Use Case Diagram (Диаграма на потребителските случай)

В процеса на проектиране на софтуерния продукт, диаграмата на потребителските случаи е първата диаграма, която се създава от проектантите, когато се започне проект. Тази диаграма позволява да се опишат на най-високо ниво целите на потребителя, които системата трябва да изпълнява. Тези цели не е необходимо да са задачи или действия, а може да са по общи изисквания към функционалността на системата. С други думи това е техника за определяне на функционалните изисквания на една система. Те описват типичните взаимодействия между потребителите и системата, предоставят описание на начина, по който тя се използва.

Use Case - Примери за използване

Примерите за използване (Use-cases) са основани на сценарий техника в UML, която идентифицира актьорите в едно взаимодействие и която описва самото взаимодействие.
Множество от примери на използване би трябвало да опише всички взаимодействия със системата.
Диаграми на последователностите могат да се използват, за да добавят подробности на примерите за използване като показват последователността от обработки на събития в системата.

Use case е съставен от множество от **сценарии**.

Сценарии

Сценариите са примери от реалния живот, как системата може да се използва.
Те трябва да включват:

- Описание на началната ситуация
- Описание на нормалния поток от събития
- Описание на това, което не е както трябва
- Информация за други дейности, извършвани в същия момент
- Описание на състоянието, в което сценарият завършва

Пример 1: Дефиниране на сценарии за библиотечна система

Начални предположения: Потребител се е идентифицирал (логнал) в системата и е намерил списание, съдържащо статията.

Нормално протичане: Потребителят маркира статията за копиране. Той (тя) е подканен от системата или да подаде информация за абонамент на списанието или да укаже как ще плати за статията. Алтернативни методи за плащане са кредитна карта или указване на номер на акаунт на организация.

След това потребителят е помолен да попълни формуляр, който съдържа подробности за транзакцията, която след това е изпратена на системата.

Формулярът се проверява и ако е ОК, PDF версия на статията се натоварва в работната област на потребителския компютър и потребителят се уведомява, че тя е на разположение. Потребителят е да маркира принтер и статията се отпечатва. Ако статията е маркирана като ‘print-only’, то тя се изтрива веднага след завършване на отпечатването.

Какво може да се случи: Потребителят може да не успее да попълни правилно формуляра. В този случай формулярът трябва му се представи отново. Ако и вторият път формулярът е неправилен, то искането на потребителя се отхвърля.

Плащането може да бъде отказано от системата. Искането на потребителя се отхвърля.

Свалянето на статията може да не е успешно. Опитите се повтарят до успех или до край на сесията.

Може да не е възможно отпечатването на статията. Ако статията не е маркирана като ‘print-only’, тя се оставя в работната област. В противен случай се изтрива и потребителската сметка се кредитира със стойността на статията.

Други дейности: Едновременно сваляне на други статии.

Състояние на системата в края: Потребителят е логнат. Свалената статия е изтрита от системата, ако е маркирана като 'print-only'.

Пример 2: Дефиниране на сценарии на покупка от Web базиран онлайн магазин:

Сценарий 1: Клиентът преглежда каталога и добавя желаните артикули в кошницата. Когато пожелае да плати, предоставя информация за доставката и данни от кредитната си карта и потвърждава плащането. Системата проверява оторизацията на картата, потвърждава продажбата и изпраща потвърждаващо e-mail съобщение.

Сценарий 2: В случай, че оторизацията на кредитната карта е неуспешна.

Сценарий 3: Редовен клиент, от който не се изисква информация за доставката и данни от кредитната карта.

Всички сценарии са различни въпреки приликата между тях, но сходството им се състои в това, че потребителят има една и съща цел – да закупи продукт. Потребителят не винаги успява да го направи, но целта остава. Тази цел е ключът към потребителските случаи – набор от сценарии, свързани помежду си от обща потребителска цел.

Описание на сценариите – представляват серия от стъпки, описващи взаимодействието между потребител и система.

Всеки сценарий е поредица от стъпки, описващи взаимодействието между потребител и система.

Основен сценарий на успех:

1. Клиент преглежда каталога и избира артикули за покупка.
2. Клиента преминава към потвърждаване на покупката.
3. Клиента попълва информацията за доставка (адрес, спешна или 3-дневна).
4. Системата представя пълна информация за цената, включително разходите за доставка.
5. Клиента попълва данните от своята лична карта.
6. Системата оторизира покупката.
7. Системата потвърждава незабавно продажбата.
8. Системата изпраща на клиента потвърждаващо съобщение по e-mail.

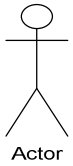
Разширения:
3а: Клиентът е редовен клиент
1. Системата показва текущата информация за доставка, цена и разплащане.
2. Клиента може да приеме тези подразбиращи се стойности или да дефинира нови; следва връщане към основния сценарий на успех и стъпка 6.

6а: Системата не успява да оторизира покупката с кредитна карта.
1. Клиента може да въведе отново своите данни от кредитната карта или да анулира покупката.

Пример 3: Use case за търсене по ключова дума:
1. Клиентът въвежда ключовата дума в полето за търсене.
2. Клиентът натиска бутона за търсене.
3. Търсенето се изпълнява.
4. Резултатите се показват.

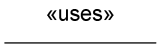
Алтернатива:
Ако търсенето пропадне в стъпка 3, тогава потребителят е препратен за ново търсене – екран от стъпка 1.

Компоненти при Use case диаграми:

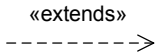


Актьор

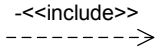
Случай на употреба



Асоциация



Разширение

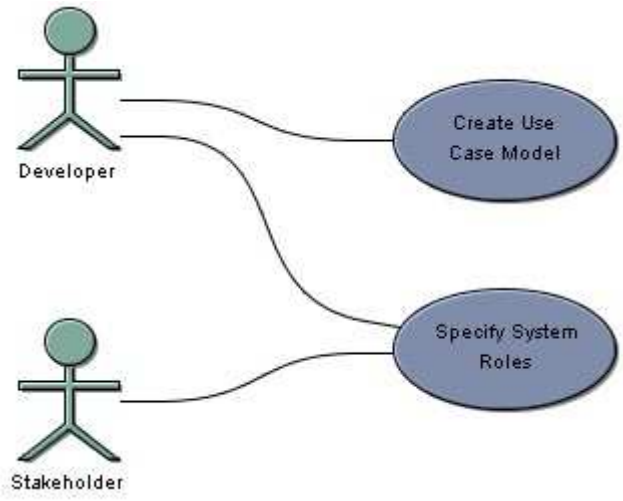


Включва

Обобщение

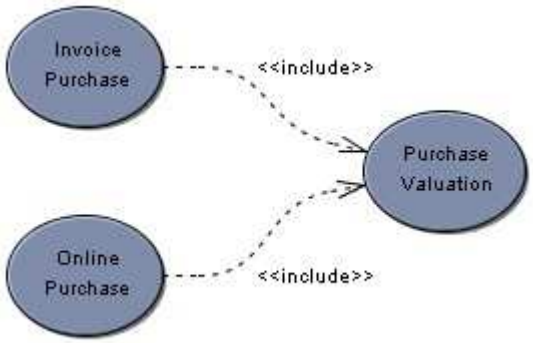
Actor (Актьор) – роля, която един потребител играе по отношение на системата. Може да бъде клиент, представител по поддръжка на клиенти, мениджър продажби, продуктов аналитик. Актьорите изпълняват случаите на употреба. Един актьор може да изпълнява много случаи на употреба и един случай на употреба може да се изпълнява от много актьори. Актьора клиент може да представлява множество хора. Един човек може да играе ролята на няколко актьора. Актьора може да не е човек, а друга компютърна система, ако системата извършва услуга за нея.

Associations – плътна връзка без стрелка. **Associations** са връзка между **actors** и **use cases** и означава, че актьора осъществява потребителския случай:

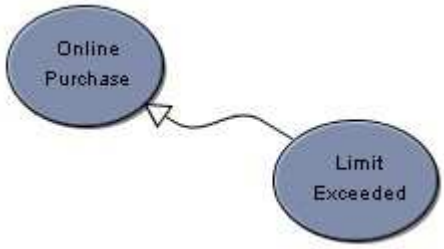


В горната диаграма човечетата са актьорите, а елипсите – потребителски случаи. И двамата потребители разработчик и спонсорът са отговорни за определяне на ролите на системата, но само разработчика създава модела.

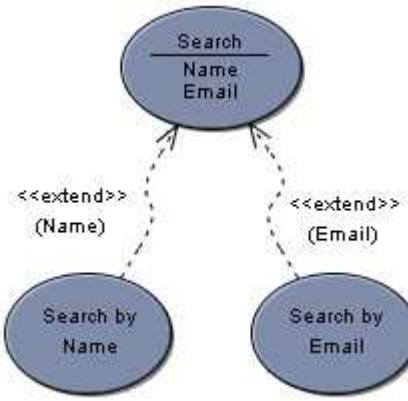
Includes – Връзка показваща, че един Use sace включва друг Use case. Примера илюстрира как и покупката с фактура и покупката онлайн се включват в сценария описан от покупателната оценка. В обобщение, includes връзката е да отстрани повторенията на сценария в съставните случаи. Може да се каже, че includes е като извикване на процедура.



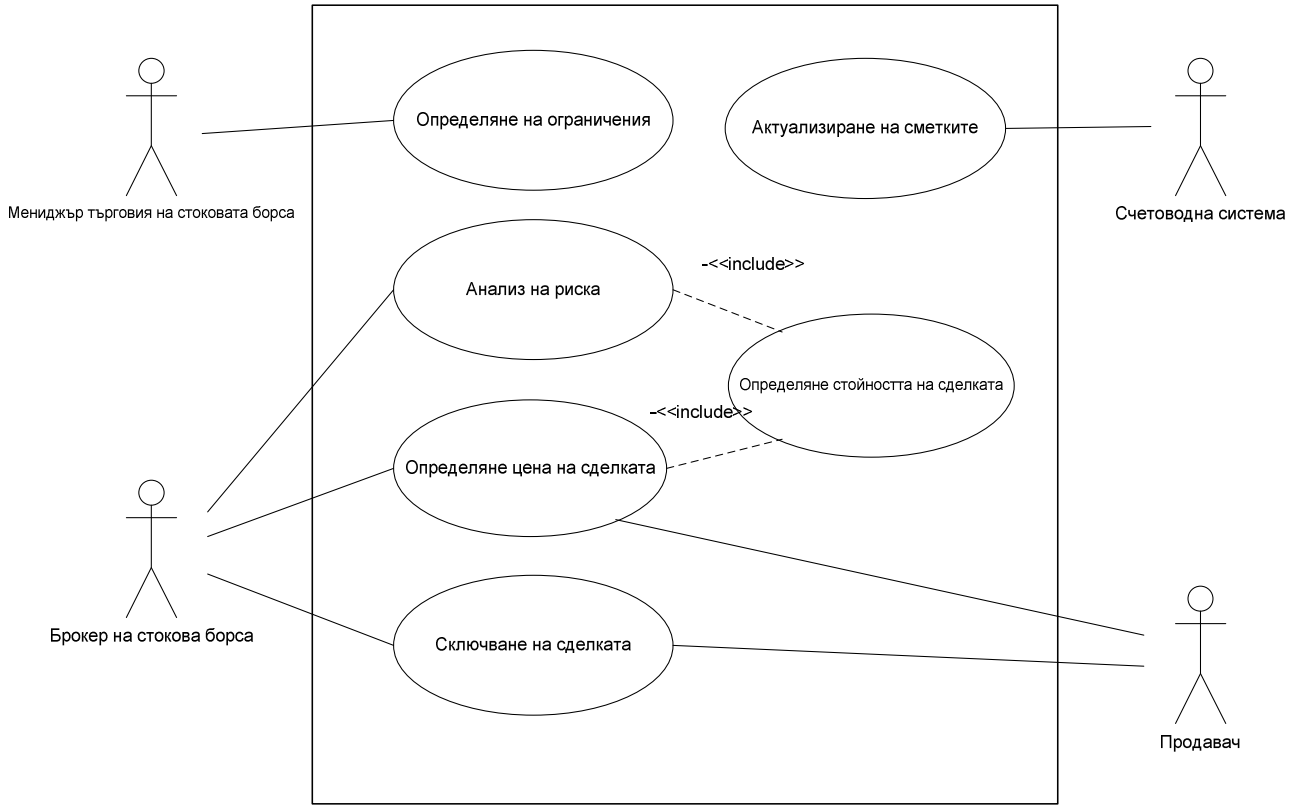
Generalization. Когато един потребителски случай описва разновидност на друг потребителски случай се използва връзката **generalization**. Тя е специализация на някои случаи. Целта трябва де бъде една и съща за потребителските случаи.



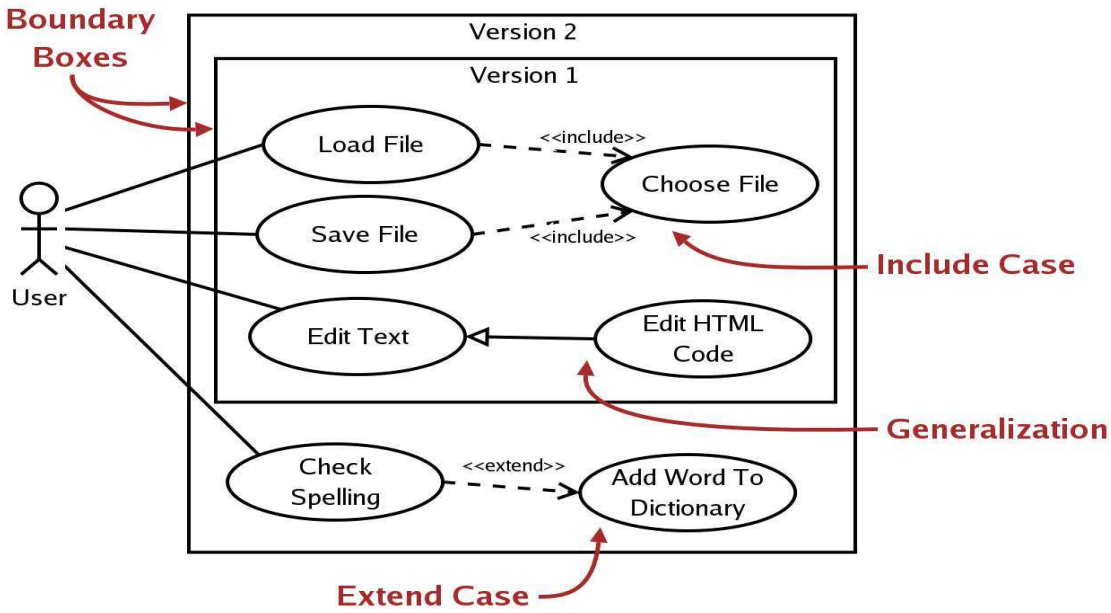
Extends. В някой случаи е възможно да се опишат различията в поведението. Тогава трябва да се дефинира **extension points** в разширения **потребителския случай**. Може да се каже, че extend е като извикване на процедура, която се извиква понякога, в зависимост от някакво условие.



Пример за използване на Use case диаграма за описание на потребители и случаите на употреба:



Пример за Use case диаграма на текстов редактор:

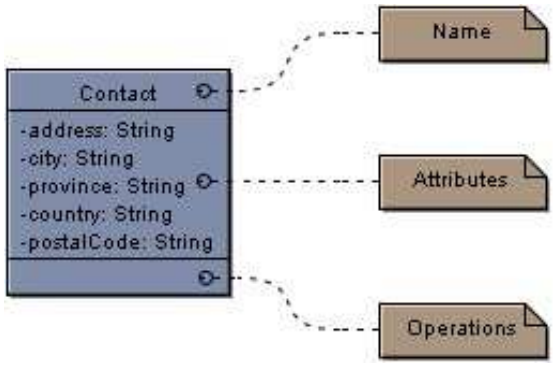


Class Diagram

Диаграмите на класове са в основата на обектно-ориентираното проектиране. Те описват типовете на обектите в системата и статичните връзки между тях.

Показва множеството класове, интерфейси и взаимодействията между тях. Основният елемент на диаграмата на класовете е класът. В обектноориентираното програмиране класовете се използват за представяне на обектите в системата. Те описват обектите от реалния свят.

Класът *Contact* е пример на прост клас, който съхранява информация за място.



Класовете се представят като множество обекти с общи атрибути, операции, връзки. Интерфейсът е колекция от операции, задаващи поведение.

Класът е разделен на три секции:

Име на класа

Атрибути на класа – характеристиките на класа. В примера класа *Contact* съдържа атрибутите address, city, province, country и postal code.

Формат на атрибутите:

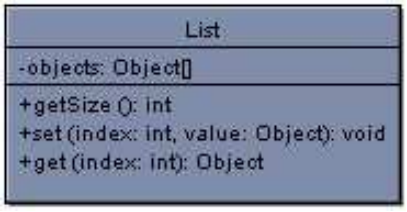
visibility name: type = defaultValue

Видимост на атрибутите:

- **-** Private. Атрибутите са видими само за операциите на класа
- **+** Public. Атрибутите са видими извйн класа и са част от интерфейса
- **#** Protected. Атрибутите са видими само за класовете, наследяващи текущия
- **~** Package

В обектно-ориентираното програмиране е препоръчително да се съхраняват атрибутите като private и да се използват методите за контролиране на достъпа до данните.

Операции на класа – списъка на операциите на класа представя функции или задачи, които могат да бъдат изпълнени с данните на класа.



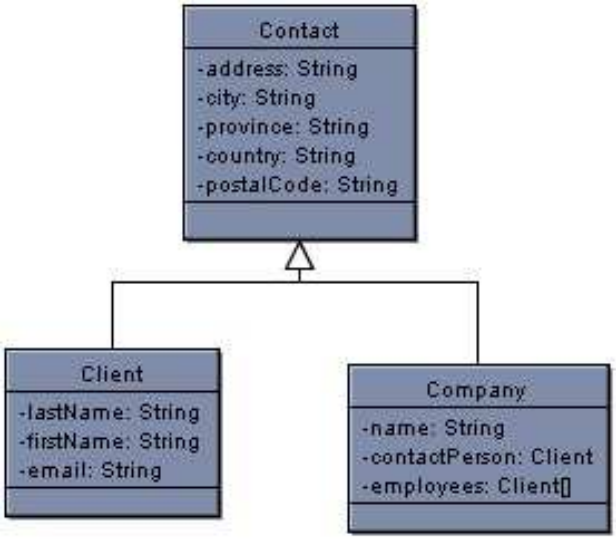
В класа *List* има един атрибут (private array of Objects) и три операции

Формата за операциите на клас е:

visibility name (parameters): type

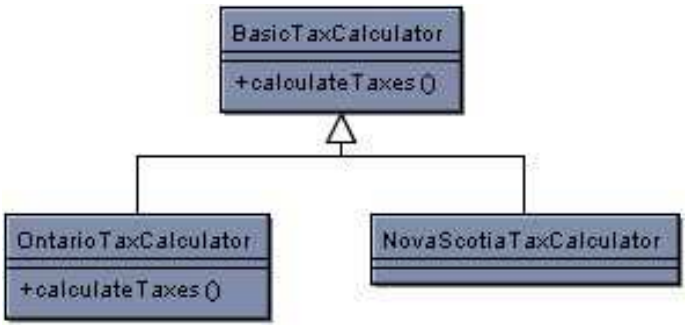
Generalization

Generalization връзката между два класа се използва, за да покаже, че един клас включва всички атрибути и операции на друг клас, но добавя към тях допълнителни атрибути и операции.



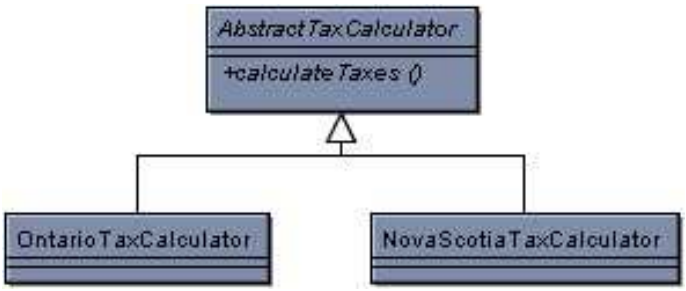
На диаграмата е показан класа *Contact* с два подкласа. Може да се каже, че класовете *Client* и *Company* **inherit**, **generalize** или **extend** *Contact*. За *Client* и *Company* всички атрибути на *Contact* (address, city, и т.н.) съществуват, но са добавени и допълнителни. Класа *Contact* може да се каже, че е **superclass** на *Client* и *Company*.

Когато се използва връзка generalization, класовете наследници могат да предефинират операциите на базовия клас – операции, които са дефинирани в базовия клас, но дефинират нова реализация.



Например *OntarioTaxCalculator* предефинира метода на BasicTaxCalculator. Всъщност кода е различен, но метода се извиква по същия начин.

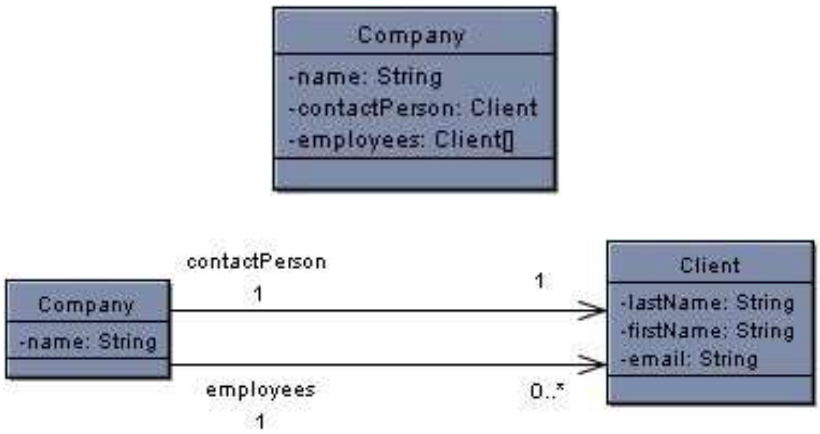
Възможно е да се дефинира метод в суперкласа като абстрактен (**abstract**). Ако класа има абстрактни операции, то и класа е абстрактен. Абстрактните методи и класове се означават като курсивни (italic). Не всички операции на абстрактния клас са абстрактни.



Абстрактната операция *calculateTaxes* в *AbstractTaxCalculator* трябва да бъде имплементирана в класовете наследници OntarioTaxCalculator и NovaScotiaTaxCalculator.

Associations

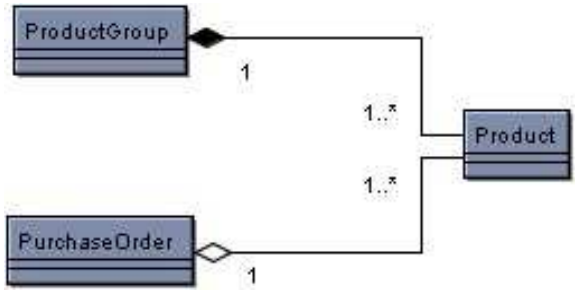
Класовете могат също да съдържат връзки към друг клас. Класът *Company* class има два атрибута, които са свързани с класа Client.



Първата асоциация представя атрибута *contactPerson*e. Съществува един *contactPerson*e в един *Company*. Това означава, че всяка компания има само едно лице за контакти и за всяко лице за контакти има само една компания. Втората асоциация описва съществуването на нито един или много служители в компанията.

0	0
1	1
1..*	1 или много
1..2, 10..*	1, 2 или 10 и нагоре, но не от 3 до 9

Aggregation and Composition



Примера показва асоциациите **aggregation** и **composition**.

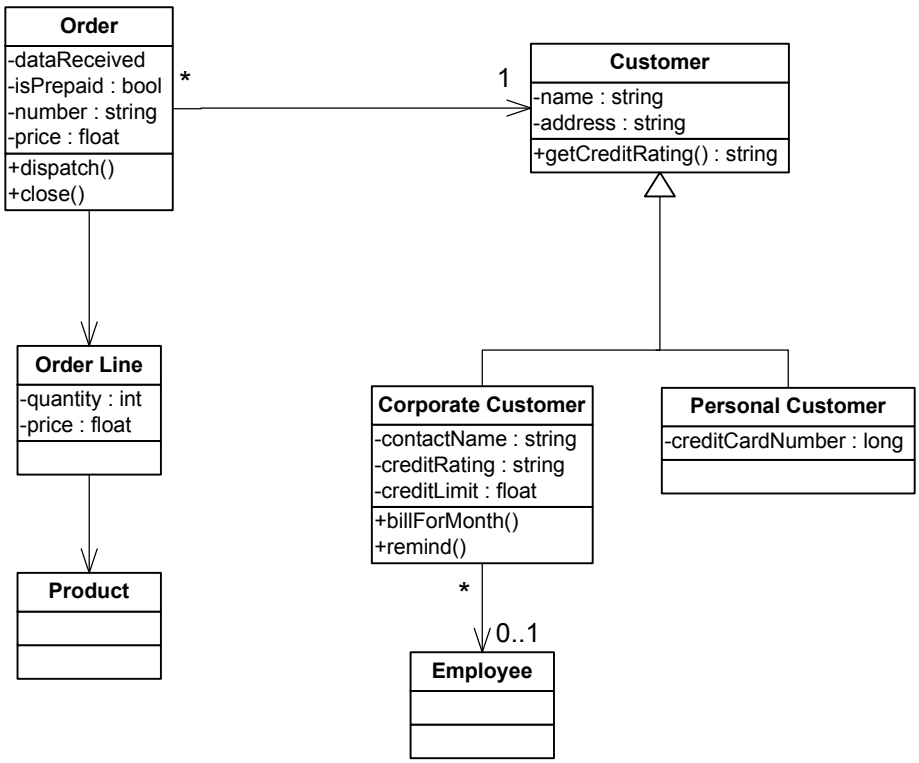
ProductGroup е **съставен от** *Products*. Ако *ProductGroup* се унищожи, се унищожава *и Products*

PurchaseOrder е съвкупност от *Products*. Ако *PurchaseOrder* се унищожи, *Products* съществува.

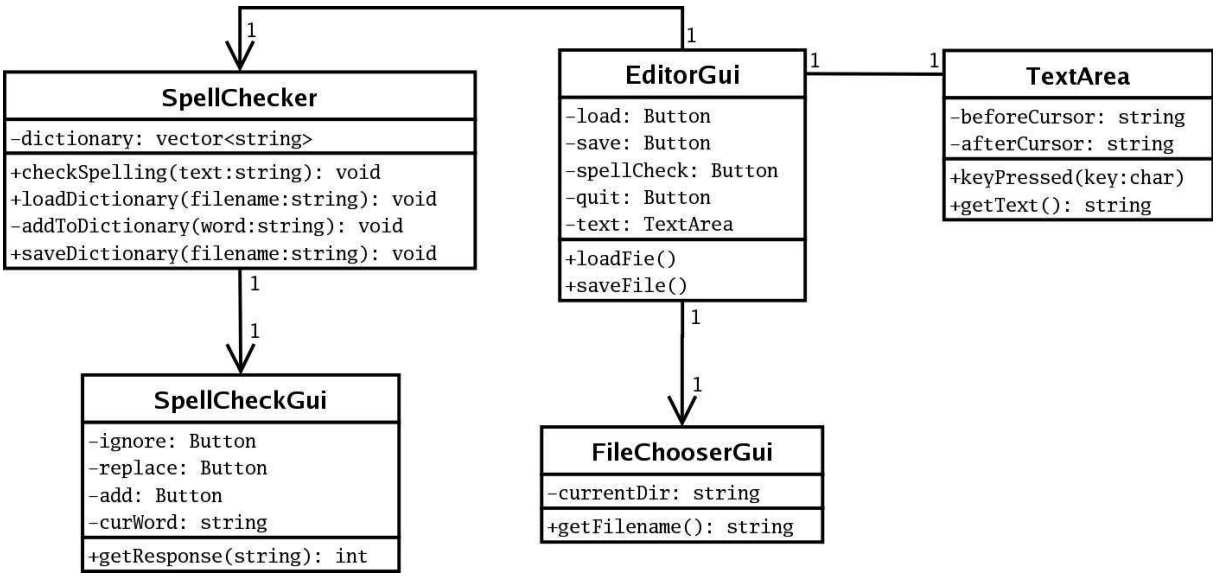
Зависимост (Dependencies)

Съществува зависимост между два елемента, ако промяната в единия има ефект върху другия. Например, ако един клас извиква операция на друг клас, тогава съществува зависимост между двата класа. Цел при проектирането на системата е да се сведат до минимум зависимостите

Пример за диаграма на класове



Пример за диаграма на класовете на текстов редактор:



Пакети (Packages)

Пакетите позволяват групиране на обектите. В много обектноориентирани езици (като Java), пакетите се използват за представяне на класовете и интерфейсите.

Показва как системата се разделя на логически групи, чрез изобразяване на зависимостите между тях. Името на пакета може да е просто или уточнено, ако той е вложен в друг пакет. Пакетът може да съдържа classes, interfaces, components, nodes, collaborations, use cases, diagrams и други packages. Дефинира собствено пространство на имената.

Видимостта на елементите в пакета се контролира както при класовете:

- protected се означава с #. Елементите protected са видими само за пакети, наследяващи текущия.
- Private се означава с –. Елементите private са невидими извън пакета.
- Public се означава с +. Елементите public се включват в интерфейсната част на пакета.

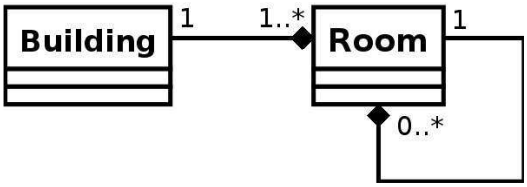
Пакетните диаграми съдържат множество пакети и връзките между тях.

Различаваме 3 вида диаграми на пакети:

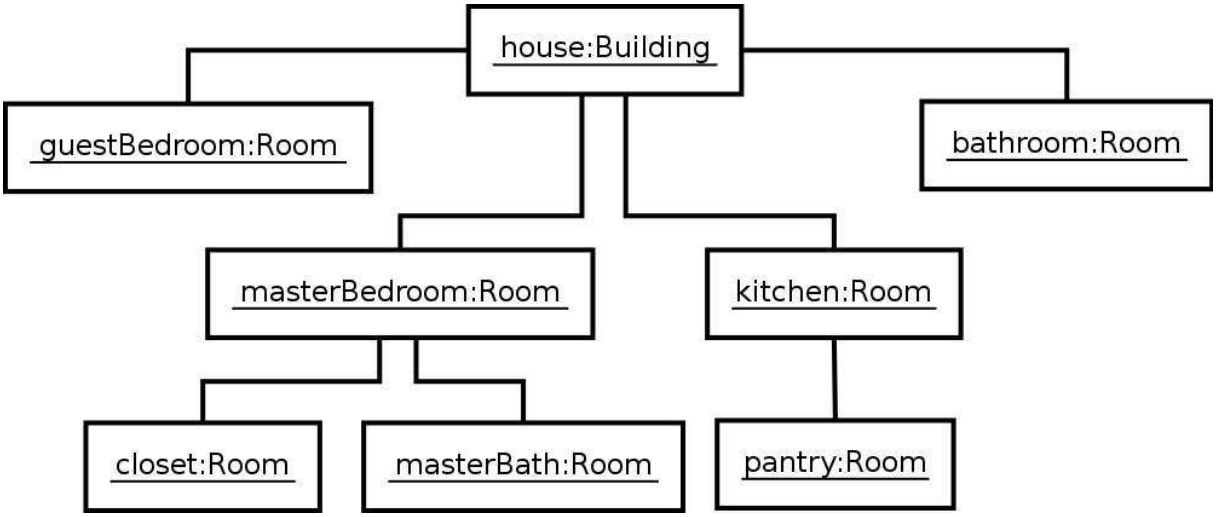
- Class Package Diagrams
- Data Package Diagrams
- Use Case Package Diagrams

Object Diagrams

- Object Diagrams показват инстанциите на класа и техните връзки
- Използват се за изясняване на комплексни връзки



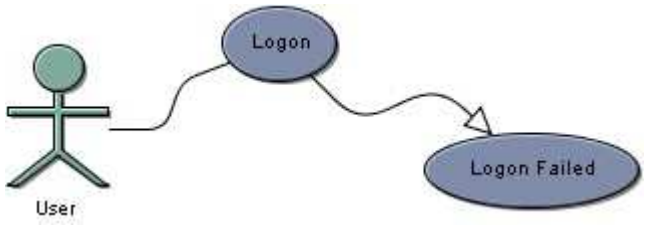
Пример за диаграма на обекти, как са свързани инстанциите на класовете.



Диаграми на взаимодействие - Sequence and Collaboration

След като вече са определени случаите на употреба и са моделирани обектите в системата чрез диаграми на класове, може да се разработи динамичното поведение на системата. Диаграмите на взаимодействие описват начина, по който групи от обекти постигат съвместно някакво поведение.

Случаите на употреба обхващат взаимодействието между потребителите и системата. Диаграмата на взаимодействието описва поведението на един единствен случай на употреба, показвайки сътрудничеството на обектите в системата. Тази диаграма показва обектите в системата и съобщенията, които са разменят между тях.



Пример: Потребител се логва в системата. Случаят на употреба се описва със следните стъпки:

1. Показва се диалога за логване
2. Потребителя въвежда потребителско име и парола
3. Потребителя натиска бутона OK или клавиша enter
4. Потребителското име и парола се проверяват и потвърждават
5. Потребителя се допуска до системата

Алтернатива: Logon Failed – ако на стъпка 4 не са потвърдени потребителското име и парола, допуска се потребителя да опита отново.

Класове в системата

Класът *LogonDialog* има public методи за показване и скриване на прозорец и private метод, който се извиква, когато потребителя натисне бутона КО или клавиша enter.

Клас *LogonManager* включва метод, който връща true, ако логването е успешно и false, ако не е.

Клас *DatabaseAccess* ще позволи да се изпълни заявка към базата данни.

След като са моделирани класовете, може да се направят диаграмите на взаимодействието.

Инстанции и съобщения

Диаграмите на взаимодействието са композирани главно от инстанции (обекти) на класовете и съобщения между тях. **Инстанция** е реализацията на клас. Ако ние имаме клас *Доктор*, инстанции са *Д-р Джоунс*, *Д-р Смит*, и т.н.

В UML, инстанциите се представят с правоъгълник, в който се записва:

instanceName: datatype

Може да се зададе име на инстанцията или да не се зададе, но винаги трябва да се зададе datatype.

Под името може също така да се изброят атрибутите и техните стойности, специфични за този случай на употреба. Това се прави когато стойностите са много важни.

Под името, вие можете също така да се изброят атрибутите и техните стойности. В Визуалния Случай, вие можете да изобразите атрибути от вашия клас и влизате в новите стойности специфични за този пример. потребност на Атрибути само бъде показана когато те са важни и вие нямате да определите точно и показвате всички от атрибутите на класик

Съобщенията представляват извиквания на операциите. Ако инстанция извиква операция, се изпраща съобщение. Също така при завършването на операцията се връща обратно съобщение към инстанцията, която е инициирала повикването.

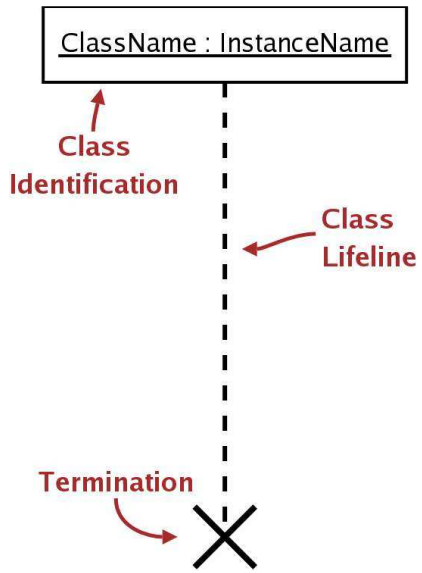
Sequence

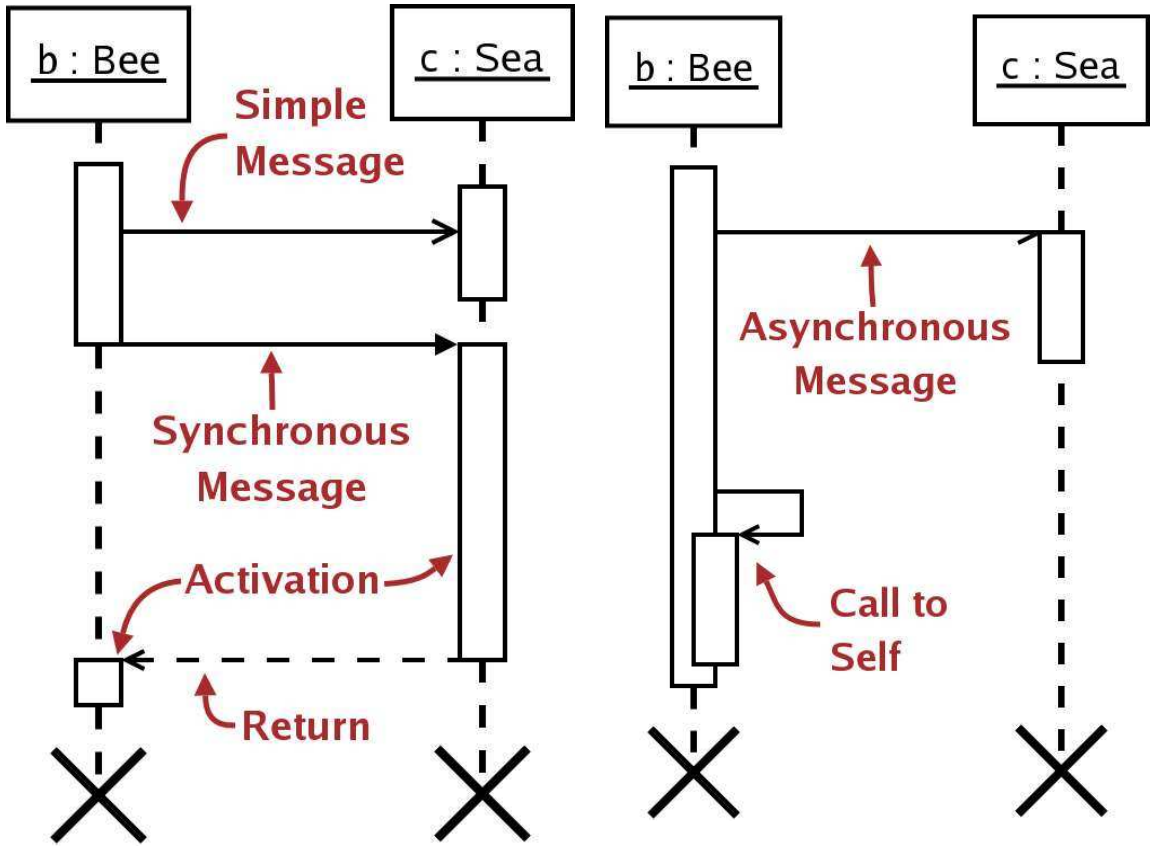
Диаграма на последователностите се използва, за да се разгледа поведението на няколко обекта в рамките на единствен случай на употреба. Те показват добре съвместната работа между обектите. Диаграмата на последователностите показва реда, в който се случват нещата, като последователност от съобщения.

- Потокът на време е показан отгоре до долу, това означава, че съобщенията по-високо на диаграмата се случват преди тези, които са по-ниско
- Показва кое съобщение се изпраща от един обект към друг обект и кога се изпраща.
- Правоъгълниците са **инстанции** на представените класове и вертикалните линии са линиите на живот
- Хоризонталните линии показват класовете или техните роли.
- Стрелките (връзки) са **съобщения** - повиквания на операции и връщане от операции
- Съобщенията са означени с операциите, които се изпълняват. Може да се задават параметрите.

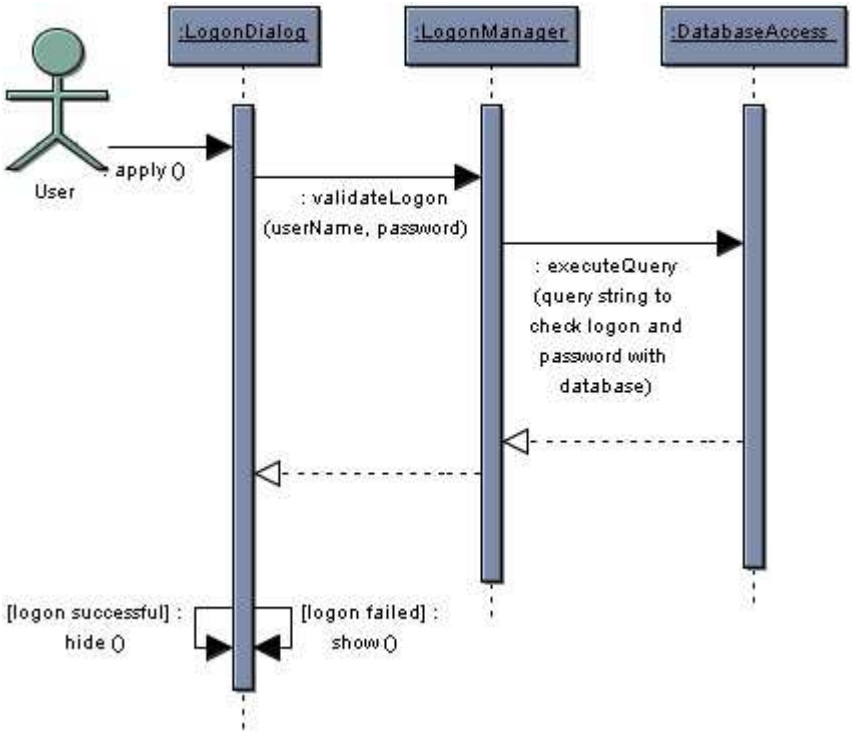
– *Class Identification* – правоъгълник с подчертано име на инстанция и клас *InstanceName : ClassName*.
– *Class Lifeline* – прекъсната линия, показваща съществуването на обект.
– *Termination* - с X в края се означава разрушаването на обект.

- *Activation* – Правоъгълник върху линията на обекта означава активирането на обекта.
- *Simple message* – линия със стрелка, показваща съобщенията мейду обектите.
- *Syncronous message* - линия с плътна стрелка, показваща синхронните съобщения.
- *Asynchronous message* - линия с половин стрелка, показваща асинхронните съобщения.
- *Call to self* – обект извиква себе си.

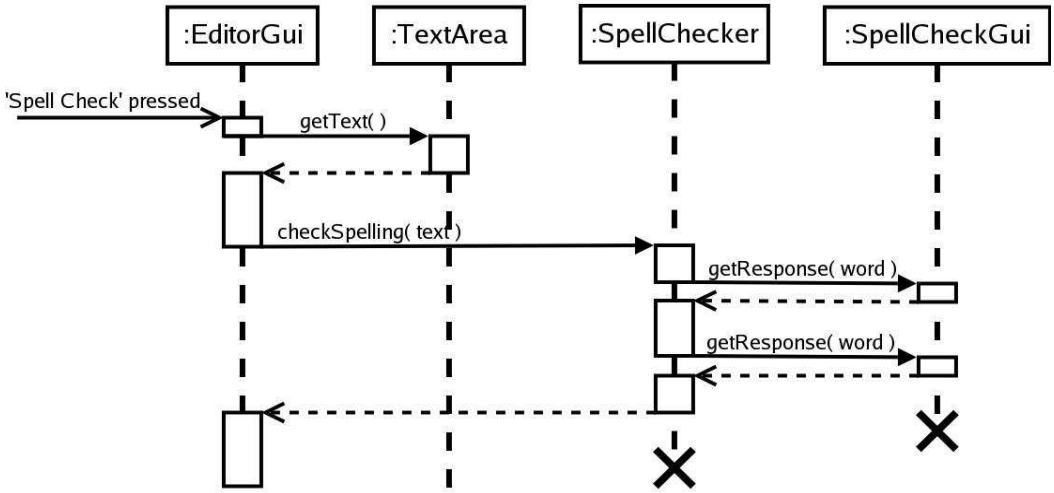




Пример за диаграма на последователност при логване в системата:



Пример за Sequence diagram на текстов редактор – проверка на правописа:

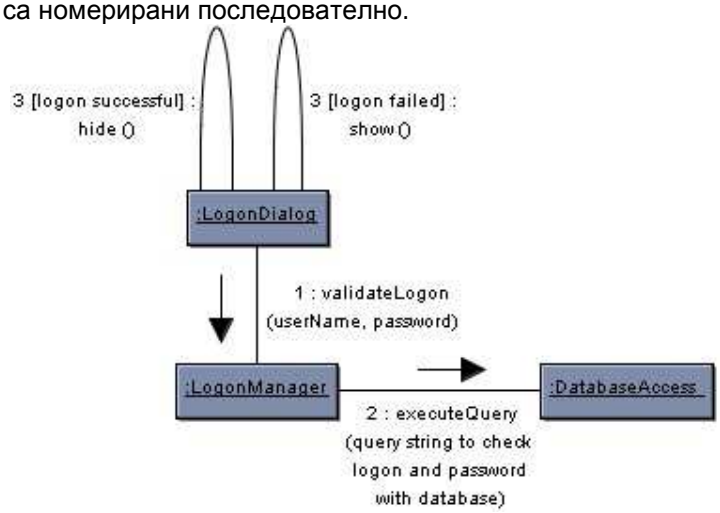


Collaboration Diagrams

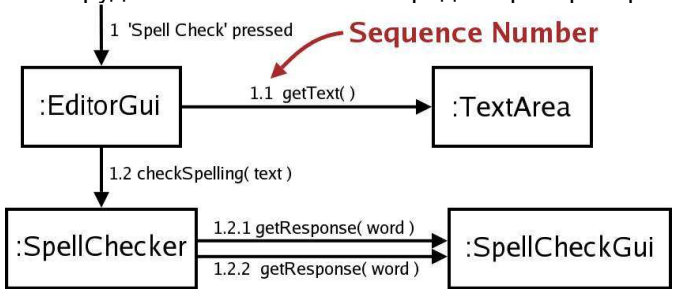
Диаграмата на сътрудничеството е по-комплексна. Тя показва взаимодействията, организирани около обектите и връзките между тях.
Диаграмата на сътрудничество моделира потока на съобщенията между обектите. Тук времето отсъства.
Класовете се представят в правоъгълници с имена. Форматът е: *instance/role name : class name.*
Инстанциите се подчертават.

- Съобщенията са както в диаграмата на последователностите.
- Съобщенията имат пореден номер.
- Времето се представя като номер на последователност на изпълнение.

Пример за взаимодействията при логване в системата като диаграма на сътрудничеството. Всички съобщения са номерирани последователно.



Пример за диаграма на сътрудничеството на текстов редактор – проверка на правописа.



Activity and State Diagrams

До тук са представени диаграми на взаимодействието, които демонстрират поведението на няколко обекта при изпълнение на еднократни случаи на употреба. Когато искаме да покажем последователността на събития в по-широк мащаб се използват диаграми на дейности и на състояния.

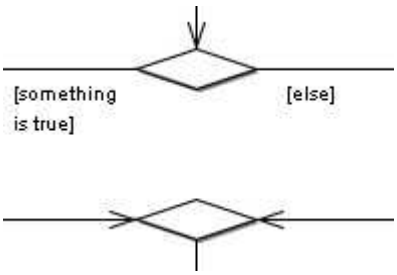
Activity Diagram

Диаграмата на дейности е техника за описване на процедурна логика, бизнес процес и работен поток. Тя е като блок-схема, но поддържа паралелно поведение. Може да има разклонения както блок-схемите. Показва логиката на поведение. Състоянията са дейности. **Дейност** е изпълнението на задача, като това може да бъде физическа дейност или изпълнение на код. Диаграмата на дейности показва последователност от действия.

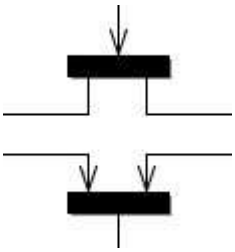
Start: Всяка диаграма на дейности има едно начало, от което започва последователността от действия.

End: Всяка диаграма на дейности има един, край където завършва последователността от действия.

Activity: действията са свързани помежду си с преходи**. Преходите** се означават със стрелки и показват посоката от предишно със следващо действие. Понякога съдържат текст:

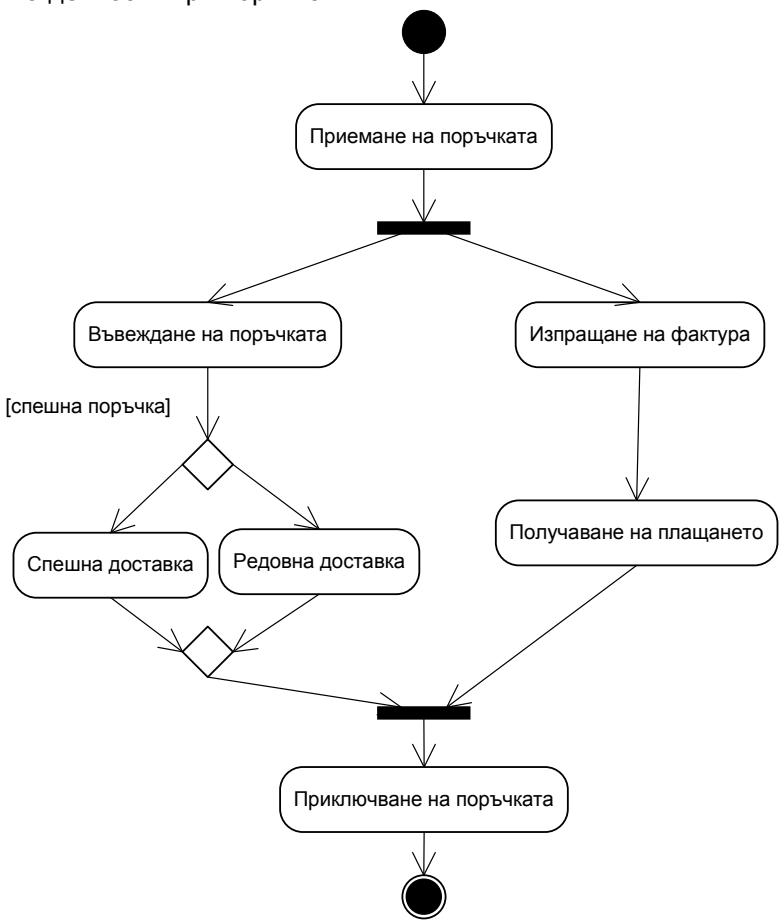


За показване на **условно поведение** се използува разклонение и сливане. **Разклонението** има единствен входящ поток и няколко ограничени изходящи потока. Всеки изходящ поток има ограничение (**guard):** булево условие, поставено в квадратни скоби. Всеки път, когато се достигне разклонение, може да се вземе само едни изходен поток, затова ограниченията трябва да бъдат взаимно изключващи се. **Сливането** се използува за завършване на условното поведение. Може да има множество входящи потоци и единствен изходящ поток.

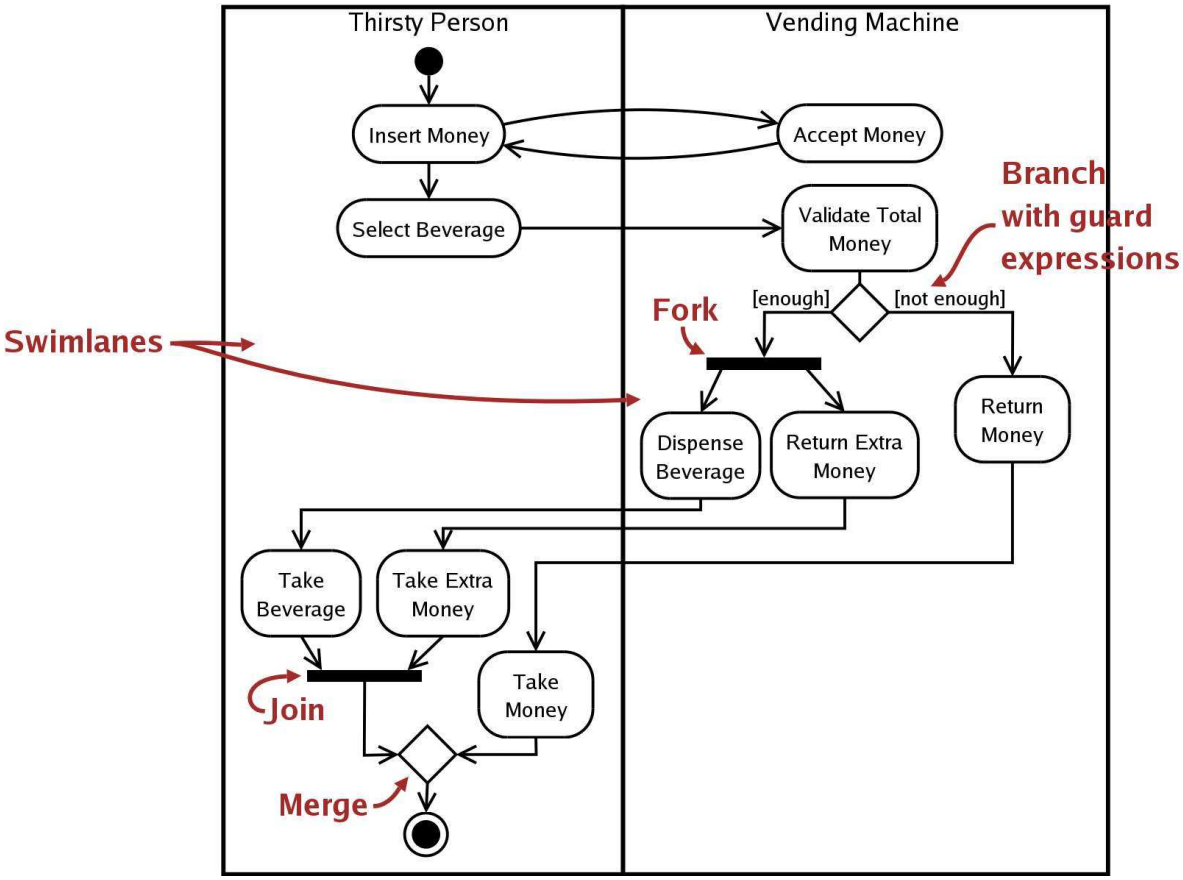


За показване на **паралелно поведение** може да се използва **разклонение и свързване**. **Разклонението** има единствен входящ поток и няколко изходящи потока. **Вилата** (връх) има един преход да влиза в и което и да било число на преходи да излизат, всички ще бъдат взети**. Свързването** представлява края на паралелното поведение и има няколко входни потока и един изходен.

Пример за диаграма на дейности при поръчка.



Пример за диаграма на дейности на машина за продажба на храна.



Диаграмата на дейности (Activity diagram) се използват, за да покажат workflow в паралел и условно. Диаграмата на дейности дава възможност на този, който извършва процеса, да избира реда, в който изпълнява действията. Тя указва съществените правила за последователност, които трябва да се следват. Това е важно за бизнес моделирането, тъй като процесите често настъпват паралелно Те са полезни за показване на паралелизма на последователен алгоритъм, когато се анализират стъпките в бизнес процес. Това също така е полезно за едновременно (concurrent) изпълнявани алгоритми, където независимите нишки могат да извършват действията паралелно.

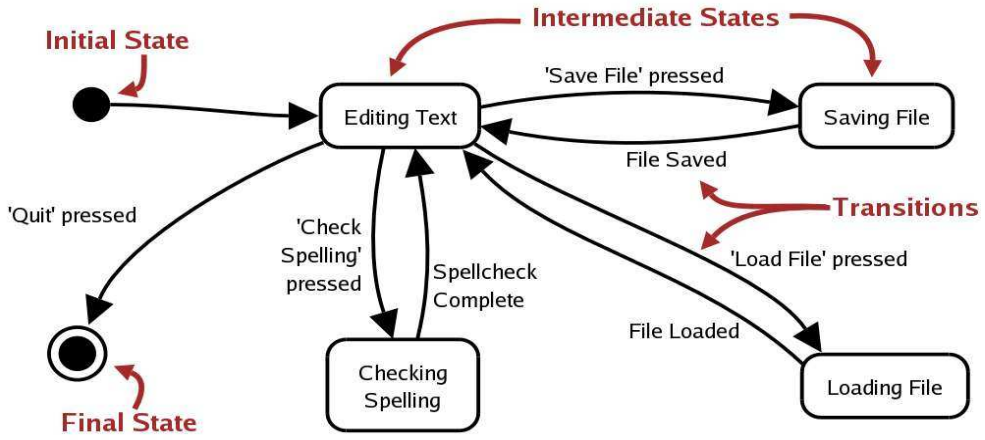
State Diagram

State диаграмата показва състоянията, в които може да бъде даден обект и преходите между състоянията. Показва промяната на обектите във времето – как обекта се променя от началото до края.

Състоянията са представени като закръглен правоъгълник с името на състоянието. Незадължително може да се включи **активност,** която представлява по-дълга изпълняваща се задача през това състояние.
Свързващите състояния са **преходи.** Тези представляват **събитията, които** са причина обекта да промени състоянието от едно към друго. Преходите имат етикети, които се състоят от три части: **предизвикващо събитие [ограничаващо условие] / дейност**. И трите части са незадължителни. Предизвикващото събитие обикновено представлява единствено събитие, което води до промяна на състоянието. Ограничаващото условие (ако има такова) е булево условие, което трябва да бъде **true**, за да се изпълни преходът. Действията представляват задачи или поведение, които се извършват по време на преходите. Действията са различни от активност – действията не могат да бъдат прекъснати, докато активността може да е прекъсната от постъпващо събитие. Липсващата дейност е индикатор за това, че по време на прехода не се извършва нищо. Липсващо ограничаващо условие означава, че винаги при възникване на събитието преходът се изпълнява. Липсващо предизвикващо събитие се среща рядко, но се случва. Това означава, че преходът се изпълнява незабавно.

Както диаграмата на дейности, диаграмата на състоянията има едно **начало** и един **край.**

Пример за диаграма на състоянията на текстов редактор.



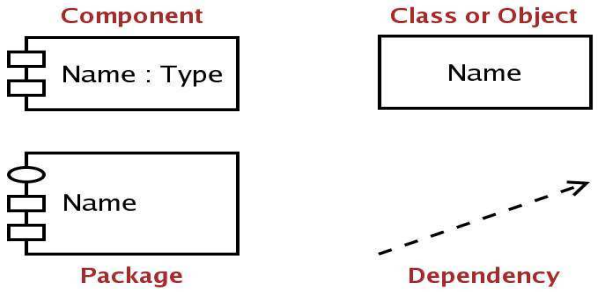
Диаграмата на състоянията (State diagram) показва промяната на обект след време и е полезна за описване поведението на обект в няколко случая на употреба, но не толкова добра при описване на поведението, което включва няколко обекта, работещи съвместно.
Полезно е да се комбинират диаграмата на състояния с други диаграми. Например диаграмата на взаимодействие са подходящи за описване на поведението на няколко обекта в един-единствен случай на употреба, а диаграмите на дейности се използват при показване на общата последователност от дейности за няколко обекта и случаи на употреба.

Тези диаграми се използват само когато имат определена цел. Не е необходимо да се прави диаграма на състоянията за всеки обект в системата, както и диаграма на дейности за всеки процес.

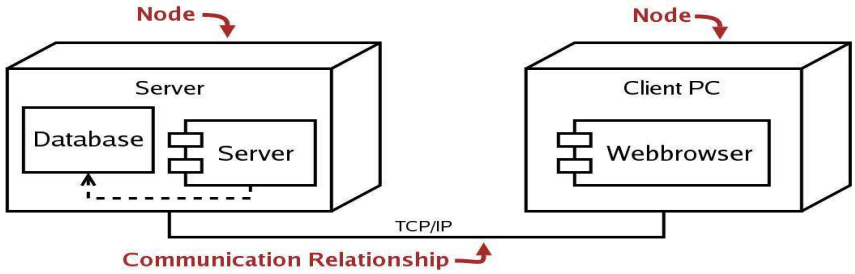
Implementation Diagrams - Component and Deployment

Всички представени до тук диаграми показват задачите, които системата изпълнява, детайлите на класовете, обектите и динамичното поведение на системата. Представянето на цялата система се осигурява с двата типа диаграми на реализация – диаграма на компонентите и диаграма на разгръщането. С диаграмата на **разгръщането** се показва как са свързани физически компонентите в системата, а с диаграмата на **компонентите** се показва как са организирани компонентите в системата. Диаграмата на компоненти показва връзките между основните части на системата. Диаграмата на разгръщане показва къде са разположени физически компонентите на системата. Като допълнение на диаграмата на компонентите, диаграмата на разгръщане използва възел *(nodes)* и комуникационни пътища *(communication relationships)*.

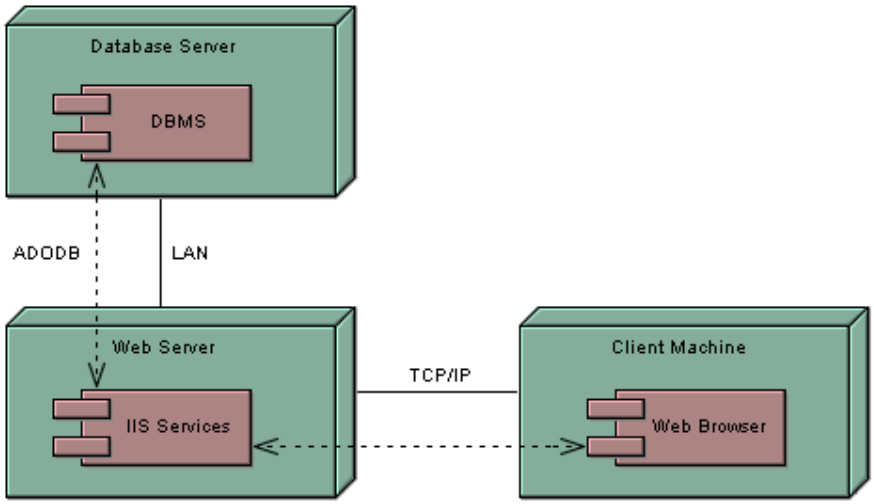
Пример за диаграма на компоненти.



Пример за диаграма на разгръщане.



По избор може да се комбинират двете диаграми:



Възлите (nodes) представят нещото, под което се изпълняват (хостват) компонентите (софтуера), а компонентите **(components)** са част от софтуера. Възел може да бъде: **хардуерно устройство** – компютър или друг хардуер, свързан към системата; **среда за изпълнение** – софтуер, който хоства или съдържа друг софтуер (операционна система и др.).
На диаграмата, възлите са свързани с **връзки,** които показват физическия път на информацията в системата. Компонентите са свързани с линии, които представляват **комуникацията** между компоненти. Може също така да се използва **интерфейси** на компоненти за показване комуникацията през интерфейс. Физическите диаграми помагат за цялостното представяне на структурата и системата.

GETI-2100: Informatique de Gestion

Information System Engineering: Analysis and Design

Section 3.2.3: Object-Oriented Modeling

Activity Diagram

Prof. Stéphane Faulkner

Univesité catholique de Louvain, 2005-2006

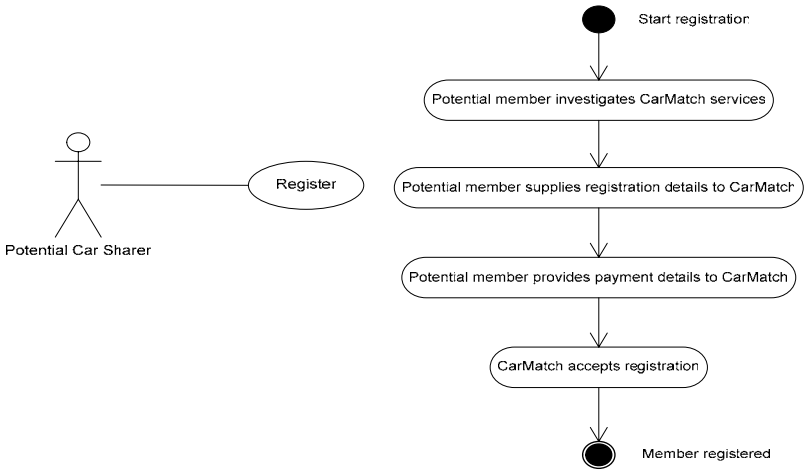
Section overview

- What is an activity diagram
- Basic concept for the use-case diagram
  - Activities
  - States
  - Transitions
- Decision points
- Swimlanes
- Forks and joins
- Iteration
- Control Icons
- Objects on activity diagrams
- How to produce activity diagrams

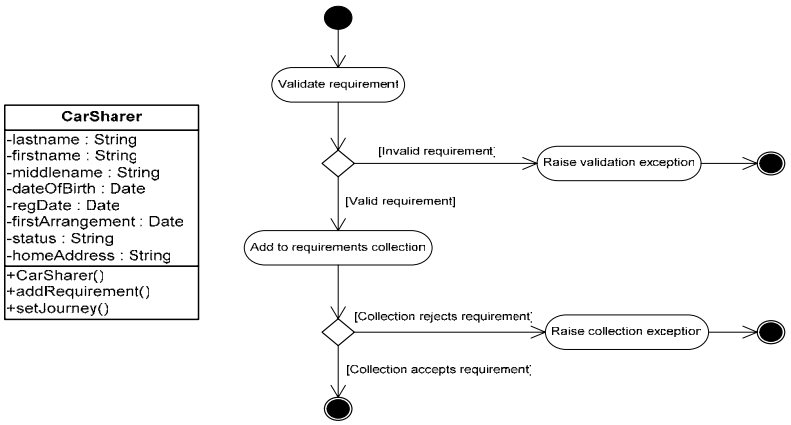
What is an activity diagram?

- Activity diagrams are a means of describing business workflows (business processes), workflows within use-cases, workflows between use-cases and workflows for complex operations
  - Workflows: generic term for modelling tasks in terms of a sequence of activities
- Activity diagrams consist of *activities*, *states* and *transitions* between activities and states
  - An activity is a specification of behavior expressed as a flow of actions
    - Actions are executable statements
    - An activity can consist of one or many actions
  - A state is a point where some event needs to take place before activity can continue
  - A transition is the movement between activities and/or state. Transitions are supposed to represent actions which occur“quickly” and are not interruptible

Activity diagram for describing use-cases

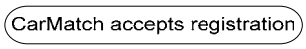


Activity diagram for describing operations



Activities

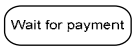
- An activity is a unit of work
  - A work can be documented as actions in the activity



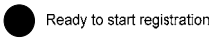
- There are four ways in which an action can be triggered:
  - On Entry: these actions are triggered as soon as the activity starts
  - Do: these actions take place during the lifetime of the activity
  - On Event: these actions take place in response to an event
  - On Exit: these actions take place just before the activity

States

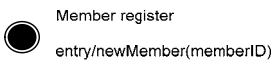
- States are used to imply waiting, not doing



- There are two special states:
  - Start state
    - Only one start state is allowed on a diagram



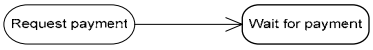
  - End state
    - There can be several end states in a worflow
    - End states can have actions to trigger events that start other process



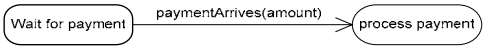
Transitions

- A transition occurs when all the actions of an activity have been completed or when an event triggers the exit from a particular state or activity

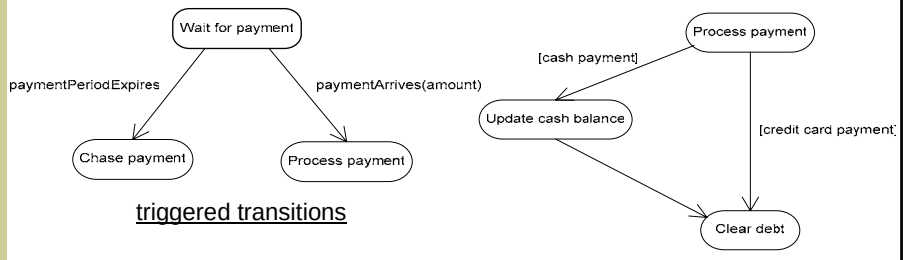
A triggerless transition:



A triggered transition:

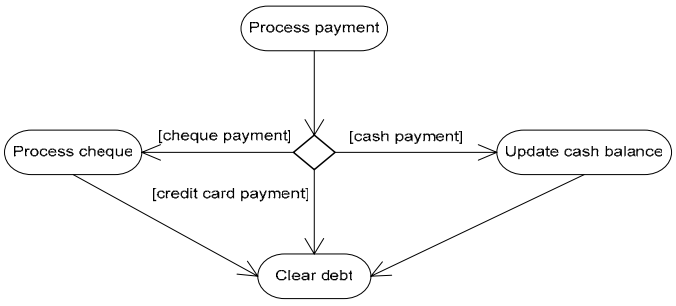


Multiple transitions



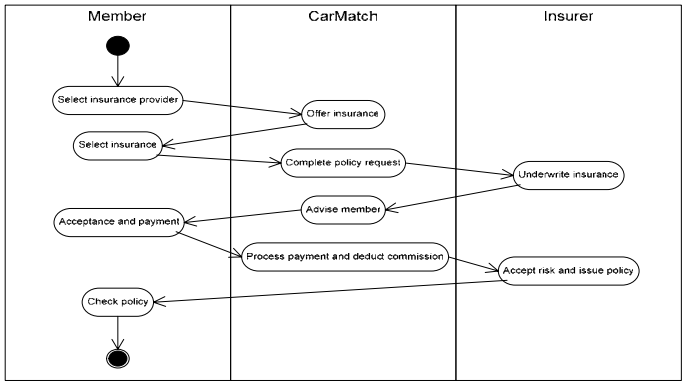
Decision points

- A decision point is a point in a workflow where the exit transition from a state or activity may branch in alternative directions depending on a condition



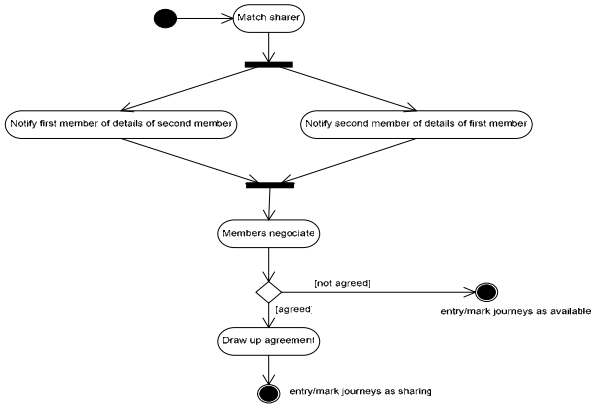
Swimlanes

- Swimlane is a useful notation for indicating where an activity takes place in a complex system

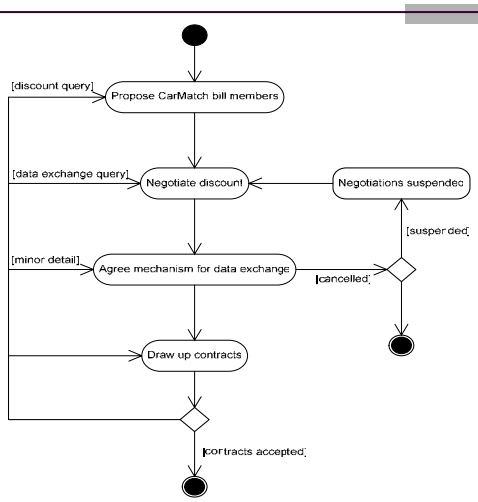


Forks and joins

- A transition can be split into multiple paths (a fork) and multiple paths combined (a join) into a single transition by using a synchronization bar

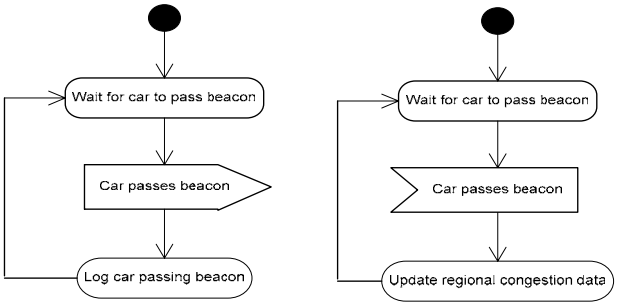


Iteration



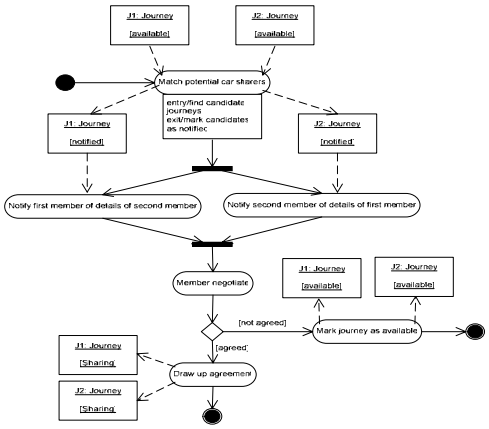
Control icons

- Transitions can also trigger events themselves
  - Two icons: the sending of a signal (i.e., event) or the receipt of a signal



Objects on activity diagrams

- All activities need to be carried out by objects in an OO implementation
  - it can useful to indicate where a flow affects an object



How to produce activity diagrams

1. Identifying key scenarios of system use-cases
2. Combining the scenarios to produce comprehensive workflows described using activity diagrams
3. Add objects flows to the diagram where significant object behavior is triggered by a workflow
4. Use swimlanes to map the activities where worflows cross technology boundaries

2. Изисквания

Видове изисквания

- **Системни**(детайлно описание на функциите, услугите и ограниченията) **и потребитески**(за потребителите – функционалните и нефункционалните изисквания на естествен език)
- **Функционални**(услуги, които с-мата трябва да представя; „потр. да търси в базата данни“, „всяка заявка да има ID, който потр. да може да копира“) и **нефункционални**(ограничения върху услугите и функционалността на с-мата)
  - Изисквания **към продукта** - определят специфичното поведение на продукта – бързина, памет, сигурност („потр, интерфейс ще се осъществи като прост HTML“)
  - **Организационни** изисквания – от политиките на организациите – стандарти за процеси, исиквания за реализацията(език за програмиране) („процесът на разработка и доументите трябва да са според дефинираните правила в XYZCo-SP-STAN-95“)
  - **Външни** изисквания – от фактори, външни за с-та – взаимодесйтвие на с-та с другите отганизации („Системата не трябва да разкрива пред операторите никаква лична информация за клиентите освен името и акаунт номера“)
- Изисквания **на областта**(от приложната област на с-мата, отразявайки характеристики на тази област могат да бъдат нови функционални изиксвания, ограничения въхру същетсвуващи вече функционални изисквания, указания как трябва да се извършат определени опрации)
- Организация на изискванията в документи - **техническо задание** – да включва както потребителските изисквания, така и системните такива за това какво и как върши с-мата

Техники за събиране на изискванията

Управление на обхвата на проекта

Събиране на изискванията

Дефиниране на обхвата

Създаване на WBS

3. Изисквания. Методи

Методи за определяне на изискванията :

- **Традиционни** – интрвюта(на стейкхолдъри – отворени и затворени), въпросници(повече време за отговор, различна гама за оценяване на въпроса), наблюдение(пасивно, активно, обясняващо), разучаване(use case, организационни документи, книги, интернет)
- **Модерни**- (за проекти с по-голяма степен на риск) прототипиране, brainstorming, joint application development

4. **Contextual design**

Контекстуално проектиране- структуриран подход за получаване на информация от конкретна област

Стъпки:

- **Контекстуално интервю** – интерпретационна сесия на работното място на потребителя
- **Моделиране** – множество модели за представяне (flow, sequence, artifact, cultural, physical)
  - **Flow** – Определяне на потока на комуникация и координация
  - **Sequence** – определяне на последователността от действия за постигането на определена цел/задача
  - **Artifact** – идентифициране на физически обекти и работа с тях
  - **Cultural** – определяне на ценностите, ограниченията и приоритетите на работното място (кой на кого влияе)
  - **Physical** – оформление на физическите обекти и дейностите, които се поддържат
- **Консолидация** – обединяване на моделите с цел получаване на детайлни обобщени модели (афинити диаграма)
- **Редизайн** на работата – създаване на визия на новите работни практики; обмисляне на по-добри начини хората да с вършат работата, без проблемите, открити в моделите; изграждане на работни сценарии и примерни сторибордове
- **User environment design** (UED)– визията, представена в **Storyboards**; идентифициране на „области“; изграждане на околната среда на потребителя (walktrough)
- **Макети и тестване с потребители** – хартиени **прототипи**
- **Практическа реализация**

5. Flow model

Определя ролите на хората и начина, по който те общуват и координират работата си. Акцентът е върху отделния индивид и взаимодействията му с другите.

Компоненти:

- Лица/ индивиди – *овал* (име на ролята, идентификатори и отговорности)
- Други хора и групи, с които лицето си взаимодейства – всеки в собствен *овал* (името на ролята и отговорностите им)
- Поток от комуникация между индивида и останалите – *стрелки* между овалите; именуване на тема/ действие
- Артефакти – *малки кутии* върху стрелките - физическите предмети, които се обменят между лицето и останалите
- Местата, в които се изпълняват определени задачи – *големи кутии*
- Проблемна област – *светкавица* - показва ситуации, в които задачата не може да се завърши успешно

6. Sequence model

Описва последователността от действия, които потребителят изпълнява, за да свърши определена работа. Включва *тригери*, които провокират стартирането на стъпките и *целите* на последователността от стъпките.

Компоненти:

- Стъпки – какво се случва, последователно подреждане на стъпките
- Подредба, повторения, разклонения – *стрелки,* които показват последователни действия; *разклонения* – решенията, които трябва да бъдат взети
- Цели – какво трябва да постигне последователността от стъпки; основна и вторични цели
- Тригери – събитието, което инициира задачата, трябва да има аналог в новата система
- Проблемни области – възникнали проблеми при изпълнение на стъпките

7. Artifact model

Чертеж или фотокопие на артефакт, анотиран, за да се посочат функциите на различните части на артефакта

Компоненти:

- Скица, фотокопие или картина на артефакта
- Части на артефакта – какви са отделните части на артефакта, кои се използват и кои не
- Информацията, съдържаща се в обекта – каква е тя, как се използва в работния процес
- Представяне – как е проектиран артефактът, за да се подпомогне използването му (пр.: фон, използване на свободното пространство, позиции)
- Анотации – как се разширява артефактът, какво се добавя към него
- Концептуално разделение – как човек раздел артефакта, за да подпомогне концептуално различни задачи (пр.: как хората разделят портфейла си, за да организират съдържанието му)
- Използване
- Проблем области – какви проблеми срещат хората при използване на артефакта

8. Life cycle models – жизнен цикъл на софтуерен продукт

Основни понятия:

- *Жизнен цикъл* на програмния продукт – период на създаването и използването му
- *Начало* – възникване на идеята за създаване
- *Край* – преустановяване на използването на последното копие на продукта
- *Софтуерен процес* – набор от дейности, необходими за разработката на продукта
- *Модел на софтуерния процес* – абстрактно представяне на софтуерния процес; дефинира набор от дейности, задачи, ключови резултати и отчетни материали, необходими за изграждането на висококачествен софтуер

- *Фаза* – отрязък от време, през което се извършват определени дейности по разработвания проект. Фази:
    - *Inception* - планиране
    - *Elaboration* - детайлизиране
    - *Construction* - изграждане
    - *Transition* – предаване
- *Итерация* – в рамките на фаза
- *Use case* – Описание на последователността от действия, включващи вариантите, които една система може да изпълни така, че да се получи определен резултат за актьора; поток от събития

Основни етапи от разработката на софтуер:

- **Анализиране на изискванията** (requirements engineering) – изискванията (software requirements) описват системата, която трябва да се разработи – функционални и нефункционални изисквания (въпроса „какво“, а не – „как“)
- **Проектиране (дизайн)** на системата
    - ***Софтуерен*** - описва как системата ще изпълни изискванията
    - ***Архитектурен*** – как задачата ще бъде разбита на подзадачи, отговорностите на отделните модули, взаимодействията между модулите, интерфейсите за връзка
    - ***Детайлен*** – структура и организация на всеки от модулите в детайли
    - ***Обектно-ориентиран*** – класове и обекти, техните отговорности, как те си взаимодействат
    - ***Вътрешен*** дизайн на класовете – описва как работи всеки клас
- **Имплементация** – процес на създаване на програмен код, който стриктно следва дизайна
- **Тестване** – проверява дали дадено решение изпълнява всички изисквания с цел намирането на дефекти

Жизнен цикъл – основни дейности

- ***Специфициране*** – дефиниране на функционални и нефункционални изисквания
- ***Проектиране и реализация*** – изграждане на програмен продукт в съответствие със специфицираните изисквания
- ***Валидиране*** – тестване на продукта, за да се удостовери, че отговаря на изискванията на потребителя
- ***Развитие*** – на продукта в съответствие с променящите се нужди на потребителя

Модели на жизнения цикъл

1. Пълни

    1.1 Едномерни

        1.1.1 Хронологични

        1.1.2 Функционални

    1.2 Многомерни

1.2.1 Двумерен

1.2.2 Тримерен

1.3 Циклични

1.3.1 Инкрементален

1.3.2 Спирален

1.4 Комбинирани

1.4.1 Rational unified process

2. Частични

3. Гъвкави

1.1.1 **Хронологични** (последователни) модели – бизнес проблемът, който се решава, може напълно да бъде разбран и описан преди дизайна на решението. Дизайн, който удовлетворява всички аспекти на проблема, може да бъде специфициран преди реализацията. Реализацията може да се извърши преди валидиране и предаване. Недостатък – последователните методи увеличават риска с времето.

Фази на разработка **от гледна точка на потребителя**:

- **Специфициране** на изискванията и анализ
- **Проектиране**
- **Реализация** на програмния продукт
- **Поддръжка**

Пример: Каскаден модел

Дефиниране на изискванията – функциите, ограниченията и целите на системата се определят съвместно с потребителите. Следва детайлното им дефиниране, което се използва като спецификация на системата

Системен софтуерен дизайн – проектиране на инфраструктурата и софтуерната архитектура съгласно функционалните и нефункционалните изисквания. Изграждане на обща системна архитектура. Детайлно идентифициране и описание на основните елементи на програмния продукт и връзките между тях.

Имплементация и unit тестване – реализация на софтуерния дизайн като набор от програмни unit-и.

Итерационно и системно тестване – отделните компоненти се интегрират и тестват като цяла система, за да се провери, че са изпълнени функционалните и нефункционалните изисквания

Експлоатация и поддръжка – използване на системата. Поддръжката включва отстраняване на грешки. Които не са били открити в по-ранни фази и усъвършенстване и разширяване на системата при възникване на нови изисквания

1.3.1 Инкрементален модел – предимства – достъпност на продукта на ранен етап с по-малък риск; недостатъци – трудно разделяне на продукта на малки части за разработка

Разработването и предаването на системата е разделено на отделни версии, наречени *increment*-и. Всяка версия реализира част от изискваната функционалност. Всеки *increment* разширява обхвата на реализираната функционалност на системата.

Потребителските изисквания се подреждат по приоритет и разработката следва приоритета.

След идентифициране на отделните *increment*-и, изискванията за първия *increment* се детайлизират.

След стартиране на разработката на един *increment,* изискванията за него се замразяват, докато изискванията за по-късните *increment*-и продължават за се развиват.

1.3.2 Спирален модел – предимства – оценка на риска след всяка фаза; недостатъци – трудности при управлението на процеса

Обединява еволюционния подход с изискванията на линейния хронологичен модел свързани с управление и контрол. Разработката се движи спираловидно като серия от инкрементални версии. Спиралния модел е разделен на определен брой рамкови дейности:

- Връзка с потребителя
- Планиране
- Анализ на риска
- Проектиране
- Реализация и предаване
- Оценка от потребителя

Всяка обиколка представлява фаза в процеса на разработка. Сектори на спиралния модел:

- Дефиниране на целите за фазата
- Оценка на риска и редукция на основните рискове
- Разработка и валидиране
- Планиране на следващата фаза

1.4.1 Rational unified process – архитектурно-ориентиран, итеративен и инкрементален софтуерен процес, проектиран като рамка за ОО софтуерно инженерство с UML. Базира се върху use case модел за описание на функционалността.

Фази: (фази на UP)

- Планиране (Inception)
    - Дефиниране на обхвата на проекта – основните бизнес изисквания се формулират чрез набор от use case-и.
    - Избор на обща архитектура – определяне на подсистеми
    - Планиране – ресурси, рискове, график

- Детайлизиране (Elaboration)
    - Разширяване и детайлизиране на use case-и
    - Анализ и дизайн
    - Изграждане на архитектура
- Изграждане (Construction)
    - Завършване на дейностите по анализ и дизайн
    - Разработване на софтуерни компоненти, които реализират отделните use case-и
    - Системни тестове – unit и интеграционни тестове
- Предаване (Transition)
    - Обучение на потребители
    - Потребителски тестове
    - Отстраняване на грешки

Добри практики в RUP

- Софтуерът се разработва итеративно
- Изискванията се управляват
- Софтуерът се моделира визуално
- Верифицира се качеството
- Контролират се промените в софтуера

3. Гъвкави (agile) методи

- Scrum
- Екстремно програмиране – XP
- Adaptive software development
- Dynamic system development method

Основни понятия

- Роли – PO, scrum master, team
- Специфични прояви – sprint planning, sprint review, sprint retrospective, DSM
- Артефакти – product backlog, sprint backlog, burndown chart

9. OOAD - Object-oriented analysis and design

UP дисциплини

- *Бизнес моделиране* – разработката на едно приложение включва моделиране на домейн обектите. При по-мащабен анализ или реинженеринг на бизнес процесите се включва динамично моделиране на бизнес процесите в цялото предприятие.
- *Изисквания* – анализ на изискванията, описание на потребителски случаи и определяне на нефункционалните изисквания
- *Дизайн* – всички аспекти на дизайна, включително цялостната архитектура, обектите, базите от данни, мрежите

Добри практики в UP

- Отбелязване на високо-рисковите и важните следствия в ранните итерации
- Непрекъснато взаимодействие с потребителите за оценка, корекции и изисквания
- Създаване на свързана, централна архитектура в ранните итерации
- Непрекъснато оценяване на качеството
- Ранно тестване
- Прилагане на потребителски случаи
- UML
- Внимателно управление на изискванията
- Практики на промяна на изискванията и управление на конфигурациите

Итеративна разработка и UP

- Итеративната разработка е в центъра на object-oriented analysis and design (OOA/D)
- Неформално процесът по разработка на софтуер описва начин
- Unified process (UP) – процес за разработката на софтуер при реализация на ОО с-ми
- Rational unified process (RUP) – детайлизирано приложение на UP

Unified modeling language (UML) – език за определяне, визуализация, създаване и документиране на документи на софтуерни системи, както и за бизнес моделиране и други не-софтуерни системи.

Използване на UML и шаблони в ООА/Д …x3

Анализ и проектиране (дизайн)

- *Анализът* набляга повече на **изследването на проблема и изискванията**, отколкото на решението. „Анализ“ е широко понятие. Най-добре определено в анализа на изискванията (изследване на изискванията) или анализа на обектите (изследване на домейн обектите). „направи правилното нещо“
- *Дизайнът* набляга повече на **концептуалното решение**, което изпълнява изискванията, отколкото на имплементацията. Пример – описание на схемата на базата от данни и софтуерните обекти. „направи нещата правилно“

Обектно-ориентиран анализ и дизайн

- По време на *ОО анализ* се набляга на **откриване и описване на обектите** и/или понятията в разглеждания домейн (предметна област). Пример – В ИС на библиотека, някои от понятията са *Book, Library* и *Patron.*
- По време на *ОО дизайн* се набляга на **дефиниране на софтуерните обекти и как те си взаимодействат** за изпълнение на изискванията. Пример – В библиотечната система, софтуерният обект *Book* може да има атрибут *title* и метод *getChapter.* Накрая по време на имплементацията или ООП, обектите се имплементират, например клас *Book* в Java.

Потребителски случаи (use case)

Анализът на изискванията може да включва описание на свързаните домейн процеси – те могат да се опишат като потребителски случаи.

Домейн модел

ООД се занимава със създаването на описания на домейните от перспективата на класификация по обекти. Декомпозицията на домейна включва идентифициране на идеите, атрибутите и асоциациите, които се разглеждат изцяло. Резултатът може да се изрази в модел на домейна, който се обозначава с множество диаграми. Които показват идеите или обектите на домейна.

Диаграми на взаимодействието (Interaction diagrams)

OOD се занимава с дефиниране на софтуерни обекти и тяхното взаимодействие. Диаграмите показват потока на съобщения между софтуерните обекти и обръщенията към методите.

10.Потребителски случаи (use case)

Потребителски случаи са широко разпространен механизъм за откриване и записване на предимно функционални изисквания. Описанието им е отлична техника за разбиране и описание на функционалните изисквания. Use case model – част от описанието на изискванията в UP. Потребителските случаи не са ОО.

Всяка цел идентифицира потребителски случай.

Понятия:

- *Актьор* – нещо с поведение – човек, компютърна система или организация.
- *Сценарии* – определена последователност от действия и взаимодействия между актьорите и системата; един възможен път през потребителския случай.
- *Потребителски случай* – множество от свързани успешни и неуспешни сценарии, които описват как актьорът използва системата за постигане на своята цел.
- *Диаграмите* визуализират имената на потребителските случаи и актьорите, както и връзките между тях.

Елементи на диаграмата:

- *Елипса* – потребителски случай – започва с глагол
- *Човече* – актьор – всичко с поведение, включително и системата
    - *Главни актьори* (отляво) – техните роли се изпълняват чрез системата (както хора, така и системи)
    - *Поддържащи* - предоставят услуги на системата
    - *Второстепенни* (отдясно) – има интерес от поведението им в потребителския случай
- *Правоъгълник със стереотип* – показва какъв е актьорът

Формати на описание на потребителските случаи

- *Кратък* – резюме – описва основния успешен сценарии
- *Неформален* – описва различни сценарии
- *Пълен* – най-подробен – описва всички стъпки и варианти в детайли

//пример за fully dressed

FURPS+

Потребителските случаи в начална фаза (Inception)

- Определяне на основните потребителски цели и записване на потребителските случаи в кратък формат.
- Определят се целите и заинтересуваните лица и границите на проекта.
- Не всички потребителски случаи се описват в пълен формат, само най-сложните.

Потребителските случаи през фазата на детайлизиране (Elaboration)

- Доизясняват се изискванията
- Детайлизират се целите и потребителските случаи
- До края на фазата почти всички потребителски случаи са описани в пълен формат

Потребителските случаи във фаза изграждане (Construction)

- Повечето основни функционални и нефункционални изисквания са стабилизирани
- Минимално писане и промяна на потребителските случаи

11. Use case modeling – a model that describes a system's functional requirements in terms of use case. A model of the system's intended functions (use cases) and its environment (actors).

12. From Inception to Elaboration

Планиране – целта е да се определят някои основни представи за целите на проекта, да се определи дали е осъществим и да се уточни дали си струва да се проучи сериозно.

- Каква е визията на проекта?
- Осъществим ли е?
- Закупуване и/или създаване на нов продукт?
- Начална преценка на разходите?
- Трябва ли да продължим или не?

Документи:

- *Визия* – дефинира **вижданията за продукта на заинтересованите страни**, като включва социалните и бизнес нужди, както и основните характеристики и базовите изисквания на системата
- *Глосарий* – включва **термини и дефиниции** (може да е като речник или да съдържа метаданни).
- *Допълнителни изисквания* – **документация, поддръжка, лицензионни права**. Тук се определят FURPS+, атрибутите, определящи качеството на ИС.

Приоритети:

1. Кратка чернова на визията
2. Определяне на целите на потребителите и опорните потребителски случаи
3. Описание на някои потребителски случаи и допълнителна спецификация
4. Прецизиране на визията

Usability

- Текстът трябва да се вижда добре от разстояние 1м.
- Да се избягват цветове, които често са неразпознаваеми от далтонисти.
- Предупрежденията на системата трябва да бъдат съпроводени и със звук.

Reliability

- При срив – локално решение
- Допълнителен анализ

Performance

- Приключване на оторизацията за по-малко от минута в 90% от случаите.

Supportability

- Приспособимост
- Възможност за допълнително конфигуриране

Други

- Ограничения за начина на реализация на продукта – желана технология- Java
- Закупени компоненти/модули за вграждане в системата – данъчен калкулатор
- Свободни компоненти с отворен код
- Интерфейси – хардуер и софтуер
- Други (лицензи, стандарти, влияние на околната среда)

13. Domain model – визуално представяне на концептуални класове или реални обекти в домейна. Представя реални концептуални класове, а не софтуерни компоненти. Основно се изготвя във фазата за детайлизация. Илюстрира:

- Домейн обекти или концептуални класове
- Връзки между концептуалните класове
- Атрибути на концептуалните класове

Елементи:

- Домейн обекти – използване на съществуващи имена от областта
- Асоциации – отношенията между класовете, които представят смислена връзка, *„нужно-да-знаещ“*

- Роли – всеки край на асоциацията – име, изрази, указващи множественост, направление
- Атрибути – логически данни за обекта, описват характеристиките на обектите

14. Activity diagrams – описва процедурна логика, бизнес процес, работен поток. Показва последователност от действия.

Компоненти:

- Действие – стрелка
- Изход – точка в кръгче
- Вход – точка
- Условно поведение
    - Описание на паралелни процеси
        - Разделяне (forking) – стрелка, черта, 2 стрелки
        - Свързване (joining) – 2 стрелки, черта, стрелка
    - Условно поведение с ограничение ([guard])
        - Разклонение (branching) – стрелка, ромбче, 2 стрелки
        - Сливане (merging) – 2 стрелки, ромбче, стрелка
- Дялове (swimlanes) – разделение с черта

15. Interaction diagrams

Елементи:

- Квадрат – клас, :инстанция, име: инстанция

Sequence diagram – от горе на долу

- Съобщения – стрелка за посока и име
- Връщане на резултат – стрелка на обратно (message syntax – returnVar = message(parameter))
- „Self“ и „this“ съобщения – вложени activation boxes – стрелка към квадратче в себе си
- Създаване на инстанции – създават се от мястото, където сочи стрелката
- Деструкция на инстанции - X
- Условни съобщения – [color = red] calculate() върху стрелката
- Итерация - *[i:=1…N]: num :=nextInt() – върху стрелката
- Серия от съобщения – рамка с *[i:=1…N]
- Multiobject итерации
- Съобщения към класове
- Рамка – например loop (more items)

Communication diagram – чрез номера

- Връзки (черти), по които протичат съобщения със стрелка
- „Self“ и „this“ съобщения – черта към себе си
- Създаване на инстанции – съобщенията, които създават инстанции, имат име *create(sth)*
- Номерация – първото съобщение не се номерира
- Условни съобщения – 1[color=red]: calculate() над чертата
- Взаимно изключващи се условия - 1a[test1] и 1b[not test1]
- Итерация – 1*[i:=1…N]: num:=nextInt()
- Съобщения към класове – например за извикване на статични класове; инстанциите се подчертават, а класовете - не

16. Statecharts diagram – промените в състоянието на обекта и събитията, които са предизвикали тези промени

Елементи:

- Преход (transition - *стрелка*) – връзка между две състояния, показваща, че обектът в първото състояние ще извърши някакво действие и ще премине във второто състояние, когато определено множество събития и условия са удовлетворени
- Събитие (event()) – задейства прехода между състоянията
- Условност – chech pin [correct]
- Действие при условие – check pin [incorrect] / incrementErrorCounter
- Вътрешно събитие – причинено от нещо в границите на системата
- Времево събитие – причинено от възникването на дата и час или откъс от време
- Self-transition – преход, чиито изходящи и крайни състояния са едни и същи; стрелка към себе си
- Start marker – точка
- End marker – точка в кръгче
- Entry/getBalance() – при влизане в състоянието
- Exit/tellBalance() – при напускане на състоянието
- Do/addBalance() – в състоянието

Евристики на Нилсен:

1. Яснота за статуса на системата

2. Връзка между с-мата и реалния свят

3. Потребителски контрол и свобода (undo/redo)

4. Консистентност

5. Избягване на грешки

6. Инструкции за използване – трябва да са видими или лесни

7. Гъвкавост и ефективност при употреба

8. Диалозите не трябва да съдържат ненужна информация

9. Ясни error msg.

10. Help & documentation