنيفات classes. للأسف، اللغة العادية فيها الكثير من الغموض و الالتباس الذي يمنع مثل هذه الطريقة الآلية. سوف نكرر ثانية – **مشاركة الزبون أمر أساسي!**

النموذج المفاهيمي في UML

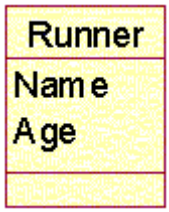
الآن، و بعد أن رأينا كيف نقوم باستكشاف المفاهيم، نحتاج لمعرفة كيفية رصد هذه المفاهيم في UML. سوف نستخدم الملامح الرئيسية لمخطط الصنفيات class diagram.

نقوم بتمثيل المفهوم الذي لدينا بمربع بسيط، رفق عنوان= اسم المفهوم (بأحرف كبيرة عادة) في أعلى المربع.



شكل 31: مفهوم "سباق" Race بحسب UML (من أجل منظومة لسباق الخيل)

لاحظ أن داخل المربع الكبير يوجد مربعين خاليين أصغر حجما. المربع الأوسط سوف يستعمل قريبا لرصد سمات attributes المفهوم – سيتم شرحه بعد لحظات. المربع السفلي يستخدم لرصد سلوك behavior المفهوم – بعبارة أخرى ، ماذا يمكن للمفهوم فعله. تقرير سلوك المفهوم أو ماهية تصرفاته خطوة معقدة، و سوف نقوم بتأجيل هذه الخطوة حتى نصل إلى مرحلة التصميم للبناء. لذلك ليس هناك ما يدعو للقلق حول السلوك الآن.



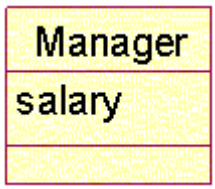
شكل 32: رصد سمات و سلوك المفهوم في UML

في المثال أعلاه، قررنا أن كل متسابق runner لديه اثنين من السمات "الاسم" و "العمر". و قد تركنا المنطقة السفلى خالية لوقت لاحق عندما نقرر ماذا يمكن ل"متسابق" أن يفعله.

إيجاد السمات

نحتاج إلى أن نقرر ما هي السمات الخاصة بكل مفهوم – و مرة أخرى، فإن جلسة عصف ذهني مع ذوي العلاقة ستكون في الأغلب أفضل وسيلة لتحقيق ذلك.

غالبا، ما تثار مجادلات حول ما إذا كان السمة في حد ذاتها هي مفهوم آخر أم لا. مثلا، لنقل أننا بصدد منظومة لإدارة العاملين وقررنا أن أحد المفاهيم فيها هو "المدير". ثم اقترحنا سمة له و هي"مرتب"، كالتالي:



شكل 33: مفهوم مدير مع سمة "مرتب"

هذا يبدو جيدا، لكن أحدهم قد يجادل بأن "مرتب" هي أيضا مفهوم. لذلك هل علينا ترقيتها من سمة إلى مفهوم؟

تجري العديد من جلسات النمذجة فتنحدر نحو مناقشات عقيمة حول قضايا مثل هذه، و النصيحة هنا هي ببساطة أن لا نقلق بشأنها: *إذا كان هناك شك، فلنقم بصنع مفهوم آخر.*

هذا النوع من المشاكل على كل حال عادة ما تنحل من تلقاء نفسها لاحقا ، كما أنه حقيقة ليس من المجدي إضاعة وقت ثمين للنمذجة على مثل هذه المناقشات!

شكل 34: مدير و مرتب، مفهومين منفصلين

إرشادات لإيجاد السمات

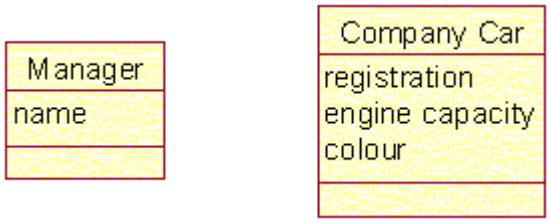
القواعد المجربة التالية قد تساعد في محاولة التقرير بين المفاهيم و السمات – لكن لننتبه للنصيحة أعلاه فلا نقلق كثيرا لمسألة التفريق هذه. *إذا كان هناك شك، فلنقم بصنع مفهوم آخر.*

- الجمل strings و الأرقام ذات القيمة المفردة عادة ما تكون سمات.[10]
- إذا وجدت خاصية property لمفهوم لا يمكنها فعل أي شيء، فقد تكون سمة – مثلا في مفهوم "مدير"، يبدو واضحا أن خاصية "اسم" تبدو كسمة. "سيارة شركة" تبدو مثل مفهوم، لأننا نحتاج لتخزين معلومات عن كل سيارة مثل رقم التسجيل و لونها.

الروابط Associations

الخطوة التالية هي تقرير حول كيفية ارتباط المفاهيم بعضها ببعض. في أية منظومة نمطية، بعض المفاهيم على الأقل سيكون لها علاقة مفاهيمية من نوع ما مع المفاهيم الأخرى. مثلا، عودة إلى منظومة إدارة العاملين، لنفحص المفهومين التاليين:

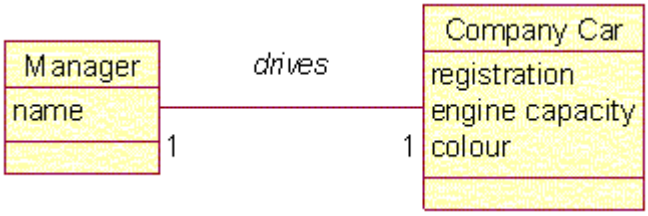
[10] لكن ليس دائما – هذه قاعدة عامة و لا يجب إتباعها بطاعة عمياء.



شكل 35: اثنان من المفاهيم مدير و سيارة شركة

هذان المفهومان مرتبطان، لأنه في الشركة الذي نحن بصدد تطوير منظومة لها، كل مدير يقود سيارة شركة.

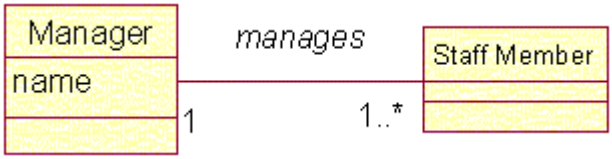
يمكننا التعبير عن هذه العلاقة في UML من خلال وصل هذين المفهومين بعضهما بخط مفرد (يسمى **رابط** association)، كالتالي:



شكل 36: "مدير" و "سيارة شركة" موصولان برابط

شيئان مهمان يجدر ملاحظتهما في هذا الرابط. قبل كل شيء، الربط لديه اسم دال – في هذه الحالة، "يقود" drives. ثانيا، هناك أرقام في طرفي الرابط. هذه الأرقام تصف الإلزامية cardinality لهذا الرابط، و تخبرنا العدد المسموح به لتمثلات instances كل مفهوم.

في هذا المثال، نحن نقول بأن "كل مدير يقود سيارة شركة واحدة"، و (في الاتجاه المعاكس) "كل سيارة شركة يقودها مدير واحد"



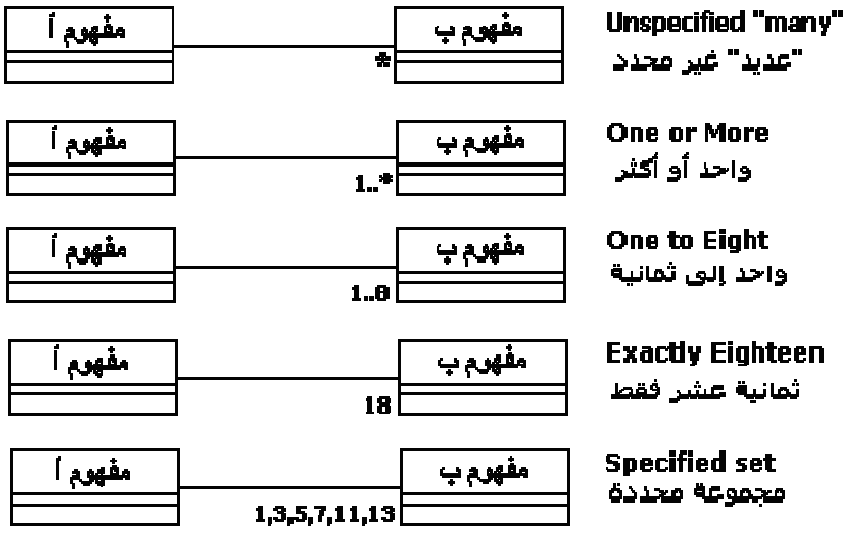
شكل 37: مثال آخر لرابط

في المثال أعلاه، نجد أن "كل مدير يدير 1 أو أكثر من العاملين"؛ و (بالجهة المقابلة)، "كل عامل يُدار من قبل مدير واحد".

كل رابط يجب أن يكون بهذا الشكل– عندما يتم قراءة الرابط باللغة العادية، يجب أن تكون بجملة مفيدة (خاصة للزبون). أيضا، عند تحديد عناوين للروابط، لنتجنب العناوين الضعيفة مثل "لديه" أو "هو مرتبط بـ" – استخدام لغة ضعيفة كهذه يمكنها بسهولة إخفاء مشاكل أو أخطاء كان يمكن الكشف عنها لو كان للروابط أسماء أكثر دلالة.

الإلزاميات المحتملة

بالأساس، لا توجد أية قيود على الإلزاميات Cardinalities التي يمكنك تحديدها. يستعرض المخطط التالي قائمة ببعض الأمثلة، برغم أنها ليست شاملة لكل الحالات. علامة * تشير إلى "عديد" many. لاحظ الفرق البسيط بين "*" و "*..1". الأول يشير بشكل غامض إلى "عديد"، فقد يعنى السماح بأي عدد من تجسدات المفهوم، أو ربما لم نتخذ قرارا بعد. الأخير أكثر تحديدا، ويعني ضمنيا السماح لعدد واحد أو أكثر.



شكل 38: أمثلة على الإلزاميات

بناء النموذج بالكامل

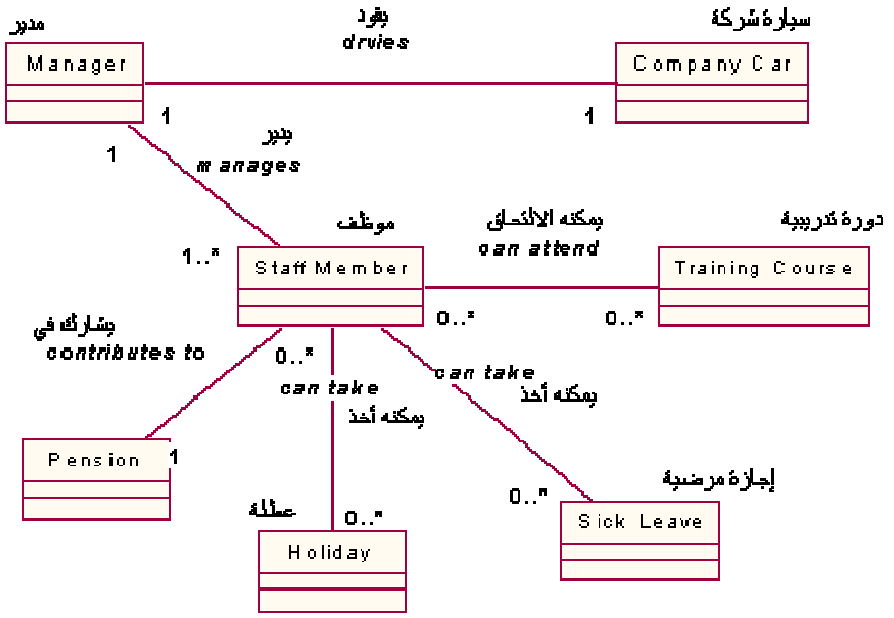
أخيرا، لنلقي نظرة على النظام المنهجي لتحديد الروابط بين المفاهيم. لنفترض أننا أكملنا جلسة طرح الأفكار و كشفنا عن عدة مفاهيم تخص نظام إدارة العاملين. مجموعة المفاهيم هذه في الشكل أدناه (تم شطب السمات للتوضيح).

شكل 39: مجموعة من المفاهيم لإدارة العاملين

أفضل وسيلة لعمل ذلك هي "تثبيت" أحد المفاهيم، و ليكن "مدير" و مراجعة كل مفهـــوم آخر مقابله. نسأل أنفسنا "هل توجد علاقة بين هذين المفهومين ؟" إذا كان كذلك، نـــشرع فورا في تسمية الرابطة بينهما، و نوع الإلزامية..

„ هل **للمدير** علاقة **بالفرد العامل** ؟ نعم، كل مدير يدير 1 أو أكثر من العمال. "
المدير و **سيارة الشركة**؟ نعم، كل مدير يقود سيارة واحدة.
المدير و **حساب تأمين تقاعدي** ؟ نعم، كل مدير يتشارك في حساب تقاعد واحد.

و هكذا حتى اكتمال النموذج. الخطأ الشائع في هذه المرحلة هو أن يتم تحديد مفهومين بينهما علاقة، ثم رسم خط رابط بينهما ثم تترك تسمية الرابط لوقت لاحق. هذا يزيد من عبء العمل لدينا. سنكتشف أنه حالما ننتهي من رسم الخطوط،؛ لن يكون لدينا أية فكـــرة عما كنا نقصده بها (و ستبدو عادة مثل السباغيتي)، و سنضطر لبدء العمل من جديد.



شكل 40: نموذج مفاهيمي بسيط، و كامل

من المهم أن نتذكر عند بناء النموذج أن الروابط أقل أهمية من السمات. أي رابط مفقود سيكون من السهل استرجاعه عند التصميم، لكنه من الصعب تدارك سمات تم إغفالها.

أمر آخر، قد يكون من المغري التمادي في إضفاء المزيد من الروابط "للاحتيـــاط"، و تكون النتيجة بعدها مخططا معقدا و أكثر إرباكا. لذلك فإن القاعدة الجيدة هنا هو أن نركز على المفاهيم و السمات، و محاولة تثبيت الروابط الأكثر وضوحا.

عند نهاية النمذجة، يجب أن يكون المخطط ذو معنى للزبون عند "إعادة قراءة" النموذج باللغة العادية.

ملخص

النموذج المفاهيمي وسيلة فعالة من أجل تحليل أعمق للمسألة.

لاحقا، سنقوم بتوسعة النموذج نحو نواحي التصميم.

سيكون هذا النموذج في النهاية أحد أهم المعطيات عند بناء التوليف code.

لبناء النموذج، استخدم تقنيات ورش العمل كما في وقائع الاستخدام.

الفصل 9: ترتيب وقائع الاستخدام

لدينا الكثير من العمل أمامنا – كيف يمكننا تقسيم العمل لتكرارات iterations بسيطة و سهلة القياد حسب ما وصفناها في بدايات هذه الدروس؟

الإجابة هي بالتركيز على وقائع الاستخدام لدينا. في كل تكرار=معاودة، نقوم بتصميم و توليف و اختبار *القليل* فقط من وقائع الاستخدام. إذا و بصورة فعالة، نحن قد قمنا فعلا بتحديد كيفية تقسيم العمل إلى تكرارات أو معاودات – الشيء الوحيد الذي لم نقم به هو تحديد ما هو الترتيب الذي به نتصدى لهذه الوقائع.

لتخطيط هذا الترتيب، نقوم بإعطاء كل واقعة استخدام *رتبة rank*. الرتبة هي ببساطة رقم يشير إلى التكرار الذي سيتم فيه تطوير واقعة الاستخدام. أية أداة Case (التصميم بمساندة الحاسوب) جيدة سوف تتيح رصد الرتب كجزء من النموذج.

لا توجد قواعد ثابتة و سريعة لكيفية تخصيص الرتب. الخبرة و المعرفة في مجال تنشئة البرمجيات تلعب دورا كبيرا في تحديد هذه الرتب. هذه بعض الإرشادات عن أي من وقائع الاستخدام يجب إعطاؤها رتبة أعلى (بمعنى، أن يتم تطويرها مبكرا):

- وقائع الاستخدام ذات المخاطرة العالية
- وقائع الاستخدام التي تعد أساسية للمعمار
- وقائع الاستخدام التي تتضمن مساحات واسعة من وظائفية النظام
- وقائع الاستخدام التي تتطلب بحثا مكثفا، أو تقنية جديدة
- "المكاسب السريعة"
- ذات العائد الكبير للمستخدم

بعض وقائع الاستخدام تحتاج في سبيل تطويرها إلى عدة تكرارات. قد يكون السبب في هذا كبر حجم واقعة الاستخدام عند التصميم و التوليف والاختبار فلا يسعها تكرار واحد. أو بسبب أن واقعة الاستخدام تعتمد على استكمال مجموعة وقائع استخدام أخرى ("بدء التشغيل" مثال تقليدي على ذلك).

يفترض في هذا أن لا يسبب مشاكل – ببساطة يتم تقسيم واقعة الاستخدام إلى بضعة نسخ. مثلا، لدينا واقعة الاستخدام الضخمة التالية، و التي سيتم تطويرها عبر ثلاثة تكرارات. في نهاية كل تكرار، لا تزال واقعة الاستخدام تؤدي مهمة مفيدة، و لكن بدرجة محدودة.

واقعة استخدام "إطلاق طوربيد":

- نسخة 1أ السماح للمستخدم بتحديد الهدف (رتبة : 2)
- نسخة 1ب السماح للمستخدم بتجهيز السلاح (رتبة : 3)
- نسخة 1ج السماح للمستخدم بإطلاق السلاح (رتبة : 5)

ملخص

وقائع الاستخدام تسمح لنا بجدولة عملنا عبر تكرارات متعددة.

نعطي رتبة لكل وقائع الاستخدام لتحديد أولوياتها في التصدي لها.

ترتيب وقائع الاستخدام مبنية على معرفتنا وخبرتنا الخاصة.

بعض الإرشادات المجربة سوف تساعد في الأيام الأولى.

بعض وقائع الاستخدام سوف تمتد لأكثر من تكرار.

الفصل 10: طور البناء

في هذا الفصل القصير، سوف نراجع ما قمنا بعمله، و ما المطلوب فعله لاحقا.

في طور التفصيل Elaboration Phase ، احتجنا لفهم المسألة بأوسع قدر ممكن، من غير الدخول في تفاصيل كثيرة. قمنا ببناء نموذج وقائع استخدام Use Case Model، و أنشأنا أكثر ما يمكن من وقائع استخدام. لم نقم بملء كامل تفاصيل وقائع الاستخدام، بل قدمنا وصفا موجزا لكل واقعة.

الخطوة التالية كانت بناء النموذج المفاهيمي conceptual model، حيث قمنا برصد المفاهيم التي ستحكم عملنا في التطوير. هذا النموذج المفاهيمي سيقدم لنا الأساسات التي سيرتكز عليها التصميم.

بعد ذلك قمنا بإعطاء رتبة لكل واقعة استخدام، وخلال ذلك، وضعنا مخططا لأولويات ترتيب تطوير واقعة الاستخدام.

بهذا يكتمل طور التفصيل. من المفترض إجراء مراجعة شاملة لهذا الطور، واتخاذ قرار بشأن ما إذا سيتم الاستمرار في المشروع أم لا. فبعد كل هذا قد نكتشف خلال طور التفصيل أننا لا نستطيع في الواقع تقديم حل مناسب لزبوننا – من الأفضل أن نكتشف ذلك الآن من أن نكتشفه في نهاية التوليف!

البناء

الآن و نحن في طور البناء، نحتاج لبناء المنتوج، و أخذ النظام لمرحلة يمكن فيها توريده لبيئة الزبون.

لنتذكر أن خطتنا العامة للتصدي للعمل هو أن نتبع سلسلة من التدفقات=الشلالات القصيرة waterfalls ، مع عدد صغير من وقائع الاستخدام يتم تطويرها في كل تكرار. في نهاية كل تكرار، سوف نراجع تقدم العمل، و يفضل تحديد الإطار الزمني للتكرار.

في الحالة المثلى، سنطمح لانجاز منظومة تعمل (ولو، طبعا، بصورة محدودة) مع نهاية كل تكرار.

كل مرحلة من التدفق ستنتج مجموعة من الوثائق أو بعبارة أخرى نماذج UML.

- عند التحليل، سنقوم بإعداد بعض وقائع الاستخدام بأكثر توسع (أو كاملة)
- عند التصميم، سنقوم بإعداد مخططات الصنفيات Class Diagrams ، و نماذج التفاعل Interaction Models، و مخططات الحالة State Diagrams
- عند التوليف Code ، سنقوم بإنتاج توليف يعمل و قد تم اختبار وحداته

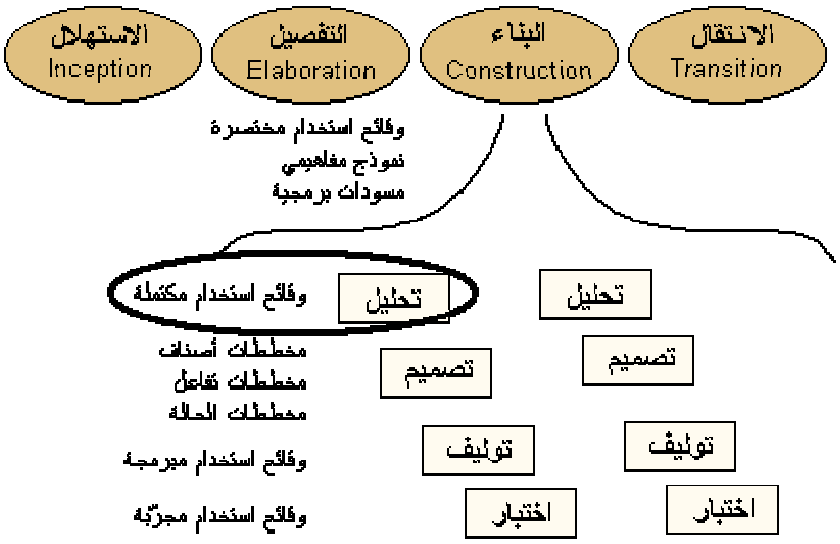
ثم يتم اختبار التكرار (كل وقائع الاستخدام يجب أن تكون شغالة بطريقة يمكن استعراضها)، بعد ذلك نصل إلى المراجعة.

ملخص

قمنا باستكمال التفصيل Elaboration ، و الآن نحن جاهزين لبدء البناء. سوف ننظر في كل نموذج على حدة ونرى كيف يفيدنا في عمليات البناء.

الفصل 11: طور البناء: التحليل

المرحلة الأولى في طور البناء هو التحليل. سنحتاج إلى زيارة ثانية لوقائع الاستخدام التي سنبنيها في هذا التكرار، و نقوم بتحسين و توسيع هذه الوقائع. إننا بالتركيز على التفاصيل الكاملة لعدد صغير فقط من وقائع الاستخدام للتكرار الواحد، سنقلص من مقدار التعقيد الذي علينا مناولته في وقت واحد.



شكل 41: مرحلة التحليل عند البناء

لنتذكر، رغم إننا في طور البناء، فنحن لا زلنا في مرحلة التحليل – ولو أن هذه المرة بتحليل أكثر تفصيلا من ذلك التحليل التي قمنا به في طور التفصيل. لذلك، يجب أن نأخذ في اعتبارنا أننا فقط مهتمون بالمشكلة أو المسألة، و ليس بالحل. إذن نحن ننظر في ما يجب أن يقوم به النظام، دون القلق بشأن كيف سيقوم به.

عودة لوقائع الاستخدام

خلال التفصيل Elaboration ، قمنا بإنتاج وقائع استخدام قصيرة و قررنا تأجيل الخوض في التفاصيل (مثل التدفق الأساسي، التدفق البديل، الشروط المسبقة و اللاحقة) حتى يأتي دور طور البناء. الآن حان الوقت لاستكمال كافة التفاصيل (لكن فقط لوقائع الاستخدام التي سنتعامل معها في هذا التكرار).

واقعة الاستخدام:	وضع الرهان
وصف موجز:	يقوم المستخدم بالمراهنة على فرس معين بعد اختيار السباق
اللاعبون:	المراهن
المتطلبات:	R2.3; R7.1
شروط مسبقة:	
شروط لاحقة:	
التدفق الرئيسي:	
تدفقات بديلة:	
تدفقات استثنائية:	

شكل 42: واقعة استخدام موجزة، وضع الرهان

الشكل أعلاه هو مثال لواقعة استخدام موجزة، و كل قسم معنون فيها يجب تعبئته. أفضل وسيلة لتوضيح عملية تعبئتها هو استخدام مثال محدد، لذا دعونا ننظر في واقعة استخدام **وضع رهان:**

شروط مسبقة

يصف هذا القسم شروط النظام التي يجب استيفاؤها قبل أن تأخذ واقعة الاستخدام مكانها في الواقع. في مثال وضع رهان، قد يكون جيدا تحديد الشرط المسبق التالي:

"المستخدم قد قام بتسجيل نفسه بنجاح".

واضح، أن نظام المراهنة يجب أن يتحقق من صلاحية الزبائن قبل أن يباشروا عملية المراهنة. عموما، عملية التحقق من المستخدم ليست جزءا من واقعة الاستخدام، لذلك يجب أن نضمن بأن الشرط قد تم تحقيقه قبل أن تبدأ عملية المراهنة.

شروط لاحقة

تصف الشروط اللاحقة الحالة التي يكون عليها النظام بعد انتهاء واقعة الاستخدام. جرى العرف أن يتم كتابة الشرط اللاحق بصيغة الماضي. على ذلك في مثالنا لوضع الرهان، سيكون الشرط اللاحق كالتالي:

"المستخدم وضع رهانه، و تم تسجيل الرهان من قبل النظام"

قد يوجد أكثر من شرط لاحق، بحسب مخرجات واقعة الاستخدام. هذه الشروط اللاحقة المختلفة يتم وصفها باستخدام لغة "إذا كان– إذاً" "if then". مثال:

"إذا كان المستخدم جديدا، إذاً، يتم إنشاء حساب للمستخدم".
"إذا كان المستخدم مسجلا، إذاً، يتم تحديث بيانات المستخدم".

التدفق الرئيسي

يصف قسم التدفق الرئيسي مجريات الأحداث المنتظر أو الغالب وقوعها أثناء واقعة الاستخدام. ومن المتوقع، في واقعة استخدام "وضع الرهان"، أن تجري بعض الأمور بطريقة خاطئة. ربما يقوم المستخدم بإلغاء الحركة. قد لا يملك المستخدم الرصيد الكافي لوضع الرهان. كل هذه الأحداث يجب أخذها في الاعتبار، لكن في الغالب، التدفق الأكثر احتمالا خلال واقعة الاستخدام هذه هي أن المستخدم سيضع رهانه بنجاح.

في التدفق الرئيسي، يجب تفصيل التفاعلات بين اللاعب و النظام. فيما يلي التدفق الرئيسي لواقعة "وضع رهان":

(1) عند استهلال وضع الرهان من قبل المراهن، يتم طلب قائمة بسباقات اليوم من النظام، و (2) عرضها على الشاشة

(3) يختار المراهن السباق الذي سيراهن عليه [A1] ، و (4) يظهر النظام قائمة بالمتسابقين في هذا السباق

(5) يختار المراهن الفرس المتسابق ليراهن عليه [A1] و يدخل مبلغ الرهان [E1]

(6) يقوم المستخدم=المراهن بتأكيد العملية و (7) و يعرض النظام رسالة تأكيد

لاحظ أن كل تفاعل بين اللاعب و النظام قد تم تجزئته إلى خطوات. في هذه الحالة، يوجد لدينا سبع خطوات في التدفق الرئيسي لواقعة الاستخدام. العلامات [A1] و [E1]سيتم توضيحها بعد قليل، عند النظر إلى التدفقات البديلة و التدفقات الاستثنائية.

التدفقات البديلة

التدفقات البديلة Alternate flows أو المجريات البديلة هي التدفقات الأقل حدوثا (لكن محتملة) خلال واقعة الاستخدام. التدفق البديل عادة ما يتشارك في الكثير من خطواته مع التدفق الرئيسي، لذلك يمكننا التنويه للنقطة التي منها ينطلق التدفق البديل. لقد قمنا بذلك في الخطوة (3) من التدفق الرئيسي أعلاه، من خلال التنويه[A1]. هذا بسبب أن المستخدم عند اختياره للسباق الذي سيراهن عليه، قد يقوم بإلغاء العملية. يمكنه أيضا إلغاء العملية عند الخطوة 5، عندما يكون عليه إدخال قيمة مبلغ الرهان.

" يقوم المستخدم بإلغاء العملية

شرط لاحق –> لا يتم وضع أي رهان"

في هذه الحالة، نتج عن التدفق البديل تغيير في الشرط اللاحق – لا يتم وضع أي رهان.[11]

[11] بعض ممارسي UML يفضلون القول بأن التدفق البديل ينتج عنه دوما نفس الشروط اللاحقة التي للتدفق الرئيسي. هذا مثال آخر على قابلية UML على تطبيقها بطرق مختلفة. البعض يفضل أن يدع التدفق البديل ليكون أي شيء ممكن لكن أقل عمومية، و ينتج عنه أي شرط لاحق يريده.

التدفقات الاستثنائية

أخيرا، الحالات الاستثنائية يتم وصفها في التدفقات الاستثنائية. بعبارة أخرى، التدفق الذي يجب أن يتم عندما يحدث خطأ، أو عند وقوع حدث لا يمكن بطريقة أخرى التنبؤ به.

في مثال "وضع رهان"، قد يكون لدينا الاستثناء التالي:

" (E1) رصيد المستخدم لا يكفي لتغطية الرهان. يتم تنبيه المستخدم و تنتهي واقعة الاستخدام"

عندما ننتقل إلى برمجة التوليف. يجب مقاربة البنود التي تحت التدفق الاستثنائي مع الاستثناءات في البرنامج – إذا كانت اللغة المستخدمة تدعم الاستثناءات exceptions، العديد من اللغات الحديثة تدعمها – مثل جافا و سي++ و دلفي و آدا.

واقعة الاستخدام بعد اكتمالها

واقعة الاستخدام:	وضع الرهان
وصف موجز:	يقوم المستخدم بالمراهنة على فرس معين بعد اختيار السباق
اللاعبون:	المراهن
المتطلبات:	R2.3; R7.1
شروط مسبقة:	قام المستخدم بتسجيل الدخول بنجاح
شروط لاحقة:	تم وضع الرهان و تسجيله من قبل النظام
التدفق الرئيسي:	(1) عند استهلال وضع الرهان من قبل المراهن، يتم طلب قائمة بسباقات اليوم من النظام، و (2) عرضها على الشاشة (3) يختار المراهن السباق الذي سيراهن عليه [A1] و (4) يظهر النظام قائمة بالمتسابقين في هذا السباق (5) يختار المراهن الفرس المتسابق ليراهن عليه [A1] و يدخل مبلغ الرهان [E1] (6) يقوم المستخدم=المراهن بتأكيد العملية و (7) و يعرض النظام رسالة تأكيد
تدفقات بديلة:	(A1) يقوم المستخدم بإلغاء العملية. شروط لاحقة –> لا يتم وضع أي رهان
تدفقات استثنائية:	(E1) رصيد المستخدم لا يكفي لتغطية الرهان. يتم تنبيه المستخدم و تنتهي واقعة الاستخدام

شكل 43: وصف لواقعة استخدام كاملة

مخطط التتابع في UML

إعداد واصفات واقعة الاستخدام أمر صعب. الكثير من الناس يجدون صعوبة خاصة في التفريق بين التحليل و التصميم – كثيرا ما تصبح واصفات واقعة الاستخدام وقد شـــابهها الكثير من خيارات تتعلق أصلا بالتصميم.

فيما يلي مثال عن واقعة استخدام "وضع رهان":

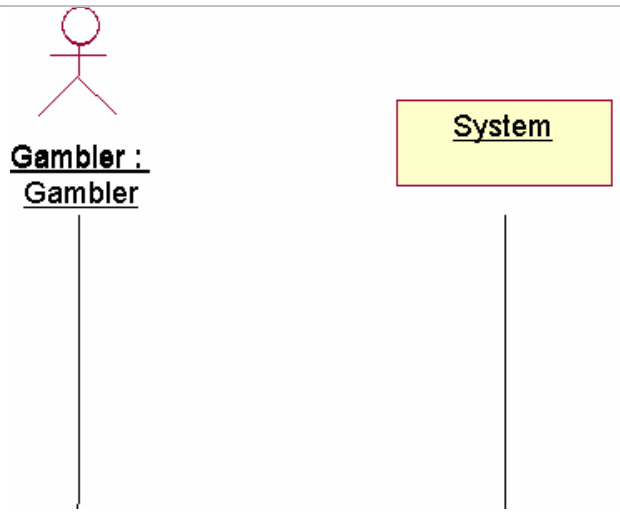
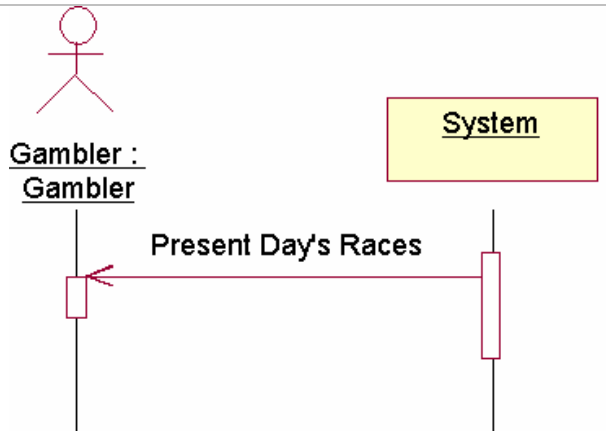
"يختار المستخدم السباق الذي سيراهن عليه. يقوم النظام باستعلام قاعدة البيانات و يجمع مصفوفة array تحوي المتسابقين في السباق"

هذا وصف سيء لواقعة استخدام. عندما نتحدث عن قاعدة بيانات الـــسباق و الإشـــارة للمصفوفات، فنحن نكبل أنفسنا بخيارات تصميم محددة.

عند بناء وقائع الاستخدام، علينا أن نتعامل مع النظام و كأنه "صندوق أســـود"، يقـــوم باستقبال الطلبات من اللاعب و إرجاع النتائج له. نحن لا يهمنا (حتى الآن) كيف يعمـــل الصندوق الأسود من أجل تلبية الطلبات.

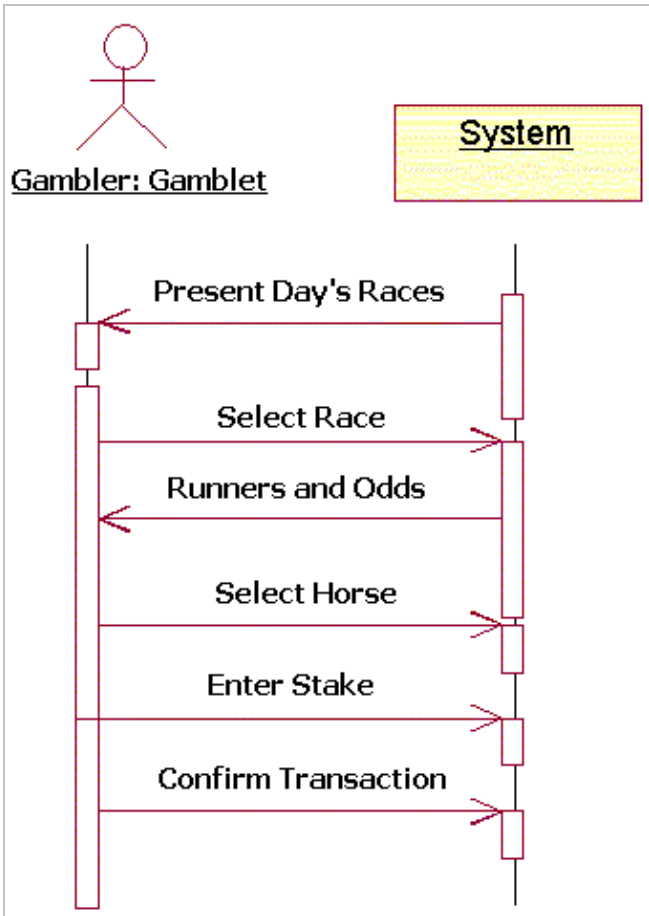
لذا و في هذا السياق، نحن ننصح باستخدام مخطط التتابع Sequence Diagram في UML. مخطط التتابع –أو التوالي– مفيد في عدة حالات ، خاصة في مرحلة التصميم. إلا أنه عموما، يمكن استخدام المخطط عند التحليل ليساعدنا في تحليل هذا الصندوق الأسود في النظام. فيما يلي سوف نرى كيف يعمل هذا المخطط:

نضع اللاعب على يسار المخطط، وعلى يمين المخطط نعبر عن كامل النظام بمربع واحد.	

بعدها، نضيف "خطوط زمنية" timelines عمودية. الخطوط تمثل مرور الزمن بالاتجاه للأسفل.	
التفاعل بين المستخدم و النظام يتم التعبير عنه كخط بسهم بين النظام و اللاعب. يتم كتابة وصف التفاعل بمحاذاة الخط.	

نستمر في إضافة التفاعلات للأسفل على خط الزمن.

المربعات الطويلة على خط الزمن تشير إلى متى يكون اللاعب أو النظام نشطا "active". هذا التنويه أو الترميز يكون أكثر أهمية عند التصميم – أما بالنسبة للآن، فعمليا ليس لها أهمية (تم إضافة هذه المربعات من قبل أداة التصميم).



حال الانتهاء من مخطط التتابع، ستكون مهمتنا سهلة وآلية تماما من أجل وصف التدفق الرئيسي لواقعة الاستخدام. لا حاجة لنا للرسم المضني لهذه المخططات لكل تدفق بديل أو استثنائي، بالرغم أنها تستحق ذلك في حالة كونها بدائل معقدة جدا أو مثيرة للاهتمام.

ملخص

في هذا الفصل، انتقلنا نحو طور البناء. و ركزنا على بعض وقائع الاستخدام في التكرار، و استكشفنا التفاصيل التي نحتاجها لتطوير واقعة استخدام كاملة.

تعلمنا أساسيات مخطط UML جديد، و هو **مخطط تتابع النظام**، و رأينا أن هذا المخطط يمكنه الإفادة عند توليد واقعة استخدام مفصلة.

الآن و نحن لدينا التفاصيل الكامنة في وقائع الاستخدام، ستكون المرحلة التالية إنتاج تصميم مفصل. كنا قد تطلعنا إلى **ماذا** – الآن سنتطلع إلى **كيف**.

الفصل 12: طور البناء: التصميم

التصميم – مقدمة

إلى هنا، أحطنا بالكامل بأبعاد المشكلة أو المسألة التي نحاول إيجاد حل لها (بالنسبة لهذا التكرار). قمنا بتنشئة واقعة استخدام للتكرار الأول و تعمقنا في تفاصيلها، و نحن الآن جاهزين لتصميم الحل الخاص بهذه المسألة.

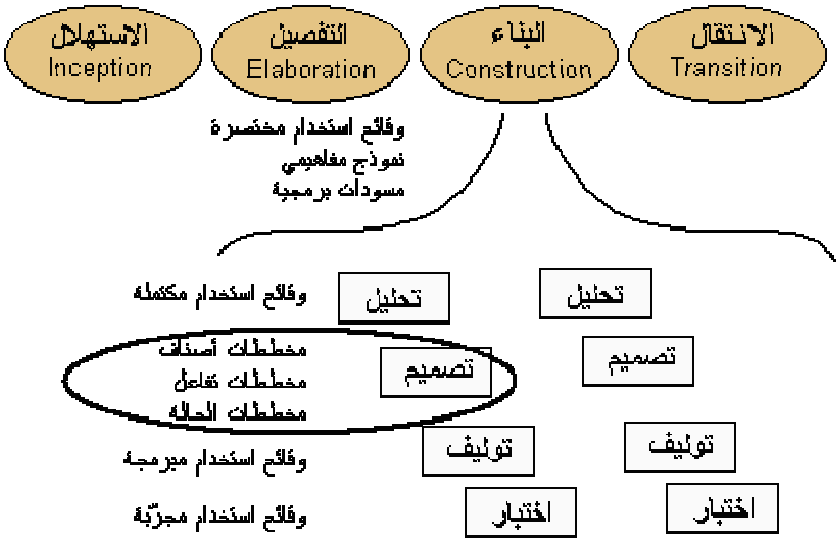
وقائع الاستخدام لا يتم إشباعها إلا بتفاعلات الكائنات. لذا في هذه المرحلة، يجب علينا تحديد الكائنات التي نحتاجها، و تحديد ما يجب أن تقوم به هذه الكائنات، و متى يجب أن تتفاعل مع بعضها.

توفر UML مخططين يسمحان لنا بالتعبير عن تفاعلات الكائنات، هما مخطط التتابع Sequence Diagram و مخطط التعاون Collaboration Diagram. هذان المخططان إجمالا متقاربان جدا (بعض البرامج المساعدة يمكنها توليد أحد المخططين من الآخر!)، مخططا التتابع و التعاون يسميان مخططات التفاعل Interaction Diagrams.

عند قيامنا بتحديد الكائنات التي نحتاجها، يجب توثيق صنفيات classes الكائنات التي لدينا، و كيف ترتبط هذه الصنفيات مع بعض. مخطط الصنفيات Class Diagram في UML يتيح لنا رصد هذه المعلومات. في الواقع، معظم العمل لتوليد مخطط الصنفيات قد تم انجازه بالفعل – سوف نستخدم النموذج المفاهيمي الذي سبق و أن قمنا بإنشائه كنقطة انطلاق.

أخيرا، نموذج مفيد سيتم بناؤه في مرحلة التصميم و هو نموذج الحالة State Model. المزيد من التفاصيل حول هذا لاحقا.

إذا، عند التصميم، سنقوم بتوليد ثلاثة أنواع من النماذج – مخطط التفاعل و مخطط الصنفيات و مخطط الحالة.



شكل 44: ما يتم تسليمه (المخرجات) في مرحلة التصميم.

تعاون الكائنات في واقع الحياة

إذا، وقائع الاستخدام لدينا سيتم إشباعها بتعاون الكائنات المختلفة. هذا ما يحدث فعلا في الحياة الواقعية. لنأخذ مكتبة ما كمثال. مكتبة يتم إدارتها من قبل أخصائية مكتبات في مكتب الاستقبال. أخصائية المكتبات مسؤولة عن تلبية طلبات الزبائن، و مسؤولة عن إدارة فهرس المكتبة. أخصائية المكتبة أيضا مسؤولة عن مجموعة من المساعدين. المساعدون مسؤولون عن إدارة أرفف المكتبة (أخصائية المكتبات لا يمكنها عمل ذلك – و إلا لن تتمكن من أداء عملها في مكتب الاستقبال بكفاءة).

الكائنات في منظومة المكتبة هذه هي:

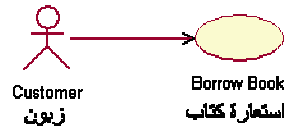
زبون

أخصائية مكتبات

مساعد

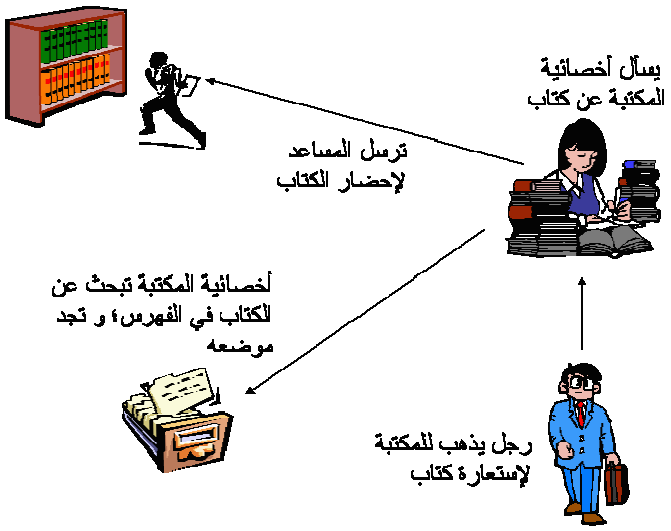
فهرس مكتبة

رفّ



لننظر في واقعة الاستخدام الجلية – استعارة كتاب

كيف يمكن إشباع واقعة الاستخدام هذه؟ لنفترض أن الزبون لا يعرف أين موقع الكتاب و يحتاج لمساعدة؟ سلسلة الأحداث يمكن أن تكون كالتالي:



1- الزبون يذهب لأخصائية المكتبة و يسأل عن كتاب "تطبيق UML".

2- أخصائية المكتبية تبحث في الفهرس عن اسم الكتاب. تجد بطاقة فهرس الكتاب، التي تبين أن الكتاب موضوع في الرف 4F.

3- أخصائية المكتبة تطلب من المساعد إحضار الكتاب من الرف 4F.

4- المساعد يحصل على الكتاب و يعطيه لأخصائية المكتبة.

5- أخصائية المكتبة تسجل خروج الكتاب و تسلمه للزبون.

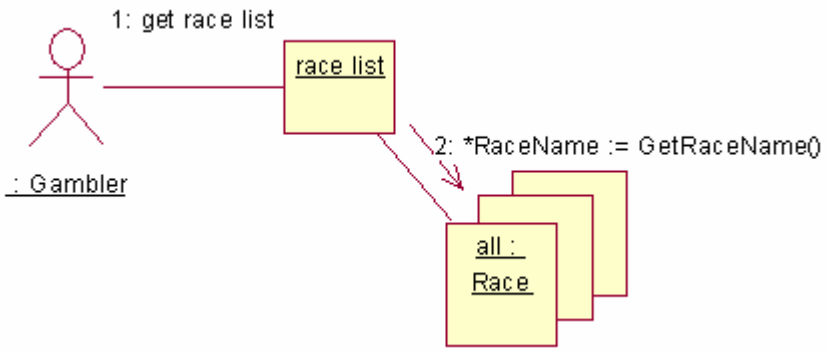
شكل 45: سلسلة الأحداث في "استعارة كتاب"

بالرغم من بساطة المثال السابق، فإنه لم يكن سهلا، بالذات فيما يخص تحديد مسؤوليات كل كائن. هذه أحد النشاطات الرئيسية في التصميم بالمنحى للكائن Object Oriented Design – وضع مسؤوليات كل كائن بصورة صحيحة. مثلا، إذا قررنا أن ندع أخصائية المكتبة تقوم بجلب الكتاب بنفسها، فنكون قد صممنا نظاما غير فعال على الإطلاق.

مخططات التعاون

في هذا القسم، سنتناول الصيغة النحوية لمخطط التعـــاون Collaboration Diagram في UML. و سنرى في القسم التالي كيف يتم استخدام هذا المخطط.

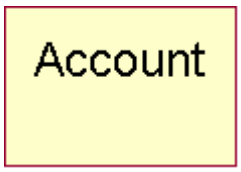
يتيح لنا مخطط التعاون رؤية التفاعلات بين الكائنات عبر مدة زمنية. فيما يلـــي مثـــال لمخطط تعاون كامل:



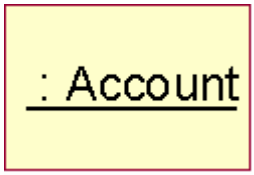
شكل 46: مخطط تعاون

صيغ التعاون: الأساسيات

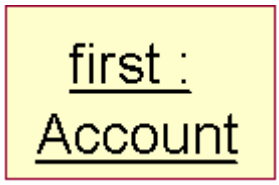
صنفية في مخطط تعاون يتم التنويه عنها كالتالي:



التجسد أو التمثل instance لصنفية (بعبارة أخرى، كائن) يتم ترميزه كالتالي:

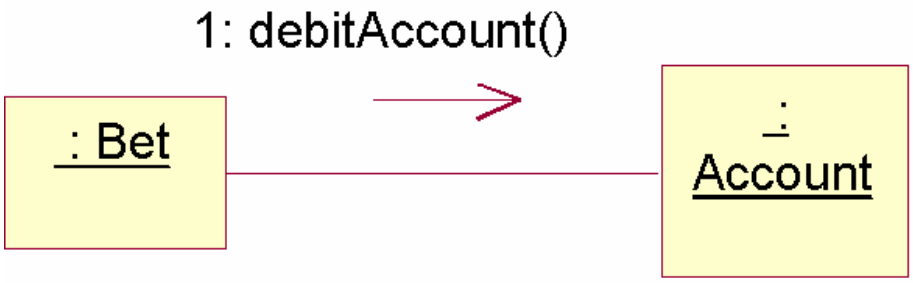


أحيانا، نجد أنه من المفيد أن نسمي تجسد الصنفية. في المثال التالي، نريد كائنا من صنفية حساب، و نريد أن نسميه "first" أول:

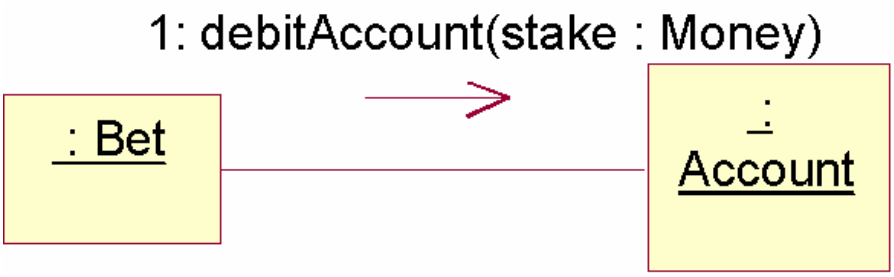


إذا أردنا لكائن أن يتواصل مع كائن آخر، نرمز لهذا بوصل الكائنين مع بعض بخط. في المثال التالي نريد وصل الكائن "bet" رهان مع الكائن "account" حساب.

حالما نقوم بترميز كائن موصول بآخر، يمكننا إرسال رسالة مسماة من كائن لآخر. في الرسم التالي، الكائن "bet" رهان يبعث برسالة للكائن "account" حساب، يخبره فيها بأن يجعل نفسه مدينا.



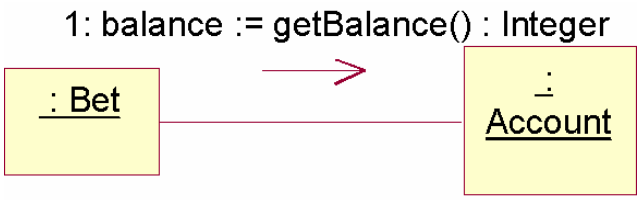
إذا أردنا إمرار معطيات parameters مع الرسالة، يمكننا تضمينها بين قوسين كما يلي. نوع بيانات المعطيات (في المثال، صنفية اسمها "Money") يمكن عرضها كخيار.



يمكن للرسالة أن تقوم بإرجاع (تناظر استدعاء الوظيفية function call في مرحلة البرمجة). ينصح بالصيغة التالية في مواصفات UML إذا كنا ننشد تصميما غير محايد لأية لغة. عموما، إذا كنت تستهدف لغة ما، يمكنك أن تكيف الصيغة لتطابق لغتك المفضلة.

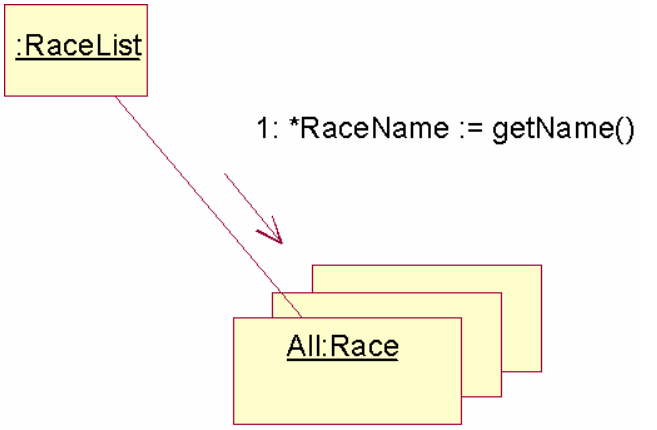
```
return := message (parameter: parameterType): returnType
```

في المثال التالي، يحتاج الكائن "bet" رهان لمعرفة رصيد حساب معين. يتم بعث الرسالة "getBalance" ، و يقوم الكائن حساب بإرجاع عدد صحيح:



مخططات التعاون: التوالي

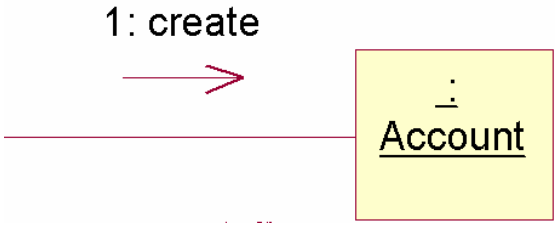
إذا أردنا إدخال متوالية loop في مخطط التعاون، نستخدم الصيغة التالية. في هذا المثال، كائن من صنفية "Race List" عبارة عن قائمة سباقات ويحتاج الكائن لتعبئة قائمته. من أجل ذلك: يطلب من كل عضو في صنفية "Race" أن يرجع له اسمه.



تشير علامة النجمة إلى أن الرسالة سيتم تكرارها، و بدلا من تحديد اسما لكائن واحـــد، استخدمنا الاسم "All" (الكل) للإشارة إلى أننا سنعيد الكرة مع كل الكائنات. أخيرا ستخدمنا ترميز UML لمجموعة كائنات عن طريق وضع مربعات الكائنات في صفوف.

مخططات التعاون: خلق كائنات جديدة

أحيانا، يقوم كائن بخلق و إنشاء create تمثل/تجسد جديد لكائن آخر. طريقة القيام بهذا تختلف بين اللغات، في UML تم توحيد عملية الإنشاء بالصيغة التالية:



حقيقة، الصيغة غريبة بعض الشيء – البعث برسالة تسمى "Create" (خلـــق/إنـــشاء) لكائن لم يوجد بعد![12]

[12] في الواقع العملي، نبعث برسالة إلى الصنفية، و في معظم لغات البرمجة نستدعي أيضا نوع خاص من الإجرائيات وهو المنشئ constructor..

ترقيم الرسائل

لنلاحظ أن كل الرسائل التي قمنا بتضمينها لحد الآن تتضمن رمز غامض و هو "1" محاد لها؟ يشير هذا إلى الترتيب الذي به يتم تنفيذ الرسالة من أجل إشباع واقعة الاستخدام. وكلما أضفنا المزيد من الرسائل (أنظر الأمثلة الذي ستلي)، نرفع رقم الرسالة بالتوالي.

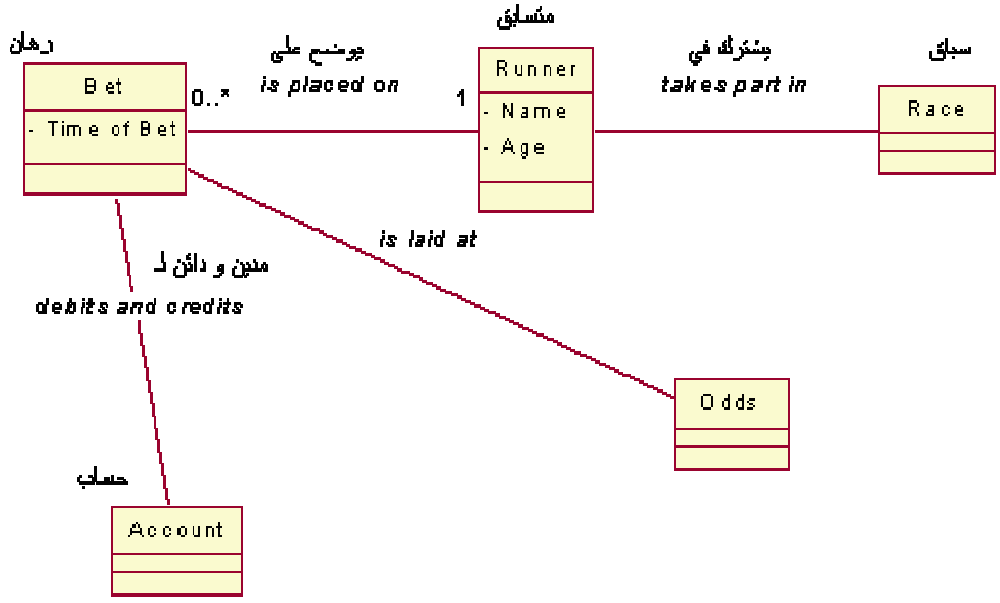
مخططات التعاون: مثال عملي

لنجمع كل هذه المفاهيم مع بعضها ونرى كيف تعمل هذه الترميزات بصورة عملية. لنبني واقعة الاستخدام "place bet" (وضع رهان) باستخدام مخطط التعاون.

هذا المثال ليس كاملا بعد، وتشوبه الكثير من الأسئلة التي لم يتم الإجابة عليها (سوف نعرض قائمة بالمسائل المعلقة في نهاية الفصل). مثلا، كوهلة أولى، يجب أن يصور المثال كيف تم بناء مخططات التعاون. سوف نعيد الزيارة لمسائل التصميم هذه في الفصول اللاحقة.

راجع الفصل السابق و وصف واقعة استخدام بعد اكتمالها ل"Place Bet" (وضع رهان).

لبناء هذا المخطط، نحتاج لبعض الكائنات. من أين نأتي بهذه الكائنات؟ حسنا، سوف يكون علينا بالتأكيد اختراع بعض الكائنات الجديدة كلما تقدمنا/ لكن معظم الكائنات المرشحة يجب أن تأتي مباشرة من صديقنا القديم، النموذج المفاهيمي conceptual الذي قمنا ببنائه في طور التفصيل. هاهنا النموذج المفاهيمي لنظام المراهنة:

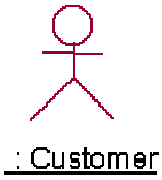


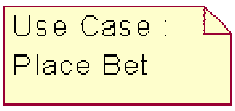
شكل 47: النموذج المفاهيمي لنظام/منظومة المراهنة

أين هي الروابط associations، مثل "is placed on" (يوضع على)، سنقوم على الأغلب باستخدام هذه الروابط لتمرير الرسائل على مخطط التعاون. قد يمكننا تقرير أننا نحتاج (كمثال) لتمرير رسالة بين "account" (حساب) و "race" (سباق). هذا أمر ممكن، لكن طالما أن الرابط لم يتم اكتشافه في المرحلة المفاهيمية، فمن المحتمل أن نخلّ ببعض متطلبات الزبون. إذا حدث هذا، فلابد من مراجعة الزبون!.

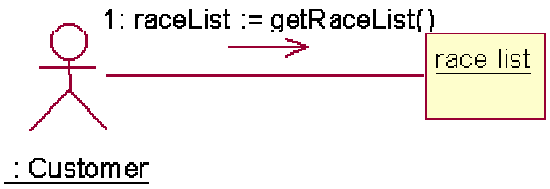
بمساعدة وصف وقائع الاستخدام و مع أخذنا في الاعتبار النموذج المفاهيمي، لنقم ببناء التعاون ل "Place Bet" (وضع رهان).

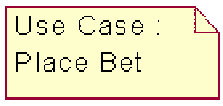
1. قبل كل شيء، نبدأ بإدخال اللاعب actor ، أي الزبون customer. رمز اللاعب ليس جزءا ضروريا في مخطط التعاون في UML، لكنه من المفيد جدا تضمينه في المخطط على أية حال.





2. الآن، و بناء على وصف واقعة الاستخدام، عندما يقوم الزبون باختيار "place bet" وضع الرهان، يتم عرض قائمة بالسباقات. إذا نحتاج لكائن يحتوي على قائمة كاملة بالسباقات ليوم معين، إذا سنقوم بخلق كائن يدعى "Race List"[13] (قائمة سباق). هذا كائن جديد لم يظهر في النموذج المفاهيمي. هذا يسمى design class صنفية تصميم، أي أنها صنفية ظهر الحاجة إليها وقت التصميم.

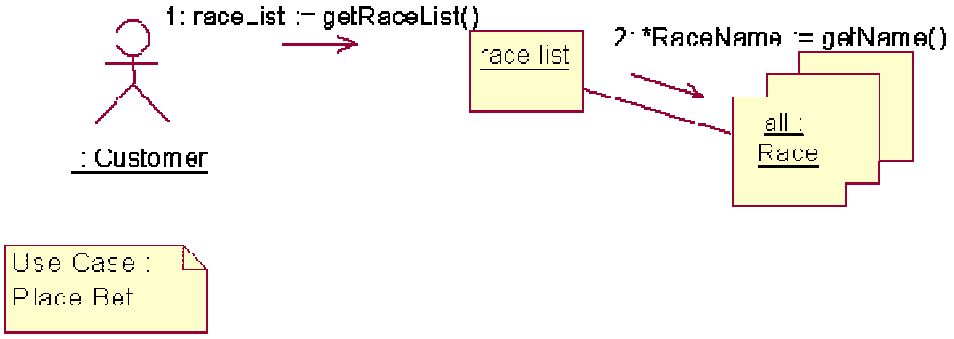




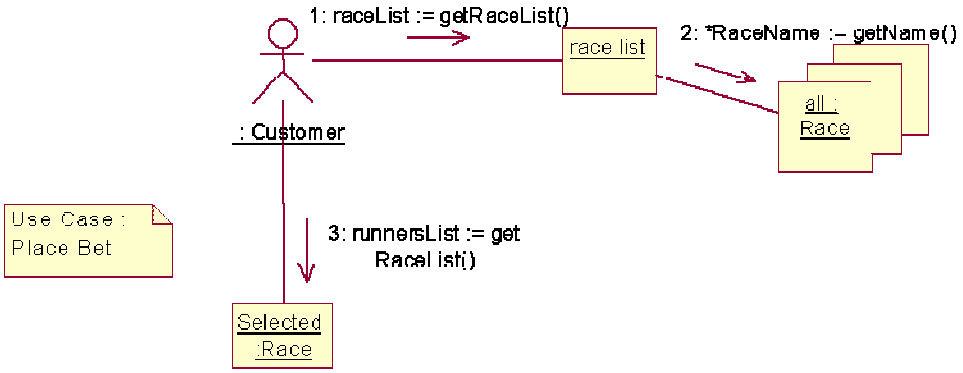
3. إذاً فإن، اللاعب سيبعث برسالة إلى كائن جديد من صنفية"Race List" (قائمة سباق)، الرسالة اسمها "getRaceList" (جلب قائمة سباق). الآن، المهمة التالية مسؤولة عنها قائمة السباق حتى تقوم بتجميع نفسها. و تقوم بهذا من خلال المرور على كل كائنات Race (سباق)، و سؤالهم عن أسمائهم.

كائن Race (سباق) تم أخذه من النموذج المفاهيمي.

[13] هذا سيكون حاوية container، أو مصفوفة array، أو أي شيء مشابه، حسب لغة البرمجة المستهدفة.

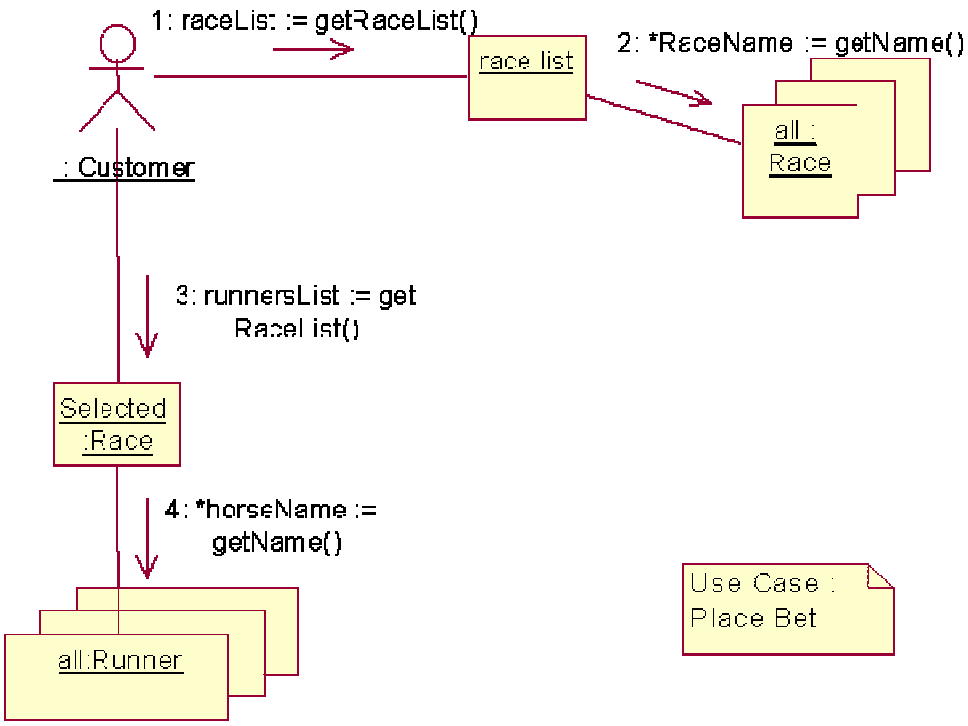


4. بعدها، نفترض أن قائمة السباقات قد تم تحويلها الآن إلى الزبون. الكرة الآن في ملعب الزبون، و حسب ما جاء في مواصفات واقعة الاستخدام (صفحة 72 واقعة الاستخدام بعد اكتمالها)، يقوم المستخدم=اللاعب الآن باختيار السباق من القائمة. يمكننا الافتراض الآن أن السباق تم اختياره. نحتاج الآن إلى قائمة بالمتسابقين على نفس هذا السباق، و من أجل هذا قررنا أن جعل كائن race (سباق) مسؤولا عن الاحتفاظ بقائمة المتسابقين فيه. سوف نرى في الفصول التالية لماذا و كيف أتخذنا مثل هذا القرار.



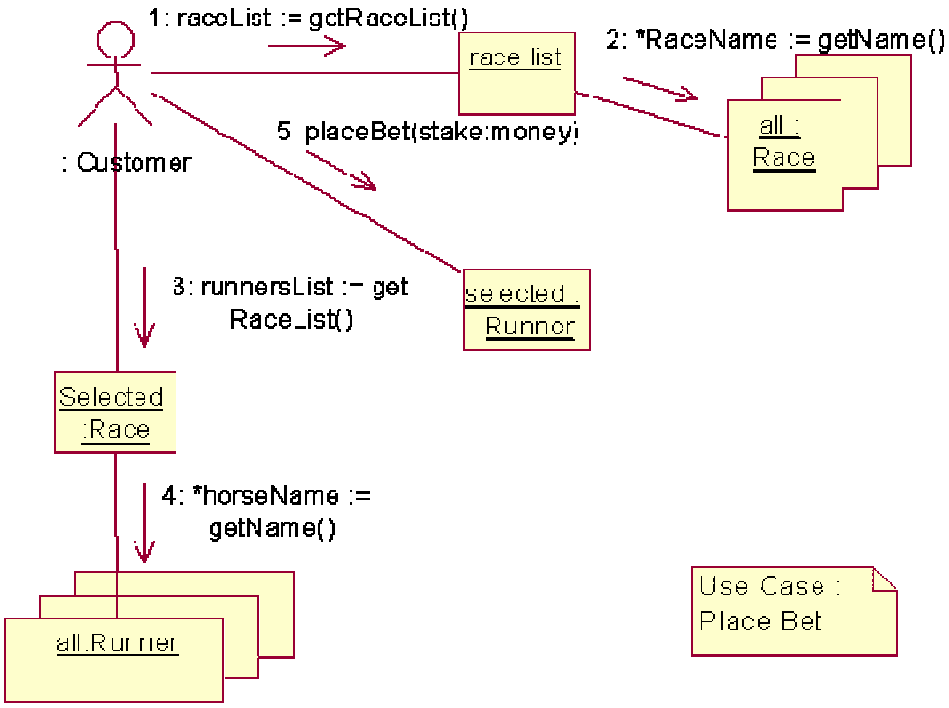
إذا، الرسالة رقم 3 يتم إرسالها للسباق الذي تم اختياره، الرسالة تطلب من السباق تزويده بقائمة من المتسابقين في هذا السباق.

5. كيف لكائن السباق أن يعرف المتسابقين المشتركين فيه. مرة أخرى، سنجعله يقوم بهذا باستخدام التوالي loop، سنجعل كائن السباق يقوم بتجميع قائمة بالمتسابقين. كيفية إنجاز ذلك في النظام الحقيقي ليس بالأمر التافه. واضح، بأننا سنقوم بتخزين المتسابقين في قاعدة للبيانات من نوع ما، لذا فإن جزءا من عملية التصميم الفعلي ستكون إنشاء آلية لاسترجاع تسجيلات الخيل من قاعدة البيانات. بالنسبة للآن عموما، سوف نكتفي بالقول بأن كائن Race (سباق) الذي تم اختياره هو المسؤول عن تجميع قائمة المتسابقين في هذا السباق.



6. قائمة المتسابقين تردّ الآن إلى الزبون. مرة أخرى الكرة في ملعبه، و حسب كراسة مواصفات واقعة الاستخدام، على الزبون الآن أن يختار متسابقا، ثم مبلغ الرهان عليه.

الآن وقد عرفنا المتسابق و الرهان، يمكننا بعث رسالة إلى المتسابق المختار، و نخبره فيها أن الرهان قد وضع عليه.



عند بناء مخطط التعاون هذا، **لم نقم** بتوأمة هذا التصور مع التصوير الفعلي لما يجب أن يكون عليه التوليف code بالضبط. المسائل التالية لا تزال معلقة و غير محلولة:

1. لم نذكر شيئا في المخطط عن كيف يقوم المستخدم بإدخال البيانات في النظام، و كيف لهذه البيانات (مثل قائمة المتسابقين) أن تظهر على الشاشة. و كأن كل هذا سيحدث بفعل سحر داخل اللاعب.
   سوف نرى لاحقا بأن هذا من دواعي التصميم الجيد. نريد من التصميم أن يكون مرنا قدر الإمكان، فإذا أدخلنا فيه تفاصيل تتعلق بواجهة الاستخدام User Interface في هذه المرحلة، نكون قد قيدنا أنفسنا لحل بعينه.
2. كيف لكائن "Race" (سباق) أن يعرف أي المتسابقين هم جزء من هذا السباق؟ واضح، أن هناك عملية من نوع ما لها علاقة بقاعدة بيانات (أو حتى شبكة) تتم هنا. مرة أخرى، نحن لا نريد أن نقيّد تصميمنا في هذه المرحلة، لذلك سنرجئ هذه التفاصيل لما بعد.
3. لماذا جعلنا من كائن "runner" (متسابق) مسؤولا عن تتبع أي رهان تم وضعه عليه؟ لماذا لم نخلق صنفية أخرى، ربما نسميها "bet handler" (مناول سباق) أو "betting system" (نظام مراهنة)؟ هذه المسألة سيتم توضيحها في الفصل التالي.

ما فعلناه هو تقرير مسؤوليات كل صنفية. ما بنيناه كان تأسيسا على النموذج المفاهيمي الذي قمنا بإنشائه في مرحلة التفصيل.

بعض الإرشادات لمخططات التعاون

مع تقدمنا خلال هذه الدروس، سوف نركز على كيفية إنتاج مخططات جيدة. منذ اللحظة، سنأخذ الإرشادات التالية في عين الاعتبار:

1. **اجعل المخطط بسيطا!!!** يبدو شائعا في صنعتنا، أنه ما لم يتضمن المخطط المئات من الصفحات و يوحي بالضخامة و التعقيد، فإن المخطط سيبدو تافها! أفضل قاعدة يتم تطبيقها على مخطط التعاون (و باقي المخططات أيضا في UML) هي أن نجعل المخطط في أبسط صورة ممكنة. إذا أصبح التعاون لواقعة استخدام معقدا أكثر، يتم تجزئته. بإنتاج مخطط منفصل لكل تفاعل بين المستخدم و النظام.
2. **عدم محاولة رصد كل تصور** scenario. كل واقعة استخدام تحتوي على عدد من التصورات المختلفة (التدفق الرئيسي و مختلف البدائل و الاستثناءات). عادة، البدائل ليست بتلك الأهمية و لا يستحق الأمر عناء تضمينها.
   الخطأ الشائع هو الحشو الزائد لكل تصور في المخطط، مما يجعل من المخطط معقدا و صعبا عند التوليف code.
3. **تجنب خلق صنفيات** classes **تحوي أسمائها على كلمات مثل:** "controller"، "handler" ، "manager" **أو** "driver" (متحكم ، مناول، مدير، مسيّر). أو على الأقل، الحذر إذا وجدنا أنفسنا نتعامل مع مثل هذه الأسماء. لماذا؟ لأن هذه الصنفيات توحي بأن تصميمنا ليس كائني المنحى object oriented. مثلا، في واقعة استخدام "Place Bet" (وضع رهان)، كان يمكن لنا أن نخلق صنفية باسم "BetHandler"(مناول رهان) تتعامل مع كل الوظائف الخاصة بالمراهنة. و لكن هذا سيكون حلا بالمنحى للفعل أو السلوك و يرتكز على الأفعال بدل أن يكون كائني المنحى. نحن لدينا بالفعل الكائن "Bet" (رهان) في مخطط التعاون، لذا لما لا نستخدمه، و نعطيه المسؤولية لمناولة الرهانات؟
4. **تجنب الصنفيات العملاقة.** أيضا، إذا انتهينا إلى بناء كائن ضخم يقوم بالكثير من العمل و لا يتعاون كثيرا مع غيره من الكائنات، نكون قد بنينا حلا يرتكز على

الأفعال actions. الحل الجيد المرتكز عل المنحى للكائن يحوي كائنات صغيرة لا تقوم بأعمال كثيرة بمفردها، و لكنها تتعاون مع غيرها من الكائنات لإنجاز أهدافها. سوف نخوض لاحقا في هذا الأمر بتفصيل أكثر.

ملخص

في هذا الفصل، بدأنا ببناء حل برمجي لما لدينا من وقائع استخدام. مخطط التعاون يجعلنا قادرين على توزيع المسؤوليات على الصنفيات التي استخلصناها خلال طور التفصيل.

لمسنا بعض القضايا التي يجب مراعاتها عند توزيع المسؤوليات، مع احتياجنا لتعلم المزيد حول هذا الأمر لاحقا. قمنا بدراسة مثال "Place Bet" (وضع رهان.)

في القسم التالي، سوف نرى كيف يمكننا توسيع النموذج المفاهيمي و تطويره نحو مخطط صنفية حقيقي.

الفصل 13: مخططات صنفيات تصميم

في مرحلة التفصيل، كنا قد قمنا ببناء نموذج مفاهيمي conceptual model. النموذج المفاهيمي يحوي تفاصيل عن مشكلة الزبون، و يركز على مفاهيم الزبون، و خصائص تلك المفاهيم. لم نقم بتحديد سلوك أيا من تلك المفاهيم.

الآن و قد بدأنا ببناء مخططات التعاون، يمكننا تطوير النموذج المفاهيمي، و نبنيه في **مخطط صنفية تصميم** حقيقي **Design Class Diagram**. بعبارة أخرى، مخطط يمكننا أن نؤسس عليه التوليف code النهائي لبرنامجنا.

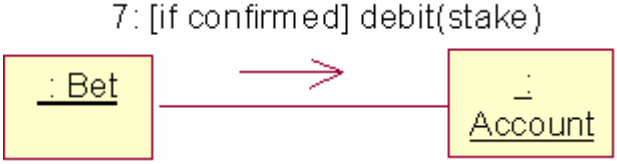
إنتاج مخطط صنفية تصميم هي عملية آلية تماما. في هذا الفصل سوف نفحص مثالا لواقعة استخدام، و كيف يتم تعديل النموذج المفاهيمي بسببها.

مديونية و دائنية الحسابات

في نهاية واقعة الاستخدام "place bet" (وضع رهان)، يبعث الكائن "bet" (رهان) برسالة إلى كائن "Account" (حساب) الخاص بالزبون، يخبره فيها بوجوب تخفيض قيمته. النموذج المفاهيمي التالي كان أساس هذا التصميم:

شكل 48: النموذج المفاهيمي لهذا المثال

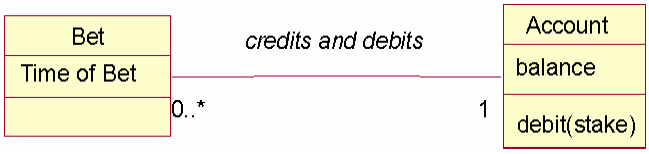
من النموذج المفاهيمي، (جزء من) التعاون التالي تم تطويره:



شكل 49: جزء من تعاون واقعة "وضع رهان"

خطوة 1: إضافة العمليات

من مخطط التعاون، يمكن أن نرى أن صنفية "Account" (حساب) يجب أن يكون لديها السلوك "debit" (مدين / اجعله مدينا). لذا نضيف هذه العملية operation إلى النصف السفلي من رمز الصنفية.

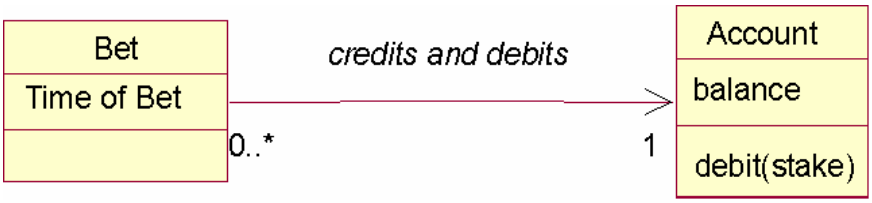


شكل 50: صنفيتان مع عملية تم إضافتها

لنلاحظ أن معظم الناس لا يهتمون بإضافة عملية *create* (خلق)، حتى لا يتم التشويش على المخطط (من المفهوم أن معظم الصنفيات تحتاج إليها).

خطوة 2: إضافة الاتجاهات

تم أيضا إضافة اتجاه الرسائل التي يتم تمريرها عبر الرابط association. في هذه الحالة، الرسالة تم إرسالها من صنفية bet إلى صنفية account، لذا نحن نوجه الرابط من المنادي إلى المستقبل:



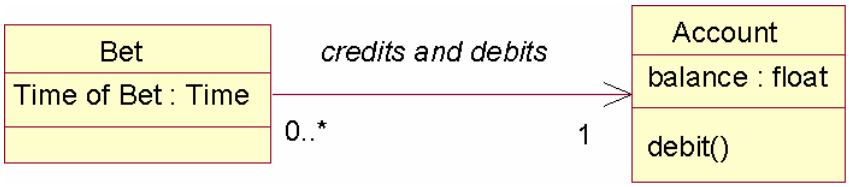
شكل 51: صنفية حساب و قد تم إضافة وجهة الحركة فيها

أحيانا، تظهر حالة يجب فيها تمرير الرسالة في كلا الاتجاهين عبر الرابط. ماذا نفعل عندئذ؟ ترميز UML لهذه الحالة أن يتم ببساطة إزالة رأس السهم من الرابط – **رابط مزدوج الاتجاه bi-direction**.

الكثير من المنمذجين modelers يعتقدون بأن الروابط مزدوجة الاتجاه خاطئة و يجب إزالتها بطريقة ما من النموذج. في الحقيقة، لا يوجد أي خطأ من حيث المبدأ في العلاقات المزدوجة ، لكنها توحي بتصميم سيئ. سوف نستكشف هذه المشكلة في فصل لاحق.

خطوة 3: تحسين السمات

يمكننا أيضا في هذه المرحلة تحديد نوع بيانات datatype السمات attributes. هنا قررنا تخزين الرصيد balance الخاص بالحساب كعدد غير صحيح float.



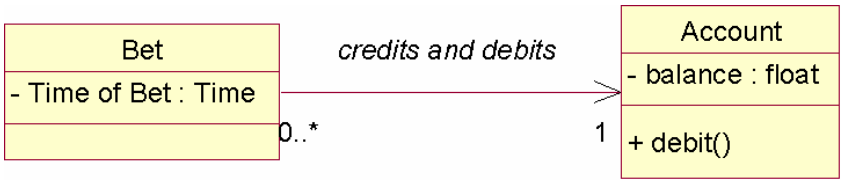
شكل 52: إضافة نوع البيانات

خطوة 4: تحديد المنظورية

أحد المفاهيم الأساسية في المنحى الكائني هو التغليف encapsulation – فكرة أن البيانات التي يحملها الكائن تبقى خاصة به و محجوبة عن العالم خارجه (عن غيره من الكائنات).

يمكننا أن نضع إشارة تبين أي من السمات و العمليات تكون عامة public أو خاصة private في مخطط صنفية UML، من خلال اسباق اسم السمة/العملية بعلامة زائد (بالنسبة للعامة) و علامة ناقص (بالنسبة للخاصة).

كل السمات ستكون خاصة، ما لم يكون هناك سبب جيد وملح (و نادرا ما يكون). عادة، العمليات تكون عامة، ما لم تكن وظائف مساعدة، التي لن تستخدم إلا من قبل العمليات التي تتضمنها الصنفية نفسها.



شكل 53: مخطط الصنفية، مكتملا!

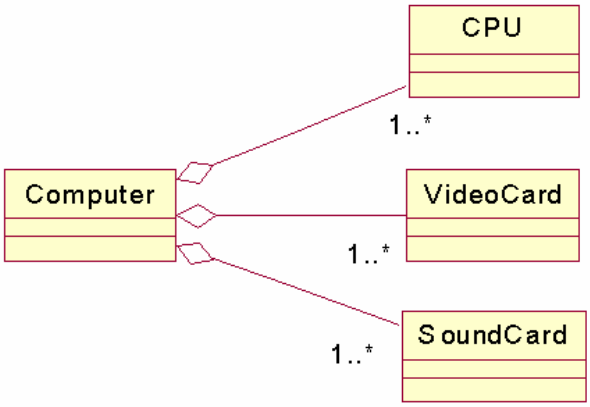
الآن و قد اكتمل مخطط الصنفية، لدينا الآن ما يكفي من المعلومات لإنتاج التوليف code. سوف نختبر عملية التحول إلى توليف في فصل لاحق.

التجمع

إحدى الملامح المهمة في التصميم الكائني المنحى هو مفهوم **التجمع** Aggregation – فكرة أن كائن واحد يمكن بناؤه من (تجمعه من) كائنات أخرى.

مثلا، في نظام حاسوب نمطي ، الحاسوب هو تجمع من وحدة المعالجة و بطاقة الرسوميات ، و بطاقة الصوت وهكذا.

يمكننا التنويه للتجمع في UML باستخدام رمز التجمع – شكل معين عند نهاية وصلة الرابط.



شكل 54: حاسوب مجمع من كائنات أخرى

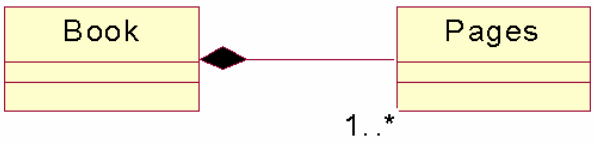
إذا رصدنا تجمعا في نموذجنا المفاهيمي، فقد يكون من الأوضح أن نقوم بتبيان هذه الحقيقة صراحة، باستخدام رمز التجمع.

التكون

مفهوم آخر شبيه جدا بالتجمع هو **التكون** composition أو التركيب، التكون هو أقوى من التجمع، بمفهوم أن العلاقة تفرض أن لا وجود للكل بدون الأجزاء.

في مثال التجمع أعلاه، إذا أزحنا بطاقة الصوت، فإن الحاسوب سيظل حاسوبا. بالمقابل، الكتاب لا يكون كتابا بدون صفحاته، لذلك نقول أن الكتاب مكون من صفحات.

طريقة التنويه لهذا شبيهة بالتجمع، باستثناء أن المعين هذه المرة يكون ممتلئا، كالتالي:



شكل 55: الكتاب مكون من صفحة أو أكثر

إيجاد التجمع و التكون

إيجاد هذه العلاقات على مخطط الصنفية أمر مفيد، ولكن ليس حاسما في نجاح تصميمنا. بعض ممارسي UML يذهبون لأبعد من ذلك، و يدعون أن هذه العلاقات هي إسهاب زائد، و يجب إزالتها (التجمع و التكون يمكن نمذجتهما كرابط له اسم مثل "يتكون من").

عموما، طالما أن التجمع أحد المفاهيم الرئيسية في المنحى للكائن، فالأمر بالتأكيد يستحق الإشارة إلى وجوده صراحة.

ملخص

في هذا الفصل، رأينا كيفية تطوير نموذج الصنفية، بناء على عملنا في التعاونات. التحول من النموذج المفاهيمي إلى نموذج لصنفية تصميم أمر في غاية السهولة و الآلية، و لا يجب أن يسبب الكثير من العناء.

الفصل 14: أنماط توزيع المسؤولية

في هذا القسم، سوف نبتعد قليلا من عمليات التنشئة و التطوير، و ننظر عن قرب في المهارات التي تسهم في البناء الجيد لتصاميم كائنية المنحى.

بعض النصائح التي سيتم تقديمها في هذا الفصل قد تبدو بديهية و معروفة. في الواقع، إن انتهاك هذه الإرشادات البسيطة هي السبب في معظم المشاكل في التصميم بالمنحى للكائن.

ما هو النمط؟

النمط pattern هو حل عام جدا و مستخدم بشكل جيد لمشكلة دائمة الحدوث. بدأت نزعة الأنماط في الظهور في منتديات النقاش على الانترنت، إلا أن شعبيتها كانت من خلال كتاب تدريبي: "أنماط التصميم"Design Patterns (مرجع 6)، كتبه أربعة يُعرفون باسم "عصابة الأربعة" "Gang of Four".

لتسهيل عملية التواصل، كل نمط تصميم لديه اسم سهل التذكر (مثل Factory المصنع، Flywheel عجلة التوازن، Observer المراقب)، و توجد على الأقل حفنة من أنماط التصميم هذه التي يجب على كل مصمم يحترم نفسه أن يكون ملما بها.

سوف نرى لاحقا بعضا من أنماط التصميم التي قدمتها "عصابة الأربعة"، لكننا سوف ندرس أنماط GRASP أولا.

GRASP هي اختصار ل: "General Responsibility Assignment Software Patterns" الأنماط البرمجية العامة لتوزيع المسؤولية ، و هي تساعدنا في التأكيد على أننا قمنا بتعيين سلوكيات الصنفيات بأقوم طريقة ممكنة.

التوزيع الحكيم لمسؤولية للسلوك و التصرفات على الصنفيات المعنية يؤدي إلى أنظمة تتميز بأنها:

- أسهل للفهم
- قابلية أكثر سهولة للتوسع (إستيساعي)
- قابلة لإعادة الاستخدام
- أكثر تماسكا

الأنماط تسمى: Expert الخبير، Creator المنشئ، High Cohesion الاتساق العالي، Low Coupling الاقتران المنخفض، و Controller المتحكم. لنرى كل واحد من هـذه الأنماط:

(GRASP 1): الخبير Expert

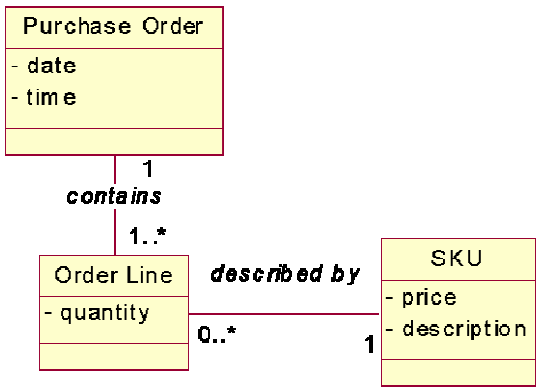
هذا هو، وكما تعنيها الكلمة، نمط بسيط جدا. إلا أنه هو أيضا أكثر الأنماط التـي يـتم تجاهلها. هذا النمط يجب أن يكون أمام أعيننا كلما قمنا ببناء مخططات التعاون أو أنشأنا مخططات صنفيات التصميم.

نمط الخبير ينص على أن يتم: "تخصيص المسؤولية للصنفية التي تملـك المعلومـات اللازمة للقيام بهذه المسؤولية"

لنلقي نظرة على مثال مبسط. لدينا ثلاث صنفيات، واحدة تمثل أمر شراء ، و أخـرى تمثل سطر لأمر شراء، و أخيرا، واحدة تمثل بند المخزون SKU[*].

فيما يلي جزء مأخوذ من النموذج المفاهيمي:

[*] Stock Keeping Unit



شكل 56: جزء مأخوذ من النموذج المفاهيمي

الآن، لنتخيل أننا نبني تعاونا لإحدى وقائع الاستخدام. واقعة الاستخدام هذه يجب أن تعرض على المستخدم القيمة الإجمالية لأمر الشراء الذي تم اختياره. أية صنفية ستضم السلوك المسمى "calculate_total()" (حساب_الإجمالي)؟

نمط الخبير يخبرنا بأن الصنفية الوحيدة التي يجب أن تتعامل مع التكلفة الإجمالية لأوامر الشراء هي صنفية أمر الشراء نفسها – لأن هذه الصنفية هي الخبيرة بكل ما يتعلق بأوامر الشراء.

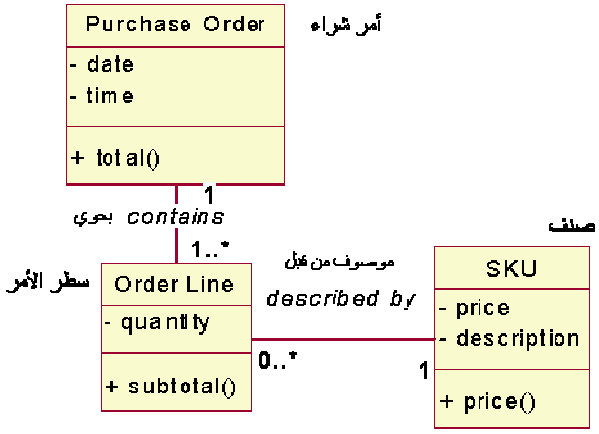
لذا، نقوم بتخصيص منهاج method: "calcuate_total()" لصنفية أمر الشراء Purchase Order.

الآن، لحساب إجمالي أمر الشراء، يحتاج أمر الشراء لمعرفة قيمة كل سطر من أسطر الطلبية أو الأمر.

سيكون تصميما ضعيفا لو جعلنا أمر الشراء يرى محتويات كل سطر من سطور الطلبية (عبر وظائف نفاذة)، ثم يقوم بحساب الإجمالي، في هذا انتهاك لنمط الخبير، لأن الصنفية الوحيدة التي يجب أن يسمح لها بحساب إجمالي سطر الطلبية هي صنفية سطر الطلبية نفسها.

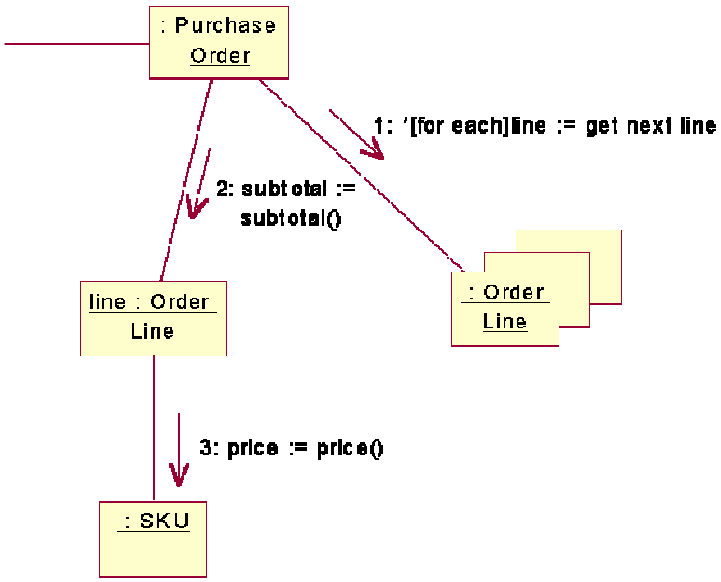
لذا نخصص سلوكا آخرا لصنفية سطر الأمر، تسمى subtotal(). هذه المنهاج يرجع لنا مجموع تكلفة سطر الأمر. للحصول على هذا السلوك، فإن صنفية سطر الأمر تحتاج

لمعرفة تكلفة الصنف من خلال منهاج آخر (هذه المرة، نفاذ) يسمى ()price السعر في صنفية الصنف.



شكل 57: سلوكيات تم تحديدها بإتباع نمط الخبير

هذا يؤدي إلى مخطط التعاون التالي:



شكل 58: ثلاث صنفيات من الكائنات تتعاون لتقديم إجمالي التكلفة لأمر الشراء

(GRASP 2): المنشئ Creator

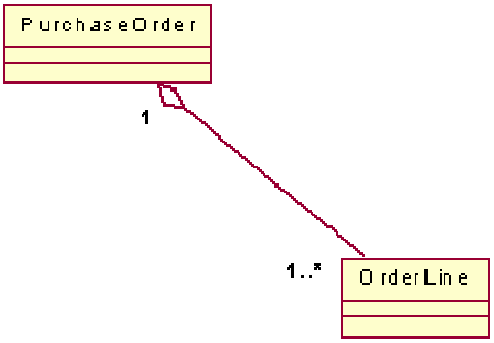
نمط المنشئ هو استعمال أكثر تحديدا لنمط الخبير. هو يطرح السؤال التالي: "من يجب أن يكون مسئولا عن إنشاء تمثلات/ تجسدات instances جديدة لصنفية معينة؟"

الإجابة هي أن:

الصنفية (أ) يجب أن تكون مسئولة عن إنشاء كائنات من الصنفية (ب) إذا وجدت حالة أو أكثر من الحالات التالية:

- تجمع كائنات (ب)
- تحوي كائنات (ب)
- تتابع التجسدات لكائنات (ب)
- تتعامل عن قرب مع كائنات (ب)
- تملك البيانات التمهيدية التي ستمرر إلى (ب) عند خلقه (بالتالي فإن (أ) خبير فيما يخص خلق (ب))

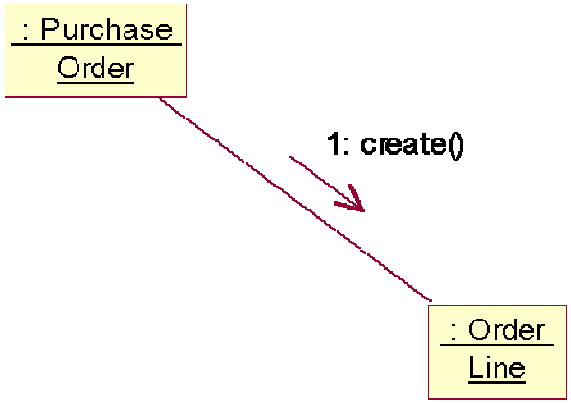
بالعودة إلى مثال أمر الشراء، لنفترض أنه تم إنشاء أمر شراء جديد. أية صنفية يجب أن تكون مسئولة عن إنشاء أسطر أمر الشراء ؟



شكل 59: أية صنفية يجب أن تخلق أمر الشراء

الحل هو أن أمر الشراء يحوي أسطر أمر الشراء، لذلك صنفية أمر الشراء (فقط هذه الصنفية) يجب أن تكون مسئولة عن إنشاء أسطر الأمر.

فيما يلي مخطط التعاون لهذه الحالة:

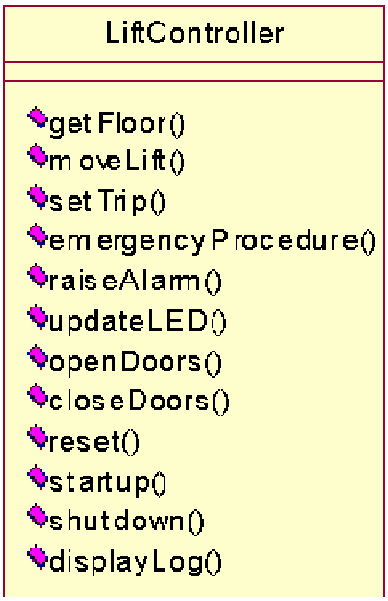


شكل 60: أمر الشراء يخلق أسطر الأمر

(GRASP 3): اتساق عال High Cohesion

من المهم جدا التأكيد على أن تكون مسئوليات كل صنفية أكثر تركيزا و ترابطا. في أي تصميم كائني جيد ، كل صنفية يجب أن لا تكون مزحومة بالأعمال و المسؤوليات غير المترابطة. كمؤشر لتصميم كائني جيد يجب أن تكون لكل صنفية عدد صغير من المنهاجيات methods.

في المثال التالي، عند تصميم نظام إدارة مصعد، تم تصميم الصنفية التالية:



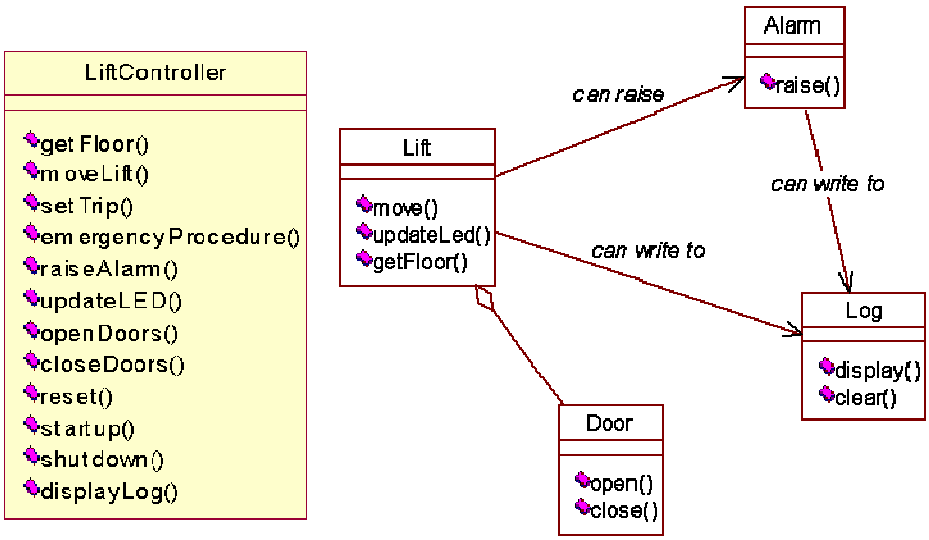
شكل 61: صنفية من نظام إدارة مصعد

هل هذا تصميم جيد، من الواضح أن الصنفية تقوم بالعديد من الأعمال- إطلاق الإنذارات، تشغيل/ إقفال، تحريك المصعد و تحديث عرض المؤشر. هذا تصميم سيء لأن الصنفية غير متسقة.

سيكون من الصعب صيانة هذه الصنفية، ليس من الواضح ما الذي يفترض بهذه الصنفية عمله.

القاعدة التي يجب إتباعها عند بناء الصنفيات هي أن كل صنفية يجب أن تمثل فقط فكرة رئيسية واحدة – بعبارة أخرى، الصنفية يجب أن تمثل "شيء" واحد من عالم الواقع.

متحكم المصعد "Lift controller" يحاول نمذجة ثلاث أفكار رئيسية مختلفة على الأقل – جرس الإنذار و أبواب المصعد و سجل الأعطال. لذلك فإن التصميم الجيد يدعو لأن نجزئ "متحكم المصعد" إلى عدة صنفيات.



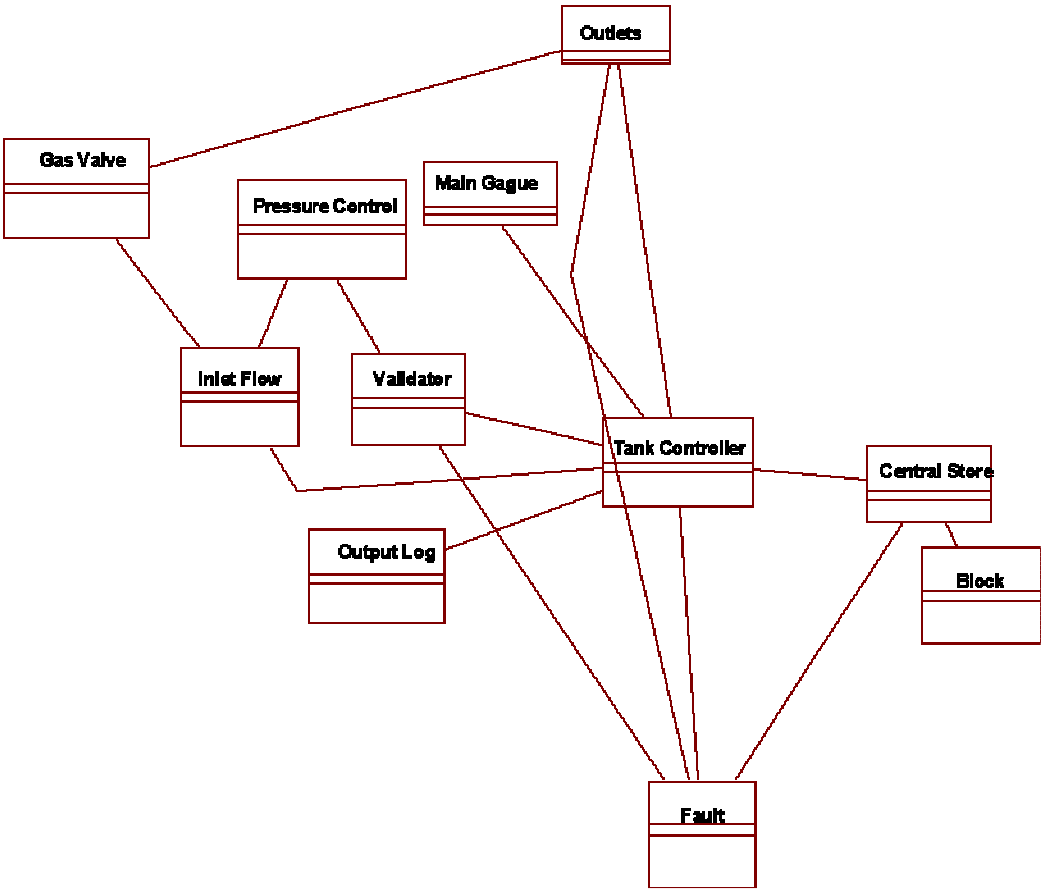
شكل 62: صنفية متحكم المصعد بعد نمذجتها في أربعة صنفيات منفصلة و لكن أكثر اتساقا

(GRASP 4): اقتران منخفض Low Coupling

يقاس الاقتران بمدى اعتماد صنفية ما على صنفيات أخرى. الاقتران الشديد يؤدي إلى صعوبة عند التغيير أو الصيانة – أي تعديل يتم على صنفية واحدة قد يؤدي إلى تموجات من التغيير تمتد عبر النظام.

مخطط التعاون يقدم وسيلة ممتازة لرصد مواقع الاقتران، الدرجات العالية من الاقتران يمكن أيضا ملاحظتها من خلال مخططات الصنفيات.

فيما يلي مثالا مأخوذا من مخطط صنفيات و يظهر فيه واضحا علامات شدة الاقتران.



شكل 63: شدة الاقتران في مخطط صنفيات

هل كل الروابط في الشكل 63 فعلا مهمة؟ مصمم هذا النظام يجب أن يسأل بعض الأسئلة الجادة حول هذا التصميم. مثلا:

- لماذا صنفية Fault (عطل) مرتبطة مباشرة بصنفية Outlets (مقابس) بينما يوجد رابط غير مباشر عبر صنفية Tank Controller (حجرة التحكم)؟ ربما تم وضع هذا الرابط لأغراض سرعة الأداء، و هو شيء مقبول، لكن الأرجح أن هذا الرابط ناتج عن إهمال عند نمذجة التعاون.
- لماذا حجرة التحكم لديها هذا العدد الكبير من الروابط. قد تكون هذه الصنفية غير متسقة incohesive و تقوم بعدد كبير من الأعمال.

إتباع النموذج المفاهيمي هو طريقة ممتازة لتخفيض معدل الاقتران. يتم فقط بعث رسالة من صنفية إلى أخرى إذا وجد رابط تم تحديده في مرحلة النمذجة المفاهيمية. بهذه

الطريقة، نلزم أنفسنا بأن نسمح بالاقتران فقط في الحالة التي يؤكد فيها الزبون أن المفاهيم هي أيضا مرتبطة في الحياة الواقعية.

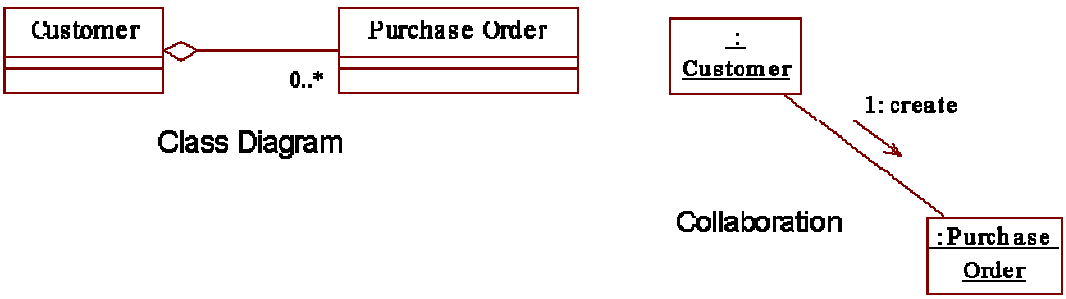
إذا وجدنا، في مرحلة التعاون، بأننا نرغب ببعث رسالة من صنفية لأخرى غير مرتبطتين في النموذج المفاهيمي، عندها نسأل أنفسنا سؤالا مهما حول ما إذا كان الاقتران موجودا في الواقع العملي أم لا. استشارة الزبون قد تساعد هنا – فقد يكون الرابط غير واضح عندما قمنا ببناء النموذج المفاهيمي.

إذا القاعدة هنا: **المحافظة على درجة الاقتران لأدناها – النموذج المفاهيمي مصدر ممتاز ينصحنا بدرجة الحد الأدنى المطلوبة. يمكن رفع مستوى الاقتران، بشرط أن نكون قد فكرنا بحرص شديد حول النتائج المترتبة!**

مثال عملي

لنتفحص نظام أمر الشراء. سوف نرى أنه في النموذج المفاهيمي قد تم تحديد أن الزبائن يمتلكون/أو تتبعهم أوامر الشراء (لأنهم هم من يصدرونها):

في واقعة استخدام "خلق أمر شراء" ، ماهي الصنفية الي ستكون مسئولة عن خلق أوامر شراء جديدة؟ بإتباع نمط المنشئ Creator Pattern فإن صنفية Customer الزبون هي التي يجب أن تكون مسئولة:

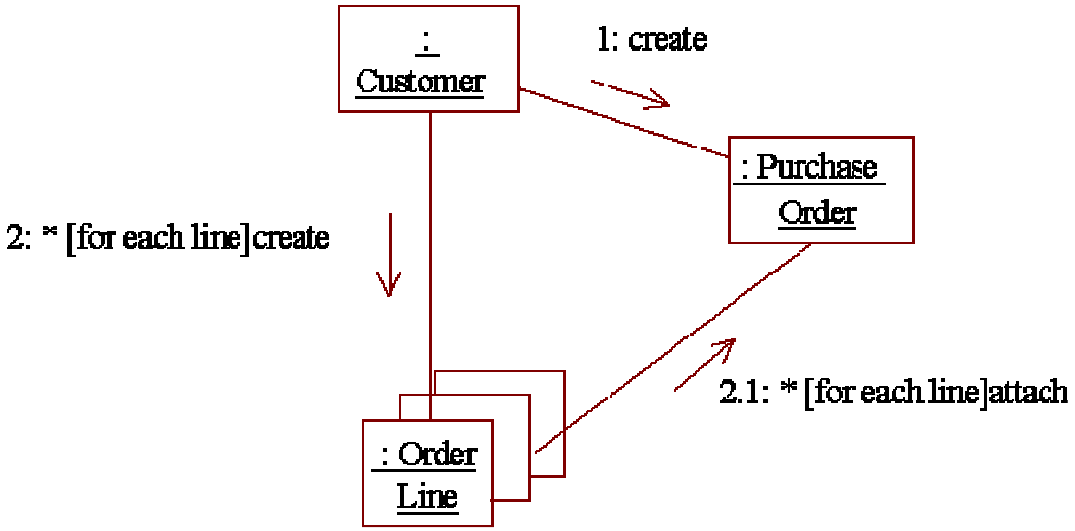


شكل 64: مخطط صنفية و تعاون لواقعة إنشاء أمر شراء

إذا، قمنا الآن بجعل الزبون و أمر الشراء مقترنين ببعض. هذا جيد، لأنهما مقترنين حتى في الواقع الفعلي.

بعدها، حالما يتم خلق أمر الشراء، تحتاج واقعة الاستخدام إلى إضافة أسطر لأمر الشراء. من يجب أن يكون مسئولا عن إضافة الأسطر؟

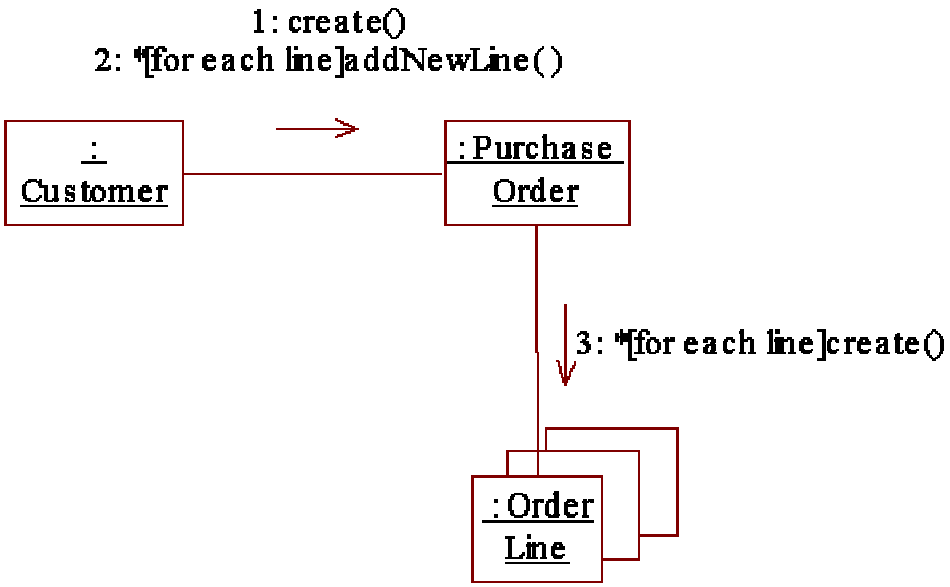
إحدى الإجابات هي أن نجعل صنفية الزبون تقوم بهذا العمل (فهي تملك البيانات التمهيدية اللازمة – كم عدد الأسطر، ما هي الأصناف، الكميات المطلوبة، و هكذا).



شكل 65: محاولة أولى. كائنات الزبون تخلق الأسطر لأنها تملك البيانات التمهيدية

هذه المحاولة أنتجت اقترانا زائفا. أصبح لدينا الآن الصنفيات الثلاث كلها تعتمد كل واحدة منها على الأخرى، في حين لو جعلنا أمر الشراء مسؤولا عن إنشاء الأسطر، سنصل لحالة يكون فيها الزبائن لا علم لديهم بأسطر الأمر.

مادام الزبون يمتلك صنفية أمر الشراء، يمكنه أن يمرر البيانات التمهيدية مباشرة لأمر الشراء من أجل أن تتولى الأخيرة إنشاء أسطر الأمر.



شكل 66: إضافة أسطر الأمر، مع اقتران أقل

إذا الآن، إذا تعرض بناء صنفية سطر الأمر للتغيير لأي سبب، فالصنفية الوحيد التي ستتأثر ستكون صنفية أمر الشراء. كل الاقترانات الموجودة في هذا التصميم كانت اقترانات تم التعرف عليها في المرحلة المفاهيمية. الزبائن يمتلكون أوامر الشراء؛ أوامر الشراء تمتلك الأسطر. و بهذا تكون الاقترانات منطقية و يمكن تبرير وجودها.

قانون ديمتر Demeter

و يعرف أيضا بالعبارة :"*لا تتحدث مع الغرباء*"، هذا القانون وسيلة فعالة لمحاربة الاقتران. ينص القانون على أن أي منهاج method لأي كائن يجب أن ينادي فقط المنهاجات التي تتبع ل:

- الكائن نفسه
- معطيات تم تمريرها داخل المنهاج
- أي كائن تقوم بخلقه
- أية كائنات لمكونات تحتفظ بها بصورة مباشرة.

اجعل الكائنات أكثر "خجلا" تنخفض نسبة الاقترانات!

كلمة أخيرة عن الاقتران

بعض القضايا الإضافية يجب أخذها في الاعتبار:

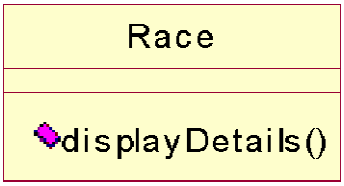
- أبدا لا تجعل سمة في الصنفية تكون عامة public – السمة العامة تسبب في سوء استعمال الصنفية (يستثنى من ذلك الثوابت constants التابعة للصنفية)
- توفير فقط منهاجات تحصيل/تسوية get/set methods عندما تدعو الحاجة
- توفير أقل ما يمكن من الواجهات interface العامة (المنهاج لا يكون عاما إلا إذا دعت الحاجة إلى استخدامه من قبل العالم الخارجي)
- لا تجعل البيانات تسري بين النظام – التقليل من البيانات التي تمرر كمعطيات
- لا تفكر في التقليل من الاقتران كعزل- تذكر الاتساق العالي و نمط الخبير! النظام عديم الاقتران كلية سوف ينتفخ بصنفيات تقوم بحجم كبير جدا من العمل. هذا غالبا ما يعبر عن نفسه كنظام ذو عدد قليل من "الكائنات النشطة" التي لا تتواصل.

(GRASP 5): الموجه Controller

نمط GRASP الأخير الذي سنتعرف عليه في هذا القسم هو نمط الموجه. لنعد إلى نظام المراهنة و واقعة استخدام Place Bet (وضع رهان). عندما قمنا ببناء التعاونات لواقعة الاستخدام هذه (راجع الشكل النهائي لمخطط التعاون في نهاية الفصل 12)، رأينا أننا لم نهتم بكيف يتم استقبال المدخلات من المستخدم، و كيف يتم عرض النتائج له.

لذلك، و كمثال، نحتاج لعرض تفاصيل السباق للمستخدم. أية صنفية ستكون مسئولة لتلبية هذه المتطلبات؟

لو استرشدنا بنمط الخبيرExpert سيقترح علينا بأنه ما دامت التفاصيل ذات علاقة بالسباق إذا فصنفية Race سباق يجب أن تكون هي الخبير لعرض التفاصيل المتعلقة بها.



شكل 67: هل على صنفية السباق أن تكون المسؤولة عن عرض تفاصيلها

قد يبدو الأمر مقبولا في البداية، ولكن في الواقع، نحن ننتهك نمط الخبير! فعلا صنفية السباق هي الخبيرة في كل ما يتعلق بالسباقات، لكن يجب أن لا تكون الخبير في أمور مثل واجهات الاستخدام الرسومية!

عموما، إضافة معلومات حول واجهات الاستخدام (و قواعد البيانات، أو أي كائن مادي آخر) داخل صنفياتنا يعد تصميما ضعيفا – تخيل لو لدينا خمس مائة صنفية في نظامنا، و معظمها تقرأ من الشاشة و تكتب فيها، و ربما تكون شاشة نصية. ماذا يحدث إذا تغيرت المتطلبات، و نريد أن نستبدل الشاشة النصية بواجهة استخدام رسومية نوافذية؟ سيكون علينا ملاحقة كل الصنفيات و القيام بعمل مضني لمعالجة ما يلزم تغييره.

سيكون أفضل بكثير الاحتفاظ بنقاء الصنفيات كما هي في النموذج المفاهيمي (سنسميها "صنفيات الأعمال" Business Classes)، و نزيل منها كل إشارة لواجهة استخدام أو قاعدة بيانات أو ما شابه.

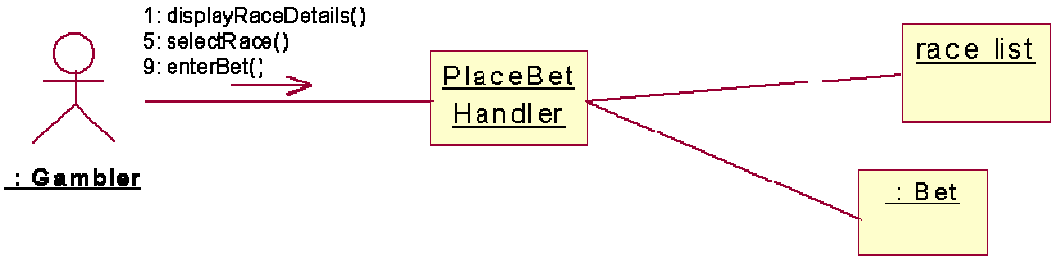
لكن كيف لصنفية السباق أن تعرض تفاصيل السباق؟

الحل – نمط الموجه

حل محتمل واحد هو استخدام نمط الموجه. يمكننا ادخال صنفية جديدة، و نجعلها بين اللاعب actor و صنفيات الأعمال.

اسم صنفية الموجه هذه عادة ما يكون: مناول+ <اسم واقعة الاستخدام> Handler<UseCaseName>. لذلك في حالتنا، نحتاج لمناول يسمى: "PlaceBetHandler" مناول_وضع_الرهان.

المناول يستقبل الأوامر من المستخدم، ثم يقرر أية صنفية يوجه لها الرسائل messages. المناول هو الصنفية الوحيدة التي يسمح لها بالقراءة من الشاشة و الكتابة فيها.



شكل 68: موجه لواقعة استخدام تم إضافته للتصميم

في حالة طُلب منا استبدال واجهة الاستخدام، فالصنفيات الوحيدة التي علينا تعديلها هي صنفيات التوجيه.

ملخص

في هذا الفصل، استكشفنا أنماط "GRASP". التطبيق الدقيق لأنماط GRASP الخمسة يؤدي إلى تصميم كائني المنحى أكثر وضوحا، و أكثر قابلية للتحوير، و أشد تماسك.

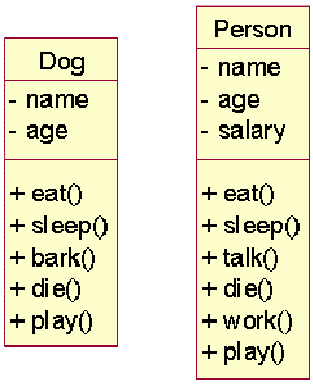
الفصل 15: الوراثة

إحدى أهم المفاهيم المعروفة و أكثرها قوة التي يتضمنها المنحى للكائن هو الوراثة Inheritance. في هذا الفصل، سنلقي نظرة أساسية على الوراثة، و متى يتم تطبيقها (و متى لا يتم تطبيقها)، و سننظر بخاصة إلى ترميز ULM للتعبير عن الوراثة.

الوراثة – الأساسيات

غالبا، ما تتشارك بضعة صنفيات في خصائص متشابهة. في المنحى للكائن، يمكننا أن نضم هذه الخصائص المشتركة في صنفية واحدة. عندها يمكننا التوريث من هذه الصنفية، و بناء صنفيات جديدة منها. عندما نقوم بالتوريث من صنفية، نكون أحرار لأن نضيف وظائف و سمات جديدة كلما دعت الحاجة.

فيما يلي مثال على ذلك. لنقل أننا نقوم بنمذجة سمات و سلوك الكلاب و البشر*. هذا ما ستكون عليه صنفياتنا:



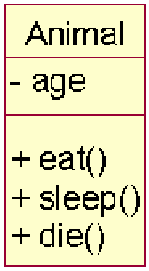
شكل 69: نمذجة الكلاب و البشر

* المترجم يعتذر مرة أخرى.

برغم اختلاف الصنفيتان، إلا أن لديهما الكثير من القواسم المشتركة. كل شخص لديه اسم، كذلك بالنسبة لكل كلب[14]. أيضا بالنسبة للعمر. بعض السلوكيات مشتركة أيضا، مثل: الأكل و النوم و الموت. إلا أن الكلام تتميز به صنفية الشخص.

إذا قررنا إضافة صنفية جديدة لتصميمنا، مثل ببغاء "Parrot" ، سيكون الأمر مضجرا و نحن نضيف كل السمات و المناهج المشتركة لصنفية ببغاء مرة أخرى. بدلا من ذلك، يمكننا ضم كل سلوك أو خاصية مشتركة في صنفية جديدة.

إذا أخذنا الخاصية "Age" عمر، و السلوكيات "Eat" أكل، "Sleep" نوم، "Die" الموت، سيكون لدينا سمات و سلوك يفترض أن تكون مشتركة بين كل الحيوانات. لذلك، نستطيع بناء صنفية جديدة تسمى "Animal"حيوان.



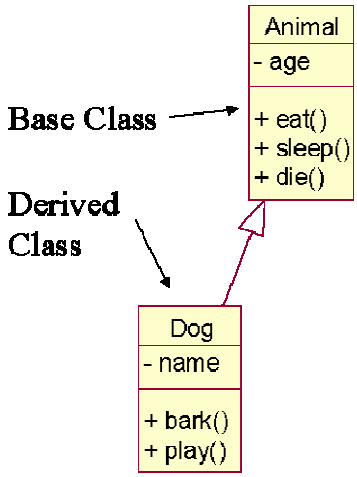
شكل 70: صنفية "حيوان" أكثر عمومية

الآن، عندما نريد إنشاء صنفية "كلب"، بدلا من البدء من لاشيء، نقوم بالتوريث من صنفية "حيوان" و نضيف عليها فقط السمات و المناهج التي نحتاجها للصنفية الجديدة.

المخطط التالي يبين هذا بصيغة UML. لنلاحظ أن في صنفية Dog "كلب" لم نقم بتضمين المناهج و السمات القديمة – لأنها موجودة ضمنا فيها.

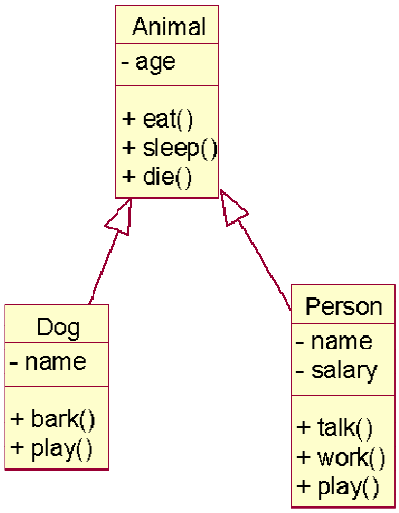
الصنفية الذي بدأنا بها تدعى **base class** الصنفية الأساس (أحيانا تدعى Superclass صنفية عُليا). الصنفيات التي أنشأناها منها تسمى derived class صنفية مشتقة (أحيانا تدعى Subclass صنفية فرعية).

[14] لنفترض أنها كلاب أليفة!



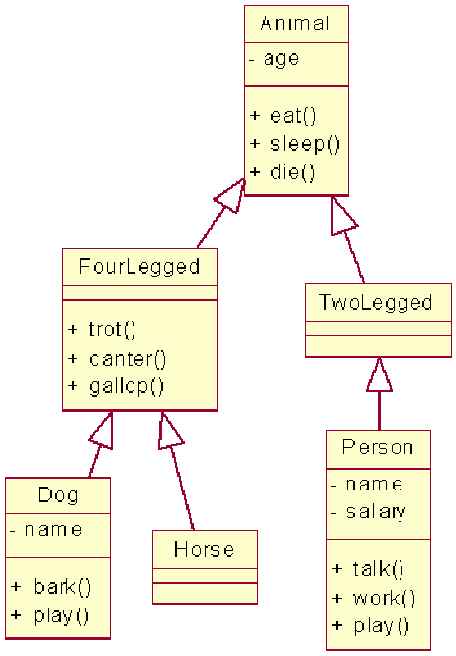
شكل 71: إنشاء صنفية "كلب" من صنفية "حيوان"

نستطيع الاستمرار بإنشاء صنفيات جديدة مشتقة من نفس الصنفية الأساس. لإنشاء صنفية شخص، تقوم بالتوريث كالتالي:



شكل 72: شخص مشتق من حيوان

نستطيع الاستمرار في عملية التوريث من الصنفيات لنشكل شجرة (هرمية) صنفيات class hierarchy.



شكل 73: هرمية صنفيات

الوراثة هي استخدام لصندوق أبيض

الخطأ الشائع في تصاميم المنحى للكائن هو المبالغة في استخدام الوراثة. فهو يؤدي إلى مشاكل في الصيانة. فالصنفيات المشتقة مرتبطة بعلاقة وثيقة مع الصنفية الأساس و أية تغييرات في الصنفية الأساس سينتج عنها تغييرات في الصنفيات المشتقة منها.

أيضا، عندما نستخدم صنفية مشتقة، يجب أن نعرف بالضبط العمل الذي تقوم به الصنفيات الأعلى؛ مما يعني البحث عبر هيكل هرمي ضخم.

هذه المشكلة تعرف بتوالد الصنفيات *proliferation of classes*.

إحدى أسباب هذا التوالد هو استخدام الوراثة في الوقت الذي لا يجب فيه استخدامها. القاعدة التي ينبغي إتباعها هي التالي:

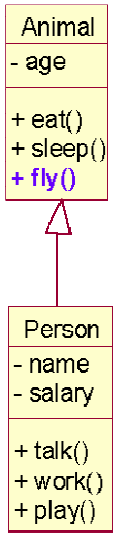
الوراثة يجب استخدامها فقط كوسيلة للتعميم.

بتعبير آخر، نستخدم الوراثة فقط عندما تكون الصنفيات المشتقة هي من نوع أكثر اختصاص من الصنفية الأساس. توجد قاعدتان نستعين بهما:

- قاعدة هي_نوع_من
- قاعدة 100%

قاعدة 100%

كل التعريفات الخاصة بالصنفية الأساس يجب أن تنطبق على الصنفيات المشتقة. إذا لم تنطبق هذه القاعدة، فهذا يعني أنه عند التوريث، لم يتم إنشاء نسخة أكثر اختصاص من الصنفية الأساس. هنا مثال على ذلك:



شكل 74: وراثة ضعيفة

في الشكل 73، منهاج ()fly يطير يجب ألا يكون جزءا من صنفية Animal حيوان. فليس كل الحيوانات تطير، لذلك فإن الصنفية المشتقة، Person شخص، لديه منهاج غريب عنه مرتبط به.

تجاهل قاعدة 100% هي أسهل طريقة لخلق مشاكل الصيانة.

الاحلالية

في المثال السابق، لماذا لم نقم ببساطة بإزالة عملية "fly" يطير من صنفية person شخص؟ هذا سوف يحل المشكلة.

المنهاجات لا يمكن إزالتها في هرمية مورثة. هذه القاعدة تم فرضها لضمان حماية مبدأ الإحلالية Substitutability Principle. سوف ننظر في هذا بقليل من التفصيل قريبا.

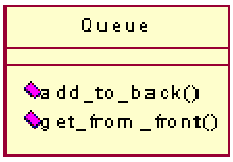
قاعدة هي_نوع_من

قاعدة هي_نوع_من طريقة بسيطة لاختبار مدى صحة هرمية الوراثة لدينا. العبارة : <الصنفية المشتقة> هي نوع من <الصنفية الأساس> يجب أن يكون لها معنى. مثلا "الكلب هو نوع من الحيوان" أو "الكلب هو حيوان".

غالبا، الصنفيات المشتقة من صنفيات أساس لا تكون فيها هذه القاعدة مطبقة، مشاكل الصيانة تكون أكثر احتمالا، فيما يلي مثال عملي:

مثال – إعادة استخدام الصفوف عن طريق الوراثة

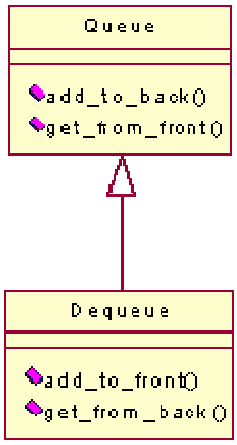
لنفترض إننا بنينا (كتوليف) صنفية لطابور queue. صنفية الطابور هذه تسمح لنا بإضافة بنود في نهاية الطابور و تزيل البنود من أمام الطابور.



شكل 75: صنفية طابور

بعد فترة، قررنا بأننا نحتاج لبناء نوع جديد من الطابور – نوع خاص من الطابور يسمى "Dequeue" طابور_د. هذا النوع من الطابور يعطينا نفس العمليات التي للطابور لكن مع سلوكيات إضافية تسمح بإمكانية إضافة البنود لمقدمة الطابور و إزالتها من خلفية الطابور. طابور من نوع مزدوج الحركة.

للاستفادة من العمل الذي أنجزناه فعلا في صنفية الطابور، يمكننا إعادة استخدام العمل والتوريث من هذه الصنفية.



شكل 76: بناء صنفية جديدة عن طريق التوريث

هل هذا التوريث يجتاز اختبارات قاعدة "هو_نوع_من" و قاعدة "100%" ؟

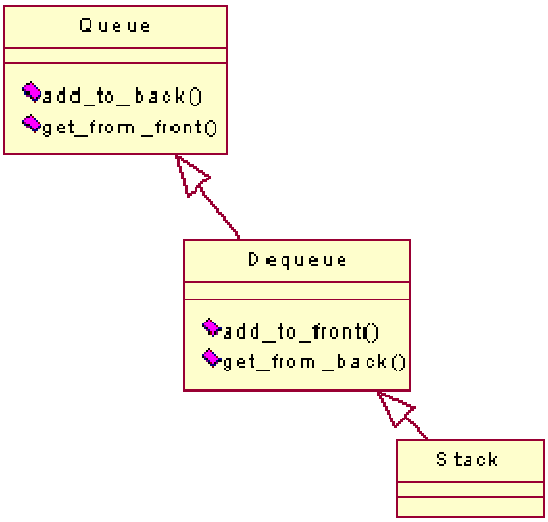
100%: هل كل المنهاجات في الصنفية الأساس طابور Queue تنطبق عل الصنفية الجديدة طابور_د Dequeue؟ الإجابة هي بنعم، لأن كل هذه المنهاجات مطلوبة.

هو_نوع_من: هل العبارة التالية لها معنى؟ "طابور_د هو نوع من الطابور". نعم، الجملة مفيدة و لها معنى، لأن الطابور_د هو نوع خاص من الطابور.

إذا الوراثة هنا صحيحة و صالحة **valid**.

الآن، لنذهب أبعد. لنقل أننا نريد إنشاء صنفية ركام Stack. نريد لها أن تدعم المنهاجين: add_to_front() و remove_from_front() (إضافة_للمقدمة، إزالة_من_الخلفية).

بدلا من كتابة صنفية الركام من الصفر، يمكننا ببساطة توريثها من صنفية dequeue طابور_د، لأن الأخيرة لديها كلا المنهاجين.



شكل 77: إنشاء صنفية Stack ركام ..دون الحاجة لعمل إضافي!

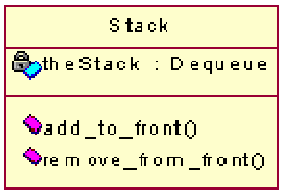
يمكننا أن نشعر بالفخر كوننا أعدنا استخدام Dequeue و أنشأنا صنفية Stack (ركام) دون الحاجة لمزيد من العمل. أي توليف يستخدم هذا الركام يمكنه إضافة بنود للمقدمة، و إزالتها من المقدمة، لذلك فنحن لدينا صنفية ركام تعمل.

برغم هذا يجب أن لا نشعر بالفخر كثيرا. *فهذا بالتأكيد تصميم ضعيف جدا*. لا ننسى أن صنفية Stack ركام قد ورثت أيضا المنهاجان: إضافة_للخلف و إزالة_من_الخلف (add_to_back() و remove_from_back()) – هذان منهاجان ليس لهما أي معنى في الركام! لذا فإن صنفية ركام لا تصمد أمام اختبار قاعدة **100%**. بالإضافة لذلك، فإن الركام لا يصمد أيضا أما اختبار **هو_نوع_من**، لأن الركام ليس فعلا نوع من Dequeue طابور_د على الإطلاق.

إذا نحن خلقنا مشكلة صيانة – يعني أن أي توليف يستخدم الركام يمكنه عن طرق الخطأ إضافة بنود لخفية الركام! كيف يمكننا مراوغة هذا الأمر و الدوران من حوله؟؟ حقيقة لا توجد أية طريقة، بدون إفساد صنفية ركام. لنتذكر أننا لا نستطيع إزالة منهاجات من صنفية عند التوريث.

الحل يكمن في استخدام التجمع aggregation بدلا من الوراثة. نقوم بإنشاء صنفية جديدة تسمى stack ركام، و نضمنها حقل من نوع Dequeue طابور_د كسمة من سماتها الخاصة.

يمكننا الآن توفير المنهاجان العموميان add_to_front() و remove_from_front() (إضافة_للمقدمة، إزالة_من_الخلفية) كجزء من صنفية ركام. إنجاز هذين المنهاجين سيكون عبارة عن استدعاءات بسيطة لنفس المنهاجين اللذان تحتويهما صنفية Dequeue طابور_د.



شكل 78: صنفية Stack وهي تعيد استخدام صنفية Dequeue بفعالية

الآن، مستخدمو صنفية Stack يمكنهم فقط استدعاء هذين المنهاجين العموميين – المنهاجات المتضمنة في صنفية Dequeue "المخفية" تعد خاصة. هذا الأمر يزيد من تأكيد أن لصنفية Stack واجهة عالية الاتساق cohesive، و ستكون أكثر سهولة عند فهمها و صيانتها.

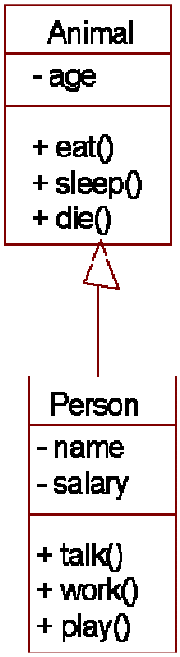
المشاكل عند الوراثة

برغم أن الوراثة تبدو و كأنها أداة قوية لتحقيق إعادة الاستخدام، إلا أن الوراثة يجب أن يتم التقرب منها بحذر. المبالغة في استعمال الوراثة يمكن أن تؤدي إلى هرميات معقدة جدا يصعب فهمها. هذه المشكلة تعرف بتكاثر الصنفيات *proliferation of classes*. المشكلة ستكون أكثر جدية عندما يتم استخدام الوراثة بطريقة غير صحيحة (كما وصف أعلاه). لذا يجب التأكيد على ضرورة الاقتصاد عند التوريث، و أنه يتطابق مع قاعدتي "100%" و "هو_نوع_من".

بالإضافة إلى ذلك، تعد الوراثة إعادة استخدام مكشوفة –صندوق أبيض–. التغليف الفاصل بين الصنفية الأساس و الصنفية المشتقة ضعيف جدا بصورة عامة، التغيير في الصنفية الأساس يمكن أن يؤثر في أية صنفيات مشتقة، و بالتأكيد، أي مستخدم لأية صنفية سيحتاج لمعرفة كيفية عمل سلسلة الصنفيات التي تعلوها.

من يستخدم صنفية جذورها مطمورة بعمق 13 صنفية متسلسلة الوراثة سوف يعاني من صداع حقيقي و هو يتعامل مع هذه الصنفيات – كم صنفية يمكن للمرء أن يتصارع معها في ذهنه في وقت واحد؟؟

منظورية السمات



شكل 79: وراثة بسيطة

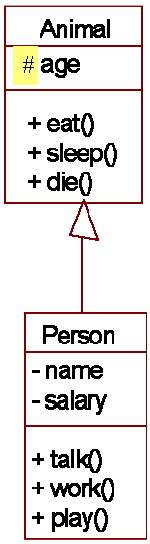
لنلاحظ شجرة الوراثة في الشكل أعلاه. من المهم تفهم أن الأعضاء الخاصين في الصنفية الأساس Animal غير منظورين من قبل الصنفية المشتقة Person. و بذلك فإن المنهاجات talk() ، work() و play() لا يمكنها النفاذ إلى سمة age .

هذا له معنى بطريقة ما، لأننا نستطيع أن نجادل بالقول أن المنهاجات التابعة لصنفية Person يجب أن لا تتمكن من العبث بالسمات الخاصة بصنفية Animal.

عموما، أحيانا يكون هذا القيد شديدا جدا، و قد نحتاج لأن نسمح للصنفية المشتقة بمشاهدة السمات في الصنفية الأساس. بالطبع، يمكننا جعل هذه السمات عمومية، لكن هذا سوف يحطم التغليف encapsulation و يعرض السمات لأن تتكشف أمام كل العالم. لذلك يقدم مفهوم المنحى للكائن "أرضية محايدة" تسمى المنظورية المحمية protected visibility.

العضو المحمي سوف يبقى خصوصيا أمام العالم الخارجي، لكنه يظل منظـــورا لأيـــة صنفيات مشتقة. معظم لغات المنحى للكائن تدعم المنظورية المحمية.

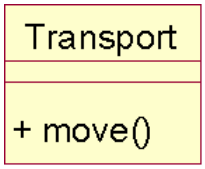
ترميز UML للعضو المحمي مبين أدناه:



شكل 80: سمة Age محمية الآن، لذلك فهي مرئية لصنفية Person. و تظل خصوصية بالنسبة لباقي الصنفيات.

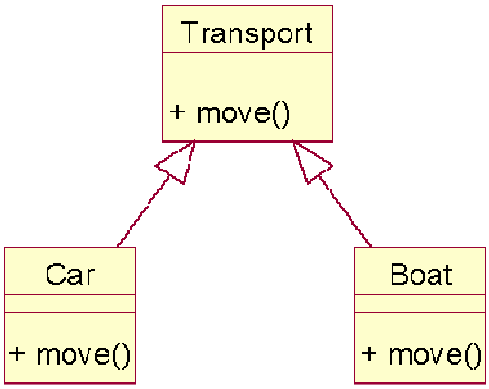
التشكل Polymorphism

الصنفيات المشتقة يمكنها إعادة طريقة بناء المنهاج فيها. مثلا، لنأخذ صـــنفية تـــسمى "Transport" (وسيلة نقل). أحد المناهج التي يجب أن تحويها هي ()move حركة ، لأن كل وسائل النقل لابد أن تكون قادرة على الحركة.



شكل 81: صنفية Transport

إذا أردنا خلق صنفية Boat مركب و صنفية Car سيارة، فإننا حتمـــا ســـنرثهما مـــن صنفية Transport وسيلة نقل، حيث أن كل المراكب يمكنها الحركة، و كل الـــسيارات يمكنها الحركة:



شكل 82: المركب و السيارة مشتقتان من وسيلة نقل. القاعدتان: هو_نوع_من و 100% تنطبقان هنا.

عموما، السيارات و المراكب تتحركان بطرق مختلفة. لذا قد نحتاج لبناء منهاجي الحركة في كل منهما بطريقة مختلفة. هذا أمر وارد جدا في المنحى للكائن، و يسمى Polymorphism التشكل أو تعددية الشكل.

الصنفيات المجردة

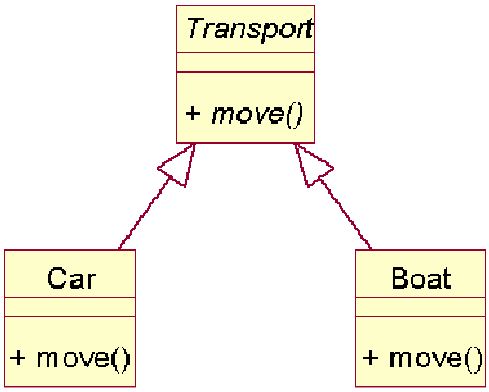
غالبا عند التصميم، نحتاج لترك منهاج ما بدون تنفيذ أو بناء، و يترك بناؤه الفعلي لاحقا "تحت" في شجرة الوراثة.

مثلا، و بالعودة للحالة أعلاه. أضفنا منهاجا يسمى move() إلى Transport. هذا تصميم جيد، لأن كل وسائل النقل تحتاج لأن تتحرك. لكننا حقيقة لا نستطيع بناء هذا المنهاج، لأن وسيلة النقل تضم نطاقا واسعا من الأصناف، كل واحد منها يتحرك بطريقة مختلفة.

ما نستطيع فعله هو أن نجعل من صنفية Transport صنفية مجردة abstract. هذا يعنى أن نترك بعض المنهاجات فيها أو ربما كلها غير منجزة.

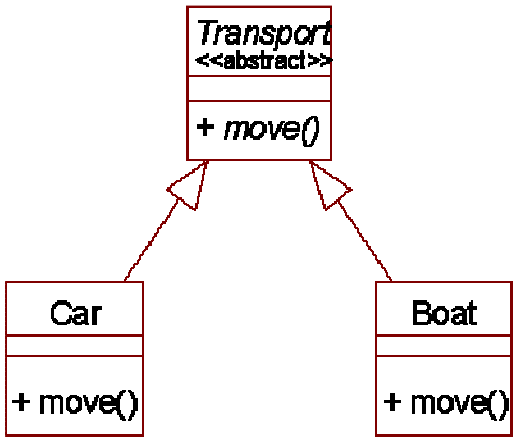
عندما نشتق صنفية سيارة من صنفية وسيلة النقل، يمكننا عندئذ بناء المنهاج الخاص بالسيارة، و هكذا مع صنفية مركب.

صيغة UML للتعبير عن الصنفية المجردة و المنهاج المجرد هي باستعمال الأحرف المائلة كما في الشكل التالي:



شكل 83: لم نقم ببناء منهاج move() في صنفية Transport

هذه إحدى الجوانب التي لم تكن فيها UML موفقة. الحروف المائلة Italics غالبا ما تكون صعبة التمييز في المخطط (و يصعب كتابها يدويا إذا رسمنا مخططا على ورقة). الحل يكمن في استعمال الوسم Stereotype* في UML على الصنفيات المجردة كالتالي:



شكل 84: تبيين الصنفية المجردة باستخدام الوسم

* الوسم هو إضافة لمفردات UML، تسمح لك بخلق أنواع جديدة من كتل البناء شبيهة بتلك الموجودة لكنها خاصة بالمسألة التي تعالجها. يعبر عن الوسم رسوميا كاسم بين قوسين مزدوجين و توضع فوق اسم عنصر آخر. كخيار، يمكن التعبير عن عنصر الوسم باستخدام أيقونة (صورة مصغرة) جديدة ترفق بذلك الوسم. من كتاب: دليل المستخدم للغة النمذجة الموحدة " The Unified Modeling Language User Guide- Booch, Rumbaugh, Jacobson " . المترجم.

قوة التشكل

يعد التشكل أو التشكلية موضوع قائم بنفسه عند تطبيق مبدأ الاحلالية substitutability. ينصّ هذا المبدأ على أن أي منهاج نكتبه و يتوقع له أن يعمل في صنفية معينة، **يمكنه أن يعمل بنجاح أيضا في أي صنفية مشتقة.**

مثال في توليف يوضح هذه النقطة – التوليف مكتوبة بلغة جافا تشبيهية Pseudocode، لكن المبدأ سيظل واضحا:

بعض التوليف الإضافي

```
public void accelerate (Transport theTransport, int
acceleration)
{
    -- some code here بعض التوليف هنا
    theTransport.move();
    -- some more code بعض التوليف الإضافي
}

--
--
--
accelerate (myVauxhall);
accelerate (myHullFerry);
```

شكل 85: توليف جافا يوضح مبدأ الاحلالية

في هذا المثال، كتبنا منهاجا يسمى accelerate تسريع. و يستقبل محدد من نوع "Transport" وسيلة نقل، و يفترض بهذا المنهاج أن يقوم بتسريع وسيلة النقل باستخدام المنهاج ()move –التابعة لوسيلة النقل–.

الآن، نستطيع بأمان مناداة هذا المنهاج accelerate، و أن نمرر له كائن سيارة (لأنها صنفية فرعية من وسيلة نقل)، كما يمكننا مناداته بتمرير كائن قارب أيضا. الوظيفية function (المنهاج) التي كتبناها ببساطة لا يهمها النوع الفعلي للكائن الذي تستقبله، مادام هذا الكائن مشتق من نوع وسيلة نقل Transport.

في هذا مرونة كبيرة، ليس فقط لأننا كتبنا منهاجا واحدا عاما يمكنه العمل مع دائرة واسعة من الصنفيات المختلفة، بل لأن هذا المنهاج يمكنه التعامل مستقبلا مع صنفيات لم ينم تصميمها بعد. لاحقا، قد يقوم أحدهم بخلق صنفية جديدة تسمى "طائرة" مشتقة من صنفية Transport، و سيظل المنهاج ()accelerate شغالا، وسيستقبل صنفية طائرة الجديدة بدون مشاكل، و بدون أي تعديل أو إعادة بناء البرنامج!

هذا هو سبب لماذا لا يمكننا إزالة منهاج method عندما نشتق صنفية جديدة. إذا سمح لنا بذلك، فقد يتم إزالة المنهاج ()move من صنفية الطائرة، و تضيع كل الفائدة التي أشرنا إليها أعلاه!

ملخص

بساطة ترميز الوراثة في UML

الصنفيات يكن ترتيبها في وراثة هرمية

الصنفية الفرعية يجب أن ترث كل السلوكيات العمومية للصنفية الأم

المنهاجات و السمات المحمية يتم توريثها أيضا

التشكل أداة بالغة القوة من أجل إعادة استخدام التوليف.

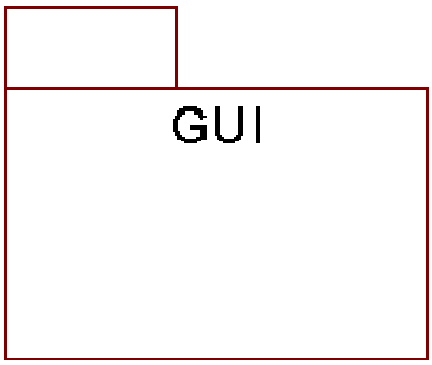
الفصل 16: معمار النظام ـ الأنظمة الكبيرة و المركبة

حتى الآن في هذا الكتاب، اعتنينا بالأنظمة "الصغيرة". بصفة عامة، كل ما قلناه حتى الآن يمكن تطبيقه بسهولة على مشاريع تضم ، لنقل 3 أو 4 مطورين، مع مجموعة من التكرارات كل واحدة منها تمتد لشهرين.

في هذا الفصل، سوف ننظر في بعض القضايا التي تحيط بتطويرات أضخم و أكثر تعقيدا. هل UML ضمن إطار العمل التكراري التزايدي قابلة لأن تتوسع؟ و ماذا يمكن لها أن توفر أيضا من أجل احتضان التعقيد الذي في مثل هذه التطويرات؟

مخطط التحزيم في UML

كل المشغولات Artefacts في UML يمكن تنظيمها في حزم أو رزم و تسمى "UML Packages"، الحزمة هي أساسا حاوية يتم فيها وضع العناصر المتصلة ببعض – تماما مثل المجلد أو الدليل في نظام التشغيل.



شكل 86: ترميز لحزمة UML

في المثال أعلاه، قمنا بإنشاء حزمة تسمى "GUI". و سنقوم على الأرجح بوضع مشغولات UML ذات العلاقة بواجهة الاستخدام الرسومية داخل الحزمة.

نستطيع عرض مجموعات من الحزم، و علاقاتها فيما بينها، في مخطط UML للتحزيم. فيما يلي مثال بسيط:

شكل 87: ثلاث حزم UML

في المثال أعلاه، قمنا بتدوين "نموذج الثلاث صفوف" three tier لتطوير البرمجيات. العناصر داخل حزمة "Presentation" تعتمد على العناصر داخل حزمة "Business".

لنلاحظ أن المخطط لا يظهر ما يوجد فعلا داخل الحزمة. لذلك، فإن مخطط التحزيم يقدم رؤية بمستوى عال للنظام. عموما العديد من الأدوات البرمجية المساعدة تتيح لمستخدمها أن ينقر على صورة الحزمة لتنفتح عارضة محتوياتها.

الحزمة يمكن أن تحتوي حزم أخرى، و لذلك يمكن ترتيب الحزم في هرميات، و مرة أخرى ، تماما مثل هيكلية مجلدات الملفات في نظم التشغيل.

العناصر داخل الحزمة

داخل الحزمة يمكن وضع أيا من مشغولات UML. عموما، الاستخدام الأكثر شيوعا للحزمة هو أن يتم تجمع الصنفيات ذات العلاقة مع بعض. أحيانا يستعمل النموذج لجمع وقائع الاستخدام ذات العلاقة مع بعض.

داخل حزمة UML، أسماء العناصر يجب أن تكون مميزة، لذا و كمثال، فإن اسم كل صنفية داخل الحزمة لابد أن يكون مميزا. عموما أحد الفوائد المهمة للحزم أنه لا يهم إذا وجد تضارب في الأسماء بين عنصرين من حزم مختلفة. هذا يوفر ميزة فورية بحيث إذا كان لدينا فريقين يعملان على التوازي، الفريق أ لا داعي لأن يقلق على محتويات حزمة الفريق ب. تضارب الأسماء لن يحدث ما دامت الأسماء تكون ضمن سياق الحزم.

لماذا التحزيم؟

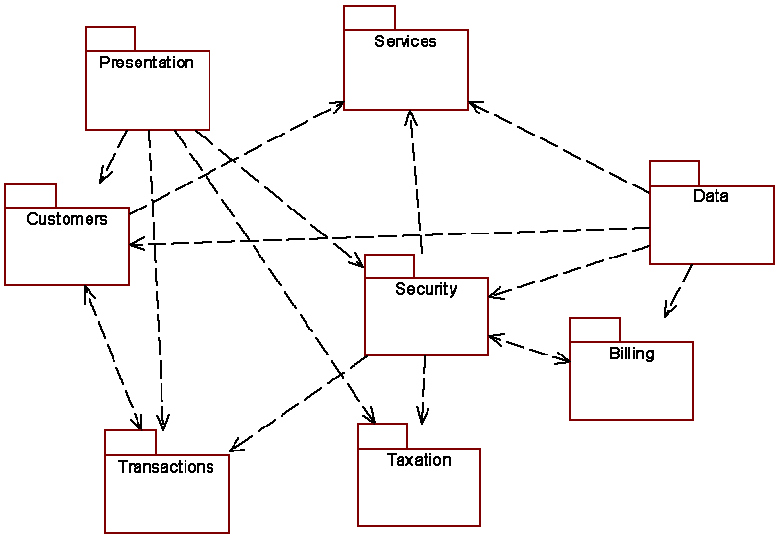
لماذا نزعج أنفسنا بالتحزيم. حسن، إذا استخدمنا التحزيم بعناية يمكننا:

- تجميع الأنظمة الكبيرة في أنظمة فرعية يسهل مناولتها
- السماح بتطوير التكرارات على التوازي

أيضا، إذا قمنا بتصميم كل حزمة بطريقة صحيحة و وضعنا واجهات واضحة بين الحزم (المزيد حول هذا قريبا)، سيكون لدينا فرصة لتحقيق تصميم يمكننا من إعادة استخدام التوليف. إعادة استخدام الصنفيات عادة ما يكون أمرا صعبا (أحيانا بمعنى أن الصنفيات تكون من الصعب معاودة استعمالها)، بينما الحزمة المصممة جيدا يمكن أن تتحول إلى مكون component صلب و متماسك قابل لإعادة الاستخدام. مثلا، حزمة رسوميات يمكن استخدامها في مشاريع مختلفة.

بعض الاستكشافات في التحزيم

لنفترض في هذا المقطع إننا نستخدم مخطط التحزيم لتقسيم الصنفيات داخل حزم سهلة عند الفهم و الصيانة.



شكل 88: هيكل حزمة حسنة التصميم؟

عدة استكشافات من الفصل الذي تحدثنا فيه عن GRASP (أنماط توزيع المسؤولية) يمكن أن تنطبق تماما على التحزيم:

الخبير Expert

أية حزمة يجب أن تتبع لها صنفية ما؟ يجب أن يكون واضحا كل صنفية لمن تتبع – إذا لم يكن الأمر واضحا فعلى الأرجح أن مخطط التصميم فيه عيب.

الاتساق العالي High Cohesion

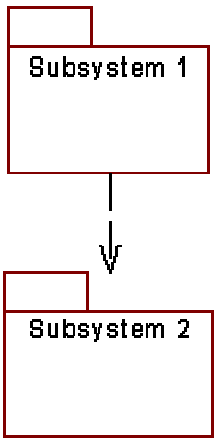
يجب على الحزمة أن لا تقوم بالكثير من العمل (و إلا سيكون من الصعب فهمها و بالتأكيد صعب إعادة استخدامها).

ضعف الاقتران Loose Coupling

يجب المحافظة على الاعتماديات بين الحزم في أدنى مستوى. المخطط أعلاه مثال للتوضيح، ويبدو بشكل مريع! لماذا يوجد الكثير جدا من خطوط التواصل عبر الحزم.

معالجة الاتصالات عبر الحزم

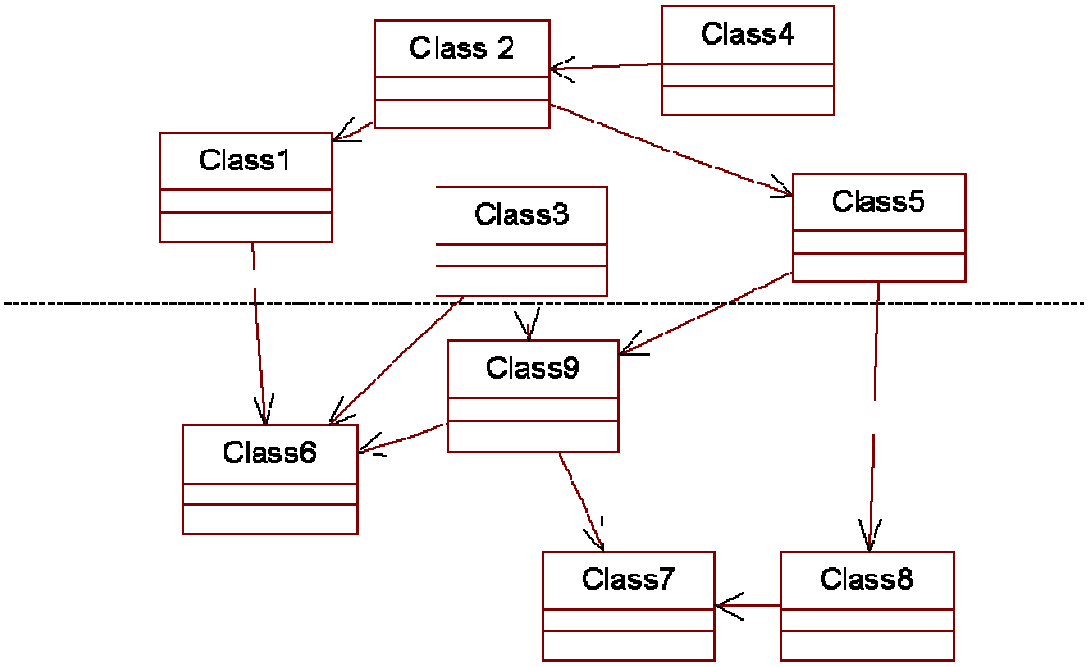
بافتراض أن لدينا حزمتين، كل واحدة تحوي على عدد من الصنفيات.



شكل 89: نظامان فرعيان تم نمذجتهما في حزمة UML

من خلال أسهم الاعتماد، يمكننا رؤية أن الصنفيات في حزمة "النظام الفرعي 1" تخاطب صنفيات في حزمة "النظام الفرعي 2".

إذا حفرنا أعمق و نظرنا لما بداخل الحزمتان، قد نرى شيئا مثل الشكل التالي (تم حذف السمات و العمليات من الرسم لغرض التوضيح):



شكل 90: صنفيات عبر نظامين فرعيين. الخط المتقطع يمثل الحدود بين النظامين

تقريبا، لدينا حالة حيث أية صنفية من حزمة "النظام الفرعي 1" يمكنها مناداة أية صنفية من حزمة "النظام الفرعي 2". هل هذا تصميم جيد؟

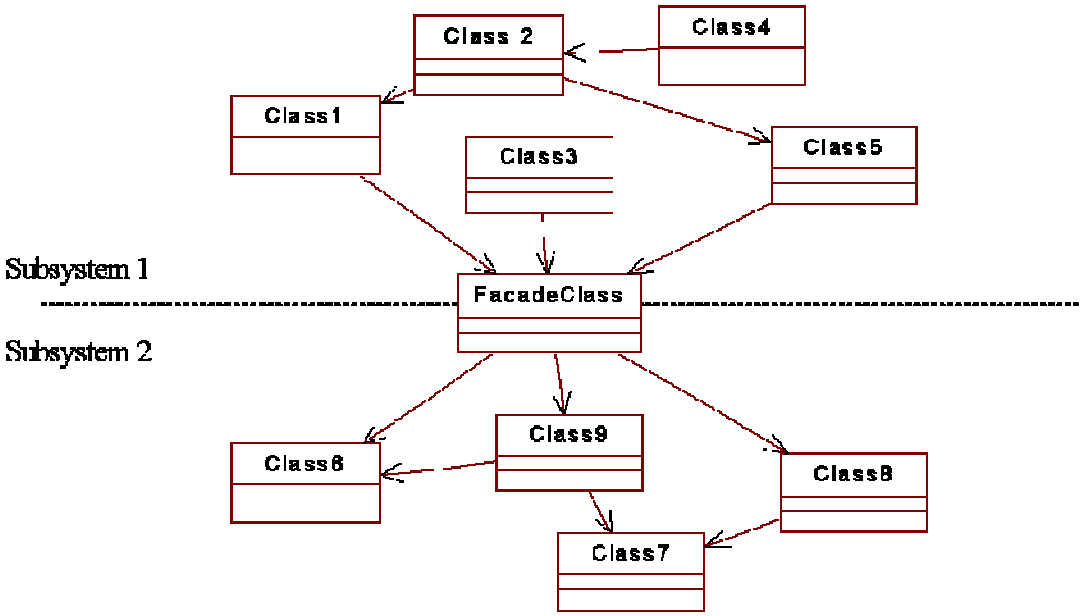
ماذا لو أردنا إزالة حزمة النظام الفرعي 1 و استبدالها بنظام فرعي جديد (لنقل أننا نقوم بإزالة واجهة استخدام طرفية كئيبة و استبدالها بواجهة استخدام رسومية حديثة تغني و ترقص).

سيتطلب هذا الكثير من العمل لفهم تأثير هذا التغيير. سيكون علينا التأكد من أن كل صنفية في النظام القديم تستبدل بما يقابلها من صنفية في النظام الجديد. فوضى شديدة،

وعمل تعزوه الأناقة. لحسن الحظ، يوجد نمط تصميم يسمى الواجهة Facade لمساعدتنا في حل هذه المعضلة.

نمط الواجهة

الحل الأحسن هو أن نوظف صنفية أخرى لتلعب دور الوسيط بين النظامين الفـرعيين. هذا النوع من الصنفيات يدعى بالواجهة Façade، و ستقدم، عبر واجهتهـا العموميـة، مجموعة من كل المنهاجات العمومية التي يدعمها النظام الفرعي.



شكل 91: الواجهة كحل

الآن، الاستدعاءات لن تتم عبر حدود النظام الفرعي، بل يتم توجيه كل الاستدعاءات من خلال الواجهة. فإذا كان يجب تغيير نظام فرعي ما، فإن التغيير الوحيد المطلوب سيكون تحديث الواجهة.

لغة جافا لديها دعما ممتازا لهذا المفهوم. فبجانـب المنظوريـات المعتـادة للـصنفية: الخصوصي، و العمومي و المحمي، تقدم جافا بعدا رابعا من الحماية تسمى حماية تحزيم Package Protection. فإذا تم تصميم صنفية كحزمة بدلا من صنفية عموميـة، فقـط الصنفيات التابعة لنفس الحزمة يمكنها النفاذ إليها. هذا يعد مستوى أقوى من التغليـف –

بجعل كل الصنفيات محجوبة ما عدا حزمة الواجهة، كل فريق يبني نظاما فرعيا يمكنه فعلا العمل باستقلالية عن الفريق الآخر.

المرتكز المعماري للتنشئة

العملية الموحدة من راشيونال تؤكد بقوة على مفهوم المرتكز المعماري للتنشئة Architecture-centric development. و تعني أساسا، أن النظام قد خطط له لأن يكون مجموعة من الأنظمة الفرعية منذ مرحلة مبكرة من تنشئة المشروع.

من خلال إنشاء مجموعة من الأنظمة الفرعية الصغيرة السهلة التناول، يمكن تخصيص فرق تطوير صغيرة (ربما فقط من 3 أو 4 أشخاص)، كل فريق يتصدى لنظام فرعي، و بقدر الإمكان يعملون على التوازي، كل مستقل عن الآخر.

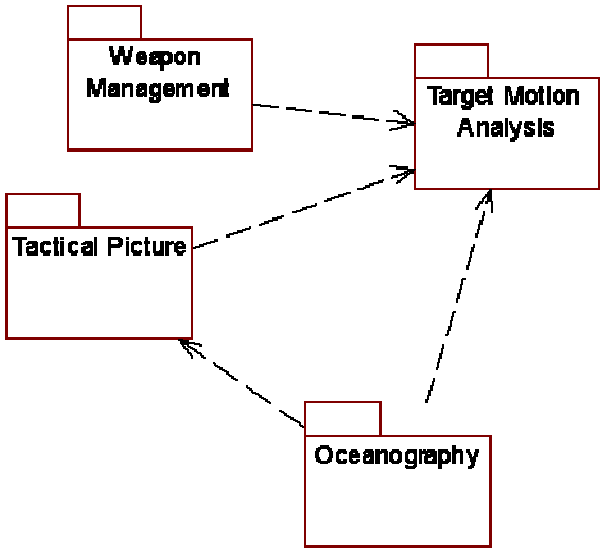
واضح أن في هذا الشأن القول أسهل بكثير من الفعل. لتبيان أهمية هذا النشاط المعماري، يتم تعيين فريق للمعماريات كامل الوقت (قد يكون شخص واحد). يتم تكليف هذا الفريق بإدارة النموذج المعماري – سوف يتولون هذا النموذج و صيانة كل شيء يتعلق بالنظام في مستواه الأعلى و صورته العامة.

بعبارة أخرى، سيتولى هذا الفريق ما يعبر عنه مخطط التحزيم. بالإضافة لذلك، سيتولى فريق المعمار مراقبة الواجهات (Facades) بين الأنظمة الفرعية. واضح، أنه مع تطور المشروع، سيكون هناك تغييرات يجب إعدادها في الواجهات، لكن هذه التغييرات يجب أداؤها من قبل فريق المعمار المركزي، و ليس من قبل المطورين الذين يعملون على الأنظمة الفرعية.

ما دام فريق المعمار يحافظ على رؤية ثابتة للنظام في مستواه الأعلى، سيكونون هم الأنسب لفهم تأثير التغيرات التي تطرأ على الواجهات بين الأنظمة الفرعية.

مثال

لنظام قيادة و تحكم، قام فريق المعماريات بوضع تخطيط أولي لمعمار النظام من خلال تحديد المجالات الرئيسية للوظائف التي يجب أن يوفرها النظام. فقاموا بإنتاج مخطط التحزيم التالي:

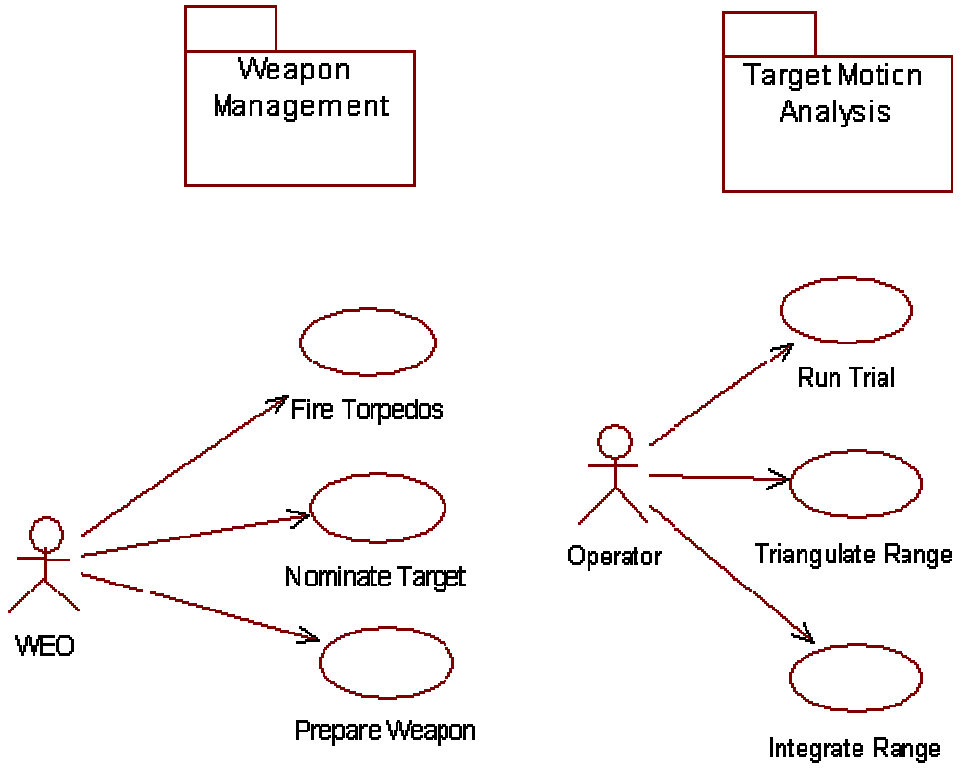


شكل 92: خطة مبدئية للأنظمة الفرعية باستخدام مخطط UML للتحزيم

لنلاحظ أن المعمار ليس نهائيا – فريق المعماريات سوف يقوم باستنباط وتوسيع مفردات المعمار مع تطورات المشروع ، لكي يحوي التشابك و التعقيد الخاص بكل نظام فرعي.

سيداوم الفريق على توصيف النظم الفرعية حتى يصير حجم كل نظام فرعي غير بالغ التعقيد و سهل الإدارة.

عندها سيكون مناسبا بناء وقائع الاستخدام لكل نظام فرعي. كل نظام فرعي يتم معاملته و كأنه نظام قائم بنفسه، تماما كما فعلنا في المراحل الأولى من هذا الكتاب:



شكل 93: نمذجة وقائع الاستخدام على التوازي

معالجة وقائع الاستخدام الضخمة

مشكلة أخرى ترافق هذا النوع من التطوير كبير الحجم هو أن وقائع الاستخدام المبدئية التي تم تحديدها في طور التفصيل قد تكون كبيرة جدا بصورة يصعب تنشئتها في تكرار واحد. الحل هو أن **لا** نطيل من أمد التكرار (هذا يسبب في بروز مشكلة التعقيد مرة أخرى). بدلا من ذلك، يكمن الحل في تجزئة واقعة الاستخدام إلى سلسلة من "الإصدارات" يسهل معالجتها.

مثلا، واقعة الاستخدام "Fire Torpedoes" إطلاق طوربيدات المبينة أعلاه، وجد، بعد طور التفصيل، أنها تحديدا واقعة استخدام ضخمة وصعبة. لذلك يتم تجزئة واقعة الاستخدام هذه إلى إصدارات منفصلة، كالتالي:

- إصدار 1 – السماح بفتح غطاء الفوهة
- إصدار 2 – السماح بضبط التروس
- إصدار 3 – السماح بإطلاق السلاح

الهدف هو ضمان أن كل إصدارة تكون سهلة الفهم، وقابلة للإنجاز في تكرار واحد. إذن واقعة استخدام إطلاق طوربيدات سوف تستهلك ثلاثة تكرارات لاستكمالها.

طور البناء

يستمر العمل في طور البناء كما تم وصفه في الفصول الأولى، لكن كل نظام فرعي يتم تطويره، تكراريا، من قبل فرق منفصلة، تشتغل على التوازي، و مستقلة عن بعضها قدر الإمكان.

في نهاية كل تكرار، يتم الدخول لمرحلة اختبارات الدمج و التكامل، حيث يتم اختبار الواجهات عبر الأنظمة الفرعية.

ملخص

تطرق هذا الفصل لبعض القضايا التي تحيط بتنشئة النظم الكبيرة الحجم. واضح أنه بالرغم من أن لغة UML تم تصميمها كي تطبق على أي قياس، فإن نقل إطار العمل التكراري التزايدي إلى المشاريع الضخمة أبعد من أن يكون عملا بسيطا.

مؤسسة راشيونال تقترح المقاربة التالية:

- تحديد الأنظمة الفرعية منذ المراحل الأولى
- جعل التعقيد قابلا للمناولة بقدر الإمكان
- التكرار على التوازي بدون هتك الواجهات
- تعيين فريق مركزي للمعماريات

نموذج التحزيم التي تقدمه UML يوفر طريقة لاحتواء التشابك الضخم، هذا النموذج يجب أن يتولاه فريق المعماريات.

الفصل 17: نمذجة الحالات

بعد أن أخذنا استراحة نظرنا فيها إلى الوراثة و معمارية النظام، سنعود الآن إلى مرحلة التصميم من طور البناء و ننظر في نمذجة الحالة state.

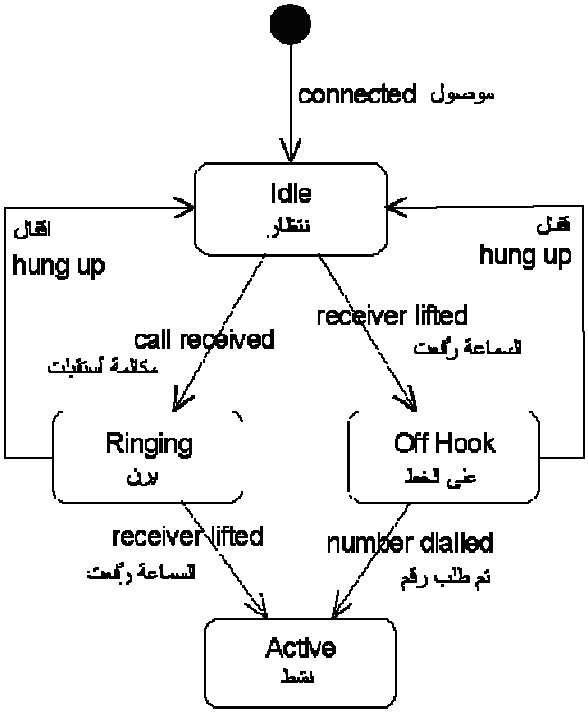
مخططات الحالة تتيح لنا نمذجة الحالات المحتملة التي يكون فيها الكائن. النموذج يسمح لنا برصد الأحداث المهمة التي يمكن أن يتعرض لها الكائن، و أيضا تأثير هذه الأحداث.

لهذه النماذج العديد من التطبيقات، لكن ربما أقوى هذه التطبيقات هي تلك التي تـــضمن أن الأحداث الشاذة و الغير مسموح بها لا يمكن لها أن تقع داخل النظام.

المثال الذي ورد في مقدمات هذا الكتاب (فصل 4 نبذة عامة عن UML : مخططـــات الحالـــة صفحة 39-40) يتحدث عن وضعية تحدث كثيرا للأسف، و يصلح لعنوان صحيفة محلية – فاتورة غاز ترسل لمستهلك مات منذ خمس سنوات!

مخططات الحالة المكتوبة بعناية تمنع وقوع هذا النوع من الأحداث الخاطئة.

مثال رسم حالة

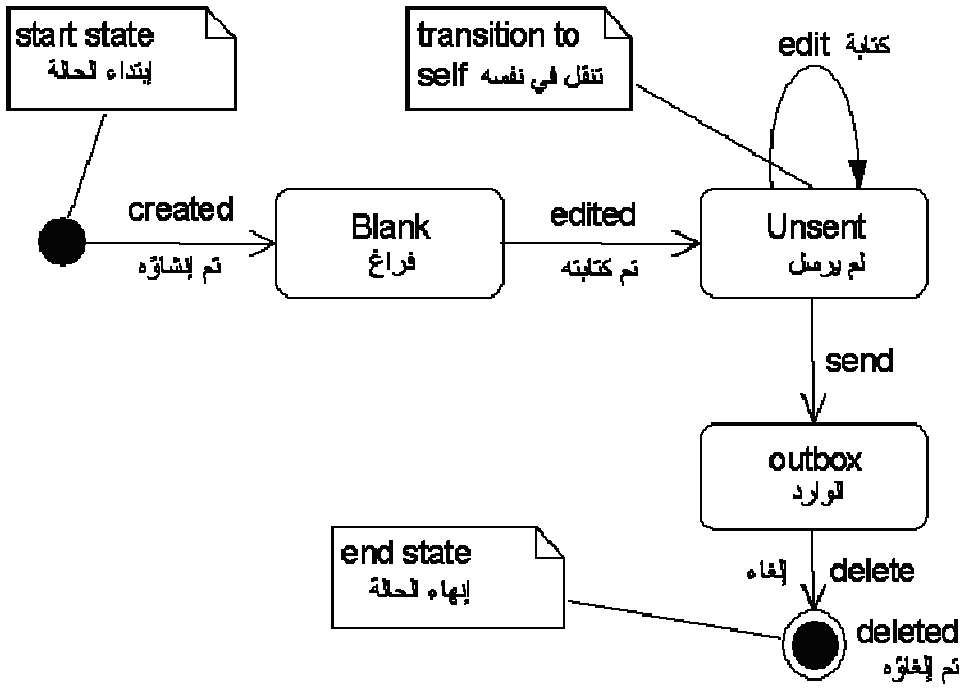


شكل 94: مثال رسم حالة؛ هاتف

سوف ننظر في الصيغة النحوية لهذا المخطط بعد قليل، لكن أساسيات المخطط واضحة. معروض هنا تتابع الأحداث التي يمكن وقوعها لجهاز الهاتف ، كذلك يعرض المخطـــط الحالات التي يمكن أن يكون عليها الهاتف.

مثلا، من حالة الانتظار، الهاتف إما أنه سيكون على الخط (إذا رفعـــت الـــسماعة)، أو سيصدر رنينا (إذا استقبل مكالمة).

صيغة مخطط الحالة

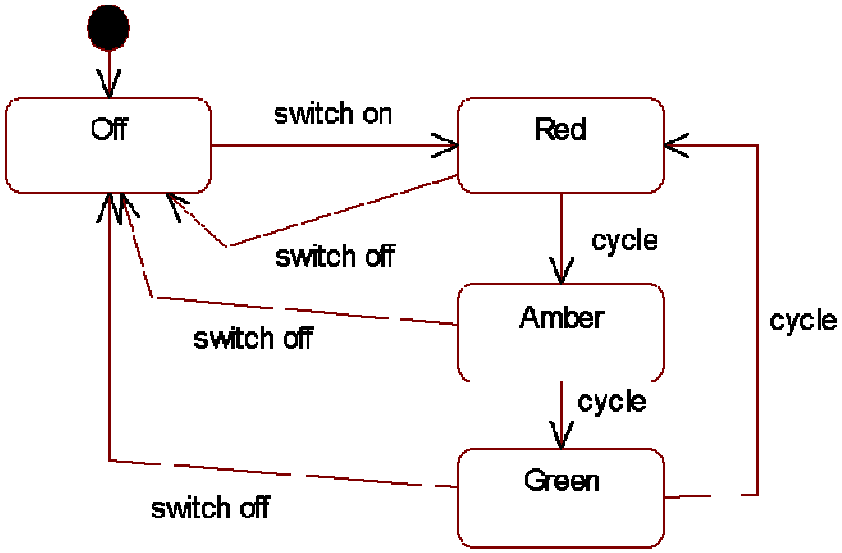


شكل 95: صيغة مخطط الحالة – مثال لبريد الكتروني

المخطط أعلاه يعرض معظم الصيغ النحوية لمخطط الحالة. سيكون للكائن حالة ابتداء (الدائرة الممتلئة)، تصف حالة الكائن عند نقطة خلقه. معظم الكائنات لديها حالة إنهاء ("عين الثور" bullseye)، تصف الحدث الذي وقع لمحو الكائن.

بعض الأحداث تتسبب في حالة تنقل تبقي الكائن على حاله. في المثال أعلاه، البريد الالكتروني e-mail يمكنه استقبال حدث "كتابة" فقط في حالة كونه "لم يرسل". لكن الحدث لا يسبب في تغيير حالته. هذه صيغة مفيدة لتبيان أن حدث "كتابة" لا يمكن وقوعه في أي من الحالات الأخرى.

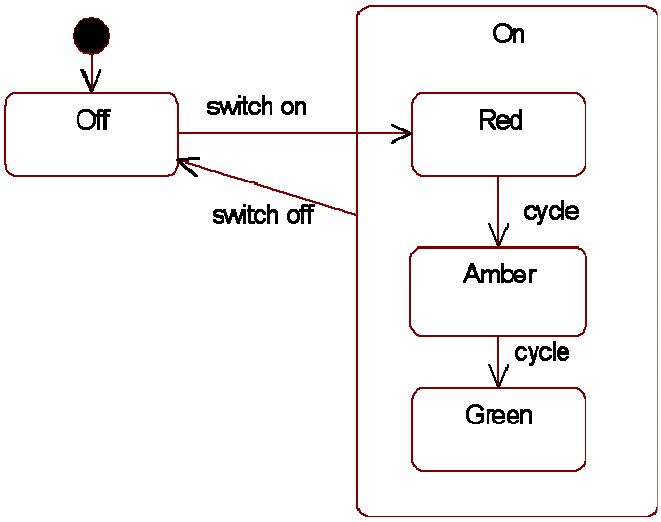
الحالات الفرعية



شكل 96: نموذج فوضوي لحالة

أحيانا، يتطلب الأمر وصف حالات داخل حالات. الشكل أعلاه صحيح تماما (يصف حالات كائن إشارة المرور الضوئية)، لكنه غير أنيق بالمرة. بشكل رئيسي تقع أحداث إطفاء الضوء switch off في أي وقت و هذه المجموعة من الأحداث هي التي تتسبب في الفوضى الحاصلة

توجد حالة عليا "superstate" في هذا النموذج. إشارة المرور قد تكون في حالة إضاءة "On" أو حالة إطفاء "Of". عندما تكون في حالة إضاءة، يمكن أن تكون في سلسلة من الحالات الفرعية : "حمراء"، "صفراء" أو "خضراء". UML في هذا السياق تسمح بوجود احتضانات "nesting" للحالات:

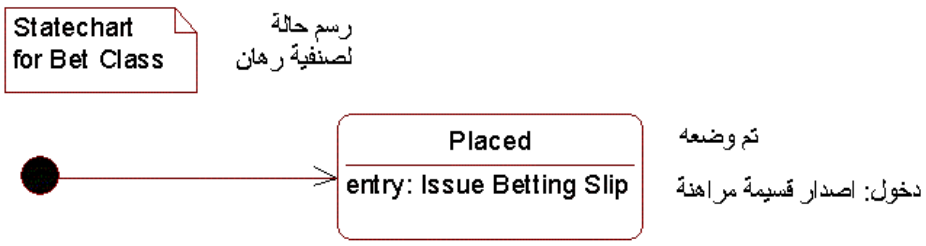


شكل 97: نموذج حالة أبسط باستخدام الحالات الفرعية

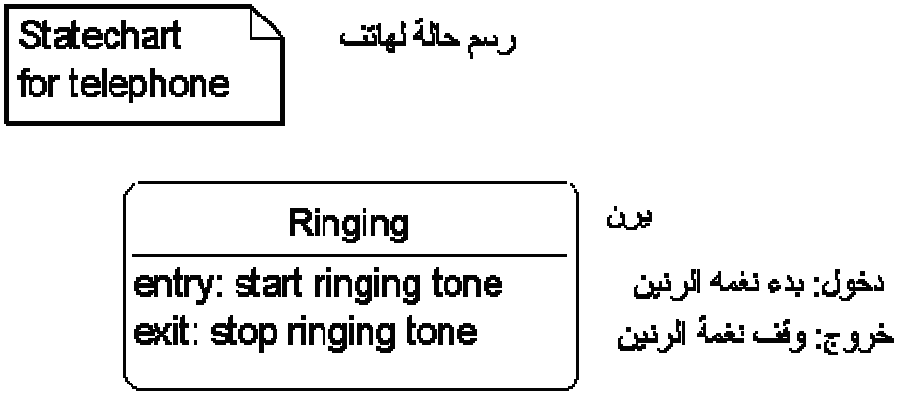
لنلاحظ أنه في المخطط أعلاه، سهم صغير موجه إلى حالة "red" أحمر يشير إلى أنهـا الحالة الافتراضية – عند ابتداء حالة إضاءة "on"، فإن الضوء سيتم ضبطه على الأحمر "Red".

أحداث دخول/خروج

أحيانا يكون من المفيد رصد أية أفعال مطلوب وقوعها عندما تبدأ الحالة. التدوين التالي يسمح بذلك:



شكل 98: هنا، نحتاج إلى إصدار قسيمة مراهنة عند تغير الحالة



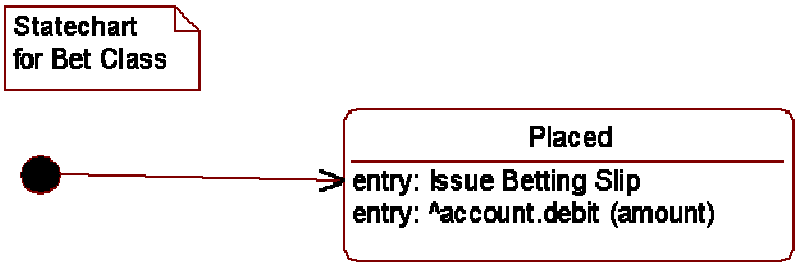
شكل 99: هنا، تبدأ نغمة الرنين مع الدخول في الحالة – تتوقف نغمة الرنين عند الخروج

أحداث الإرسال

التدوين=الترميز أعلاه يكون مفيدا عندما نحتاج إلى وضع ملاحظة بأن إجـراء معـين مطلوب أخذه. يمكننا ربط هذا بفكرة الكائنات و التعاونات collaboration. إذا كان حالة التنقل تتضمن أن رسالة يجب إرسالها لكائن آخر، يتم استخدام التدوين التـالي (بمحـاذاة مربع الدخول أو الخروج):

^الكائن.منهاج (معطيات)

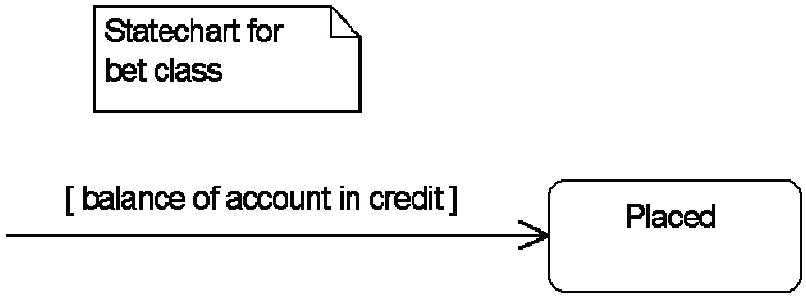
^object.method (parameters)



شكل 100: تدوين يشير إلى أن الرسالة يجب بعثها عند تحول الحالة

دفاعات

أحيانا نحتاج أن نؤكد على أنه ليس بالإمكان تحول الحالة إلا بتلبية شرط معين. لتحقيق ذلك نضع الشرط بين قوسين مربعين كالتالي:



شكل 101: هنا، التحول إلى حالة "Placed" (تم وضع الرهان) لا تتحقق إلا إذا كان رصيد الحساب دائنا

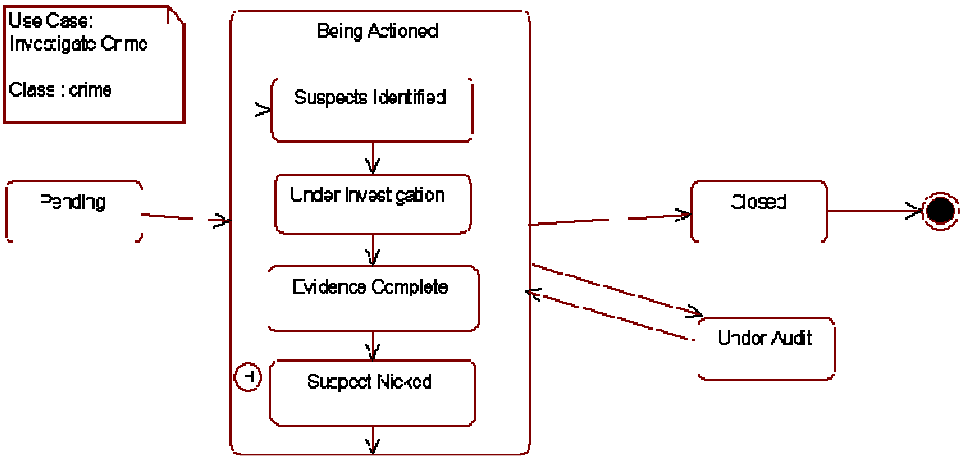
حالات التاريخ

أخيرا، بالعودة إلى الحالات الفرعية، يمكن تدوين أنه إذا تم اعتراض الحالة العليا بطريقة ما، فإنه عند استئناف الحالة العليا، فإن الحالة التي كانت عليها تبقى محفوظة.

لنأخذ المثال التالي. تحقيق جنائي بدأ في حالة توقيف "pending". حالما يتحول إلى حالة "Being Actioned" تم فتح التحقيق، التحقيق يمكن أن يكون في عدة حالات فرعية.

عموما، في أي مرحلة يمكن أن يتم مراجعة القضية. خلال المراجعة يتم تعليق التحقيق مؤقتا. حال الانتهاء من المراجعة، يجب استئناف التحقيق من الحالة التي كان فيها قبل المراجعة.

الترميز البسيط للتاريخ History (حرف "H" في دائرة) يسمح لنا بذلك كالتالي:



شكل 102: حالة تاريخ

استخدامات أخرى لمخططات الحالة

بالرغم من أن الغرض الواضح من هذه المخططات هو تتبع حالة الكائن، ففي الواقع، يمكن استخدام مخططات الحالة لأي عنصر في النظام مؤسس على حالات. أبرزها وقائع الاستخدام (مثلا، واقعة استخدام لا يمكن البدء فيها إلا إذا سجل المستخدم نفسه). حتى حالة كامل النظام يمكن نمذجتها باستخدام رسم الحالات – واضح أن هذا نموذج يفيد "فريق المعمار المركزي" في المشاريع الضخمة.

ملخص

في هذا الفصل، نظرنا في مخططات تنقل الحالة. رأينا:

- صيغة المخطط
- كيف نستخدم الحالات الفرعية
- أفعال الدخول و الخروج
- أحداث الإرسال و الدفاعات
- حالات التاريخ

رسم الحالات بسيطة جدا، لكنها تتطلب غالبا عمليات تفكير عميقة

معظم استخداماتها على الكائنات، لكن بالإمكان استخدامها على أي شيء: وقائع الاستخدام، كامل النظام، إلخ..

الفصل 18: التحول للتوليف

في هذا الجزء المختصر سنتكلم عن بعض القضايا التي تحيط بعملية الانتقال من النموذج إلى التوليف (code). و سنستخدم لغة جافا لعرض الأمثلة لذلك، جافا java لغة بسيطة جدا و يمكن بسهولة تطبيقها على أية لغة حديثة كائنية المنحى.

تحديث (مزامنة) المشغولات* Synchronising Artifacts

إحدى المسائل الرئيسية في التصميم و التوليف هو الاحتفاظ بالنموذج على خط واحد متزامن مع ما يمثله من توليف.

بعض المشروعات تميل إلى فصل التصميم عن التوليف. بحيث يتم بناء التصاميم لتكون مكتملة بأكبر قدر ممكن، و يكون التوليف مجرد عملية تحويل آلية من تلك التصاميم.

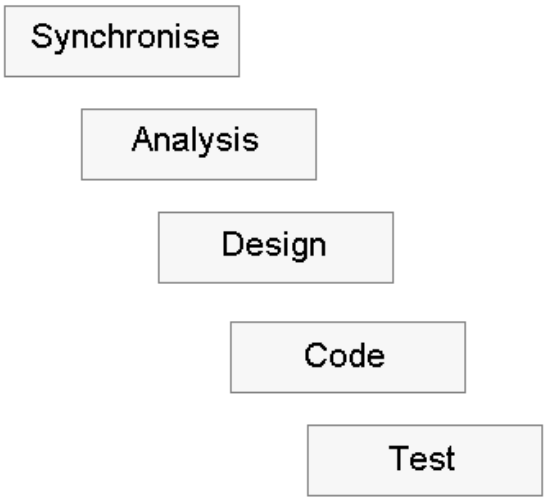
في بعض المشروعات، يتم الإبقاء على نماذج التصميم مرخية بعض الشيء و تؤجل بعض قرارات التصميم النهائية إلى مرحلة التوليف.

في كلتا الحالتين، غالبا ما ينحرف التوليف قليلا أو كثيرا عن النموذج. كيف نتصدى لهذه المشكلة؟

إحدى الحلول هو، إضافة مرحلة أخرى لكل تكرار – مرحلة مزامنة المشغولات. هنا يتم تعديل النماذج لتعكس قرارات التصميم التي تم اتخاذها خلال كتابة التوليف في التكرار السابق.

---

* المشغولة artifact: هي جزئية المعلومة التي يتم إنتاجها، أو تعديلها، أو استخدامها من قبل عملية تطوير برمجي، مساحة المسؤولية فيها محددة، وتكون عرضة لمراقبة نسخها المعدلة. المشغولة قد تكون نموذجا أو عنصر في نموذج أو وثيقة، أو قطعة برمجية.



شكل 103: مرحلة إضافية في النموذج التدفقي – المزامنة

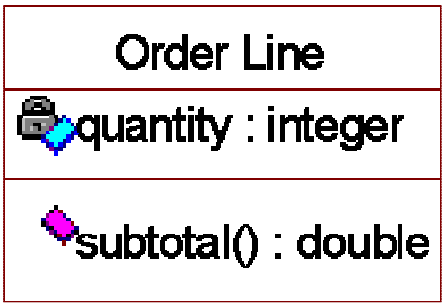
واضح، أنه ليس بالحل السهل، فغالبا، ما تقع تغييرات كبيرة. لكن يمكن المضي في هذا الحل طالما كانت التكرارات قصيرة في مدتها، و طالما أن تعقيد كل تكرار يمكن السيطرة عليه. لكن في النهاية، هذا ما كنا نسعى إليه طول الفترة السابقة!!

بعض أدوات CASE (هندسة البرمجيات بمساعدة الحاسوب) تسمح بأداء "هندسة عكسية" – أي توليد نموذج التصميم من التوليف. هذا قد يساعد في عملية المزامنة – في نهاية التكرار 1، نقوم بإعادة توليد النموذج من التوليف، ثم ننطلق للعمل من خلال هذا النموذج الجديد من أجل التكرار 2 (و نكرر العملية) . لكن يجب القول، أن تقنية الهندسة العكسية ليست متقدمة بعد، لذا فإنها قد لا تناسب كل المشروعات.

ترجمة التصاميم إلى توليف

تعريفات الصنفية في التوليف سوف تأتي من مخطط صنفية التصميم Design Class Diagram. تعريفات المنهج method تأتي معظمها من مخططات التعاون Collaboration Diagrams، كما تأتي مساعدات أخرى من توصيفات واقعة الاستخدام (التفاصيل الإضافية، خصوصا تدفقات الاستثناءات/ البدائل) و رسومات الحالة State Charts (من أجل تتبع حالات وشروط وقوع الأخطاء).

ها هنا مثال لصنفية، وكيف يمكن أن يبدو عليه التوليف:



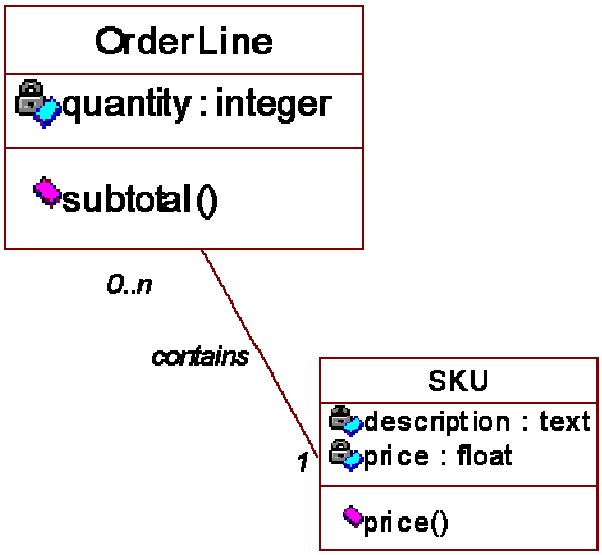
شكل 104: صنفية سطر طلبية، مع مثلين لأعضائها

التوليف الناتج قد يكون كالتالي (بعملية تحويل مباشرة):

```
public class OrderLine
{
   private int quantity;

   public OrderLine(int qty, SKU product)
   {
      // constructor  منشئ
   }
   public double subtotal()
   {
      // method definition تعريف المنهج
   }

}
```

شكل 105: مثال توليف سطر طلبية

لنلاحظ أنه في التوليف أعلاه، تم إضافة المنشئ constructor. تم إهمال منهج create() من مخطط الصنفية (فقد أصبحت كما يبدو عرفا عاما هذه الأيام)، لذا لزم إضافتها في التوليف.



شكل 106: التجمع بين أسطر الطلبية و دليل البند

يحتوي سطر الطلبية على إشارة لبند، لذا يلزم إضافة هذه الإشارة أو سمة مرجعية* للتوليف reference attribute:

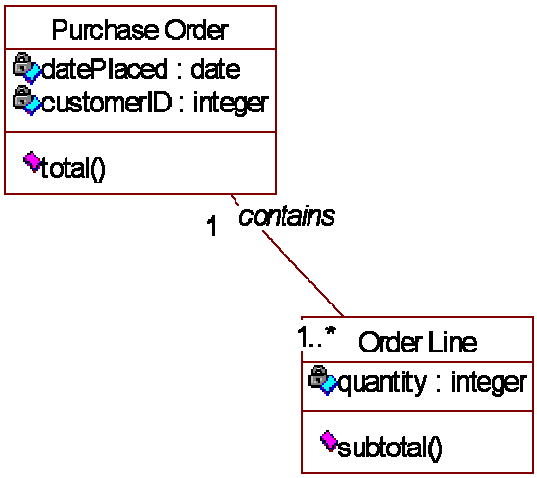
```
public class OrderLine
{
   public OrderLine(int qty, SKU product);
   public float subtotal();

   private int quantity;
   private SKU SKUOrdered;
}
```

شكل 107: إضافة سمة مرجعية (تم تنحية متن المنهج لغرض التوضيح)

ماذا لو احتاجت الصنفية للاحتفاظ بقائمة من الإشارات لصنفية أخرى. المثال الجيد لهذا هو العلاقة بين أوامر الشراء و أسطر الأمر. أمر الشراء "يمتلك" قائمة بالأسطر، كما في مخطط UML التالي:

---

* السمة الرجعية reference attribute هي سمة تشير إلى كائن مركب أخر، ليس نوعا ابتدائيا مثل نص String، أو رقم Number، و ما إلى ذلك. في مخطط صنفية التصميم، عادة تكون السمة المرجعية معرفة ضمنا و لا ينص عليها صراحة. (المترجم)



شكل 108: أمر الشراء يحوي قائمة بأسطر الأمر

التنفيذ الفعلي لهذا على متطلبات معينة (مثلا، هل يجب أن تكون القائمة مرتبة، هل سرعة الأداء مهمة، وهكذا)، لكن بافتراض أننا نحتاج لمصفوفة بسيطة، التوليف التالي سيكون كافيا:

```
public class PurchaseOrder
{
   public float total();

   private date datePlaced;
   private int  customerID;
   private Vector OrderLineList;
}
```

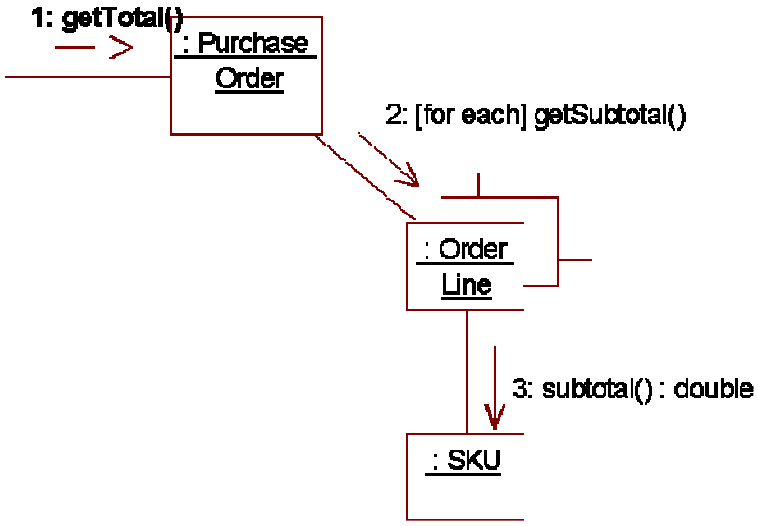
شكل 109: إضافة قائمة بالإشارات

تمهيد القائمة سيكون من مهمة المنشئ constructor. لمن لا يبرمجون بجافا أو سي++، النوع: Vector هو ببساطة مصفوفة array يمكن إعادة تغيير حجمها، و بحسب المتطلبات، قد يكفي استخدام مصفوفة نمطية.

تحديد المناهج Methods

مخطط التعاون هو المصدر الأكبر لتعريفات المنهج أو المنهاج method.

المثال التالي يصف منهج أخذ الإجمالي "get total" لأمر الشراء. هذا المنهج يرجع التكلفة الإجمالية لكافة أسطر بنود الأمر:



شكل 110: تعاون "Get total" أخذ الإجمالي

خطوة 1

لدينا منهج يسمى "getTotal()" في صنفية أمر شراء:

```
public double getTotal()
{

}
```

شكل 111: تعريف منهج في صنفية أمر شراء Purchase Order

خطوة 2

مخطط التعاون يشير إلى أن صنفية أمر الشراء الآن تقوم بإحصاء كل سطر:

```
public double getTotal()
{
  double total;
   for (int x=0; x<orderLineList.size();x++)
   {
      // extract the OrderLine from the list
      theLine = (OrderLine)orderLineList.get(x);

      total += theLine.getSubtotal();
   }
   return total;
}
```

شكل 112: توليف للحصول على الإجمالي، بإحصاء كل أسطر أمر الشراء التابعة للأمر.

خطوة 3

قمنا باستدعاء منهج يسمى "()getSubtotal" في صنفية سطر الأمر. لذلك نقوم بتنفيذه.

```
public double getSubtotal()
{
   return quantity * SKUOrdered.getPrice();
}
```

شكل 113: تنفيذ ()getSubtotal

خطوة 4

قمنا باستدعاء منهج يسمى "()getPrice" في صنفية البند SKU. هذا أيضا يحتاج لتنفيذ، و سيكون منهجا بسيطا يقوم بترجيع قيمة لسمة خصوصية private.

ترجمة الحزم لتوليف

أكدنا فيما سبق أن بناء الحزم packages يعد جانبا أساسيا في معمار النظام، لكن كيف نقوم بترجمتها إلى توليف.

بلغة جافا

إذا كنت تكتب بلغة جافا، فإنها تدعم الحزم packages بطريقة مباشرة. في الواقع، كل صنفية في جافا تتبع لحزمة. أول سطر في تعريف الصنفية يجب أن يحدد لجافا في أية حزمة يتم وضع هذه الصنفية (إذا تم تجاهل هذا، سيتم وضع الصنفية في الحزمة المبدئية "default" عفوية).

فإذا كانت صنفية SKU ستوضع في حزمة تدعى "Stock"، فإن الترويسة التالية للصنفية ستكون صحيحة:

```
package com.mycompany.stock;

class SKU
{ ...
```

شكل 114: وضع الصنفيات في حزم

أفضل من هذا، تقوم جافا بإضافة مستوى آخر من المنظورية visibility فوق المنظوريات النمطية: private، public و protected (خصوصي، عمومي و محمي). تقوم جافا بتضمين بما يعرف بحماية الحزمةpackage protection . بحيث يمكن تعريف الصنفية لتتكون منظورة فقط للصنفيات التي في نفس الحزمة – و بالطبع للمناهج التي داخل كل صنفية. هذا يوفر دعما ممتازا لفكرة التغليف encapsulation داخل الحزم. بجعل كل الصنفيات مرئية فقط للحزم التي تحويها (عدا الواجهات facades) ، و بهذا يمكن فعلا تطوير النظم الفرعية بطريقة مستقلة.

للأسف، صيغة syntax حماية الحزمة في جافا ضعيفة على نحو ما. تدوينها يتم ببساطة بتصريح declare الصنفية بدون أن يسبقها أيا من تعريفات الصنفية: private، public و protected – تماما مثل الشكل 114.

بلغة سي++

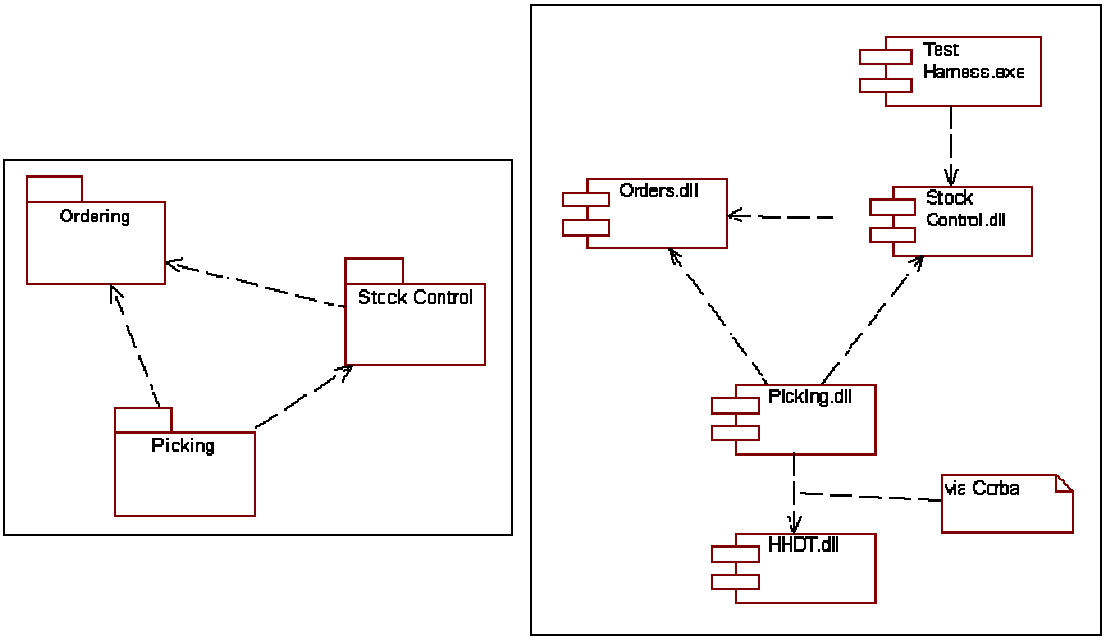
لا يوجد دعم مباشر للتحزيم في سي++، لكن أخيرا تم إضافة مفهوم نطاقات الأسماء namespaces إلى اللغة. هذا سيسمح بوضع الصنفيات في تقسيمات منطقية منفصلة ، لتفادي تعارض الأسماء بين نطاقات الأسماء (يمكننا بذلك إنشاء اثنين من نطاقات الأسماء، مثل Stock و Orders، كل منهما يحوي على صنفية تسمى SKU).

هذا يعطي بعض الدعم للتحزيم، لكن لسوء الحظ لا يوفر أية حماية عبر المنظوريات visibilities. فالصنفية في أحد نطاقات الأسماء يمكنها النفاذ لكل الصنفيات العمومية في نطاقات أسماء أخرى.

نموذج المكونات في UML

يتم في هذا النموذج عرض خريطة المكونات components المادية "الصلبة" للبرمجيات (مقابل الصورة المنطقية المعبر عنها في مخطط التحزيم).

برغم أن النموذج عادة ما يؤسس عل مخطط التحزيم المنطقي، إلا أن بإمكانه أن يحوي عناصر مادية لازمة للتشغيل و لم تكن مهمة في مرحلة التصميم. مثلا، المخطط التالي يعرض مثالا لنموذج منطقي، متبوعا بنموذج البرمجيات المادية النهائية.



شكل 115: الصورة المنطقية مقارنة بالصورة المادية

نموذج المكونات بسيط جدا. و يعمل بنفس طريقة عمل مخطط التحزيم، عارضا العناصر و الاعتماديات فيما بينها. في هذه المرة الرموز تختلف، أي مكون قد يكون أية كينونة برمجية مادية (ملف تنفيذي، ملف مكتبة حيوي DLL، ملف هدفي object، ملف مصدري، أو أيا ملف كان).

ملخص

في هذا الفصل تم بصورة مختصرة وصف العمليات الرئيسية لتحويل النماذج إلى توليف حقيقي. تعرفنا بإيجاز على مشكلة الإبقاء على النموذج على خط متواز مع التوليف، ونظرنا في فكرتين لحل هذه المشكلة.

رأينا نموذج المكونات. و كيف تساعد على وضع خريطة للعناصر المادية للبرمجيات و الاعتماديات فيها.

قائمة المراجع

[1] : Krutchten, Philippe. 2000 The Rational Unified Process An Introduction Second Edition Addison-Wesley

العملية الموحدة من راشيونال – مقدمة – طبعة ثانية

مقدمة مختصرة للعملية الموحدة من راشيونال، وعلاقتها بلغة UML.

[2] : Larman, Craig. 1998 Applying UML and Patterns An Introduction to Object Oriented Analysis and Design Prentice Hall

تطبيق UML و الأنماط – مدخل للتحليل و التصميم كائني المنحى

مدخل ممتاز ل UML، و تطبيقها في عمليات تطوير واقعية للبرمجيات. أستخدم كأساس لهذا الكتاب.

[3] : Schmuller, Joseph. 1999 *Teach Yourself UML in 24 Hours* Sams

علم نفسك UML في 24 ساعة

مقدمة شاملة حول UML، النصف الأول يركز على صيغ UML، و الثاني يعرض كيفية تطبيق UML (باستخدام عملية تشبه العملية الموحدة و تسمى GRAPPLE)

[4] : Collins, Tony. 1998 Crash: Learning from the World's Worst Computer Disasters Simon&Schuster

انهيار: التعلم من أسوأ كوارث الحاسوب في العالم

باقة ممتعة من حالات دراسية تستكشف سبب فشل العديد من المشاريع لتطوير وبناء البرمجيات

[5] : Kruchten, Phillipe 2000 From Waterfall to Iterative Lifecycle - a tough transition for project managers Rational Software Whitepaper – www.rational.com

من دورة حياتية تدفقية إلى تكرارية – انتقال صعب لمدراء المشاريع

وصف ممتاز و قصير للمسائل التي ستواجه مدراء المشاريع في المشاريع التكرارية

[6] : Gamma, E., Helm, R., Johnson, R., Vlissides, J. 1995 Design Patterns : Elements of Reusable Object Oriented Software Addison-Wesley

أنماط التصميم: عناصر قابلية إعادة الاستخدام في البرمجيات كائنية المنحى
الفهرس المميز من "عصابة الأربعة" عن مختلف أنماط التصميم

[7] : Riel, Arthur 1996 *Object Oriented Design Heuristics* Addison-Wesley

استكشافات التصميم كائني المنحى
إرشادات مجربة للمصممين بالمنحى للكائن

[8] : UML Distilled
Martin Fowler's pragmatic approach to applying UML on real software developments

مختصر UML
مقاربة عملية لتطبيق UML على عمليات تنشئة واقعية للبرمجيات

[9] : Kulak, D., Guiney, E. 2000 *Use Cases : Requirements in Context* Addison-Wesley

وقائع الاستخدام: المتطلبات في سياقها
معالجة متعمقة لهندسة المتطلبات، المقادة بوقائع الاستخدام.