Use-cases

An approach to capturing and describing software requirements and basis for use-case driven development

Use-cases

- very useful tool in requirements capture and description
- intuitive and easy to understand, so can discuss with client
- document behaviour of system from “external” point of view
- developed from scenarios in Objectory – Ivar Jacobson 1986 (Objectory now replaced by Unified Process)
- can also be **primary element in project planning** and development
- and for systems validation (type of testing)
- widely adopted by OO community, called Stories in Agile world

Use-cases Definition

- It describes a single functional requirement from a user's point of view, and describes the expected interactions between the user and the software (scenarios).
- **use case** – set of scenarios tied together by common user goal. Or a coherent unit of functionality
- **scenario** – sequence of steps describing an interaction between a user and a system
- can consider a scenario as an *instance* of a use case

Use-case misuse

- A use-case does not and should not describe the inner workings of the software, i.e. it should not be considered as a software design artifact

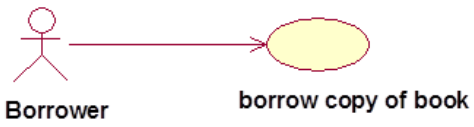
Use-case scenarios example

Use case: **borrow a copy of a book**

- scenario 1: object interactions in successful borrowing
- scenario 2: object interactions in when the maximum number of copies on loan by member already reached
- scenario 3: object interactions in when member has a fine outstanding and pays fine
- scenario 4: object interactions in when borrower is not a valid library member

Use-cases

- graphical notation: actor + use case



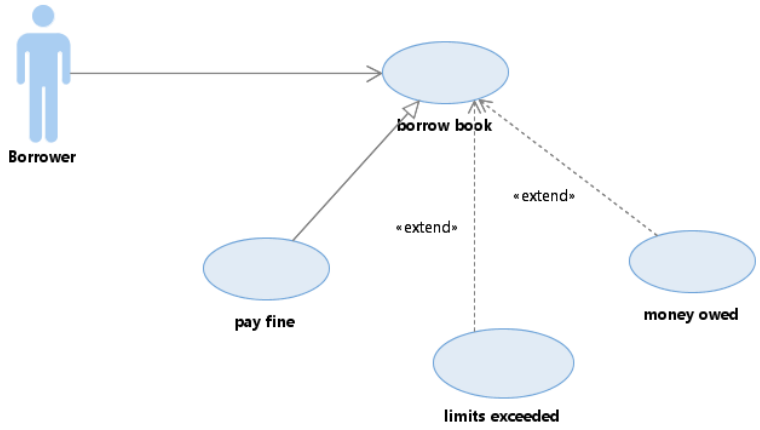
- accompanied by brief description of primary scenario in natural language
- and possibly also some alternative scenarios
- can also add preconditions and post conditions, which must be true before use case can start and after it completes
- no UML standard on this

6

Use-case template

- Martin Fowler states "There is no standard way to write the content of a use case, and different formats work well in different cases."
- He describes "a common style to use" as follows:
  - Title: "goal the use case is trying to satisfy"
  - Main Success Scenario: numbered list of steps where a step is: "a simple statement of the interaction between the actor and a system"
  - Extensions: separately numbered lists, one per Extension where an extension is: "a condition that results in different interactions from … the main success scenario". An extension from main step 3 is numbered 3a, etc.

Use-case diagram example



How big should a use-case be?

Consider online purchase example

Buy a Product

1. Customer browses through catalog and selects items to buy
2. Customer goes to check out
3. Customer fills in shipping information (address; next-day or 3-day delivery)
4. System presents full pricing information, including shipping
5. Customer fills in credit card information
6. System authorizes purchase
7. System confirms sale immediately
8. System sends confirming email to customer

Alternative: *Authorization Failure*
At step 6, system fails to authorize credit purchase
Allow customer to re-enter credit card information and re-try

Alternative: *Regular Customer*
3a. System displays current shipping information, pricing information, and last four digits of credit card information
3b. Customer may accept or override these defaults
Return to primary scenario at step 6

Use-cases – how big?

- what about situation where there is a returning customer?
- another scenario or does it merit a separate use case?
- can use an <<extend>> use-case relationship
- amount of detail depends on risk in the use case
- only detail some use-cases during elaboration phase in UP
- possibly add detail to other use-cases during later iterations

Use-case diagram example

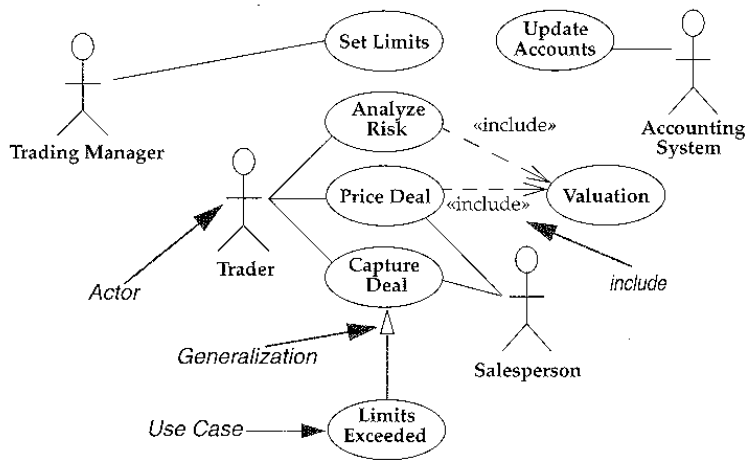


Figure 3-2: *Use Case Diagram*

Actors and use-cases

- Actor is a role users plays with respect to (wrt) the system
- many users can play same role, one user can play many roles
- actors useful for identifying use cases, first establish actors, then their associated use cases
- does not mean actor is human, e.g. can be external system such as accounting system
- actor can be initiator or that which get value from use case – primary actor
- important issue is use cases, actors only a means

Actors and use-cases

May have other uses

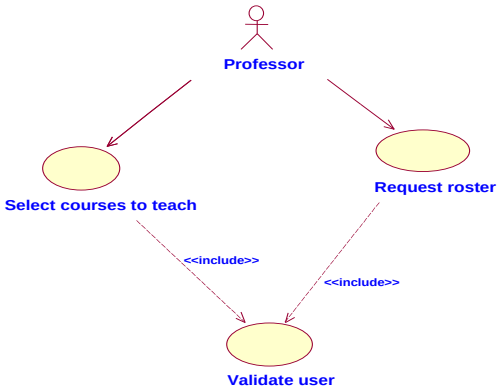
- configuring system for different types of users
- negotiate priorities among use cases – who want what
- use case may have no clear link to actor
  - e.g. *send out bill*
  - is *customer* the actor?
  - could consider *Billing Department* as actor - it gets value
- not all use cases follow from actors, e.g. embedded real time software
- response to external events may help identify use cases, event may cause
  - system reaction
  - user reaction

Use-case Relationships

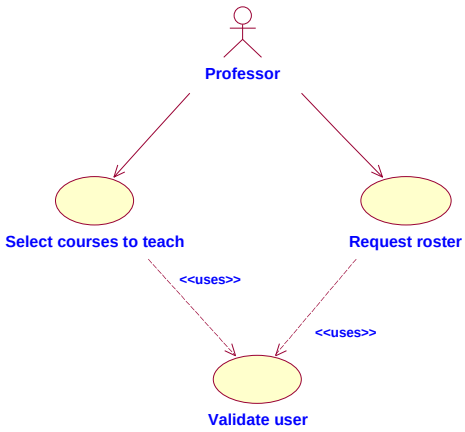
- **includes -** known as **uses** in earlier UML standards
- **Use-case generalisation**
- **extends**
- notation makes use of stereotyping of relationships
  - <<include>>
  - <<extend>>

include (or uses)

- chunk of behaviour that is same across different use cases
- e.g. *valuation* from earlier example

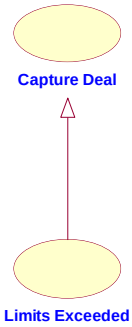


Older UML stereotype



Use-case generalisation

- use case similar to another but does a bit more
- captures alternative scenario
- sort of extends the use case
- alternative functionality put in specialised use case that refers to the base use case
- can override base use case



Extend stereotype

- similar to generalisation except with more rules
- base case must declare *extension points*
- extending use case may extend one or more extension points – indicated by text on line joining use cases

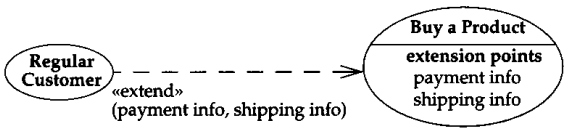


Figure 3-3: *Extend Relationship*

Extend stereotype

- The extension conditions of each use case provide a framework for investigating all the little, niggling things that somehow take up 80% of the development time and budget.

- It provides a look ahead mechanism, so the stakeholders can spot issues that are likely to take a long time to get answers for. These issues should be planned for, so that the answers can be ready when the development team gets around to working on them.

When to use stereotypes

- extend and generalisation allow splitting of a use case
  - elaboration phase: use when use case getting too complicated
  - construction phase: when use case can't be built in 1 iteration

- use *include* for **avoiding repetition** of requirements description
- use *generalisation* for casual description of variation on normal behaviour
- use *extend* for more controlled description of variation on normal behaviour

Business and System use-cases

With focus on user system (software) interaction, analyst can miss situation whew change to business process would be more useful

Can distinguish 2 categories of use-cases:

- system use case: interaction with software
- business use case: how business responds to event or customer or how to meet a user's goal
- Fowler recommends looking at business use cases first and finding system use cases for them later

When to use use-cases

- "*They are an essential tool in requirements capture and in planning and controlling an iterative project*" - Fowler
- capturing use cases is a primary task during elaboration
- must have requirements captured before you can plan for them (like waterfall?)
- can collect all use-cases first and then model
- or explore some use cases and do conceptual modelling together –helps uncover other use cases

USE cases & User Interface

- Each step of a well-written use case should present *actor* goals or intentions (the essence of functional requirements), and normally it should not contain any user interface details, e.g. naming of labels and buttons, UI operations etc., which is a *bad* practice and will unnecessarily complicate the use case writing and limit its implementation.

Advantages

- Easy to understand and facilitate communication
- The list of goal names provides the shortest summary of what the system will offer (even than user stories)
- The main success scenario of each use case provides everyone involved with an agreement as to what the system will basically do and what it will not do. It provides the context for each specific line item requirement (e.g. fine-grained user stories), a context that is very hard to get anywhere else.
- It also provides a project planning skeleton, to be used to build initial priorities, estimates, team allocation and timing.

Advantages

- The use case extension scenario fragments provide answers to the many detailed, often tricky and ignored business questions: “What are we supposed to do in this case?”
- Well-written use cases also serves as a groundwork and guidelines for the design of test cases and user manuals of the system.
- There are obvious connections between the flow paths of a use case and its test cases. Deriving functional test cases from a use case through its scenarios.

Drawbacks/Limitations of use-cases

- may miss requirements if too much emphasis is put on finding actors and their use cases
  - use case analysis and conceptual modelling may help here
- use cases are not well suited to capturing non-interaction based requirements of a system
  - such as algorithm or mathematical requirements or real time embedded requirements)
- not well suited to capturing non-functional requirements (such as platform, performance, timing, or safety-critical aspects). These are better specified declaratively elsewhere.
- no fully standard definitions of use cases

Drawbacks: Use cases and UI design

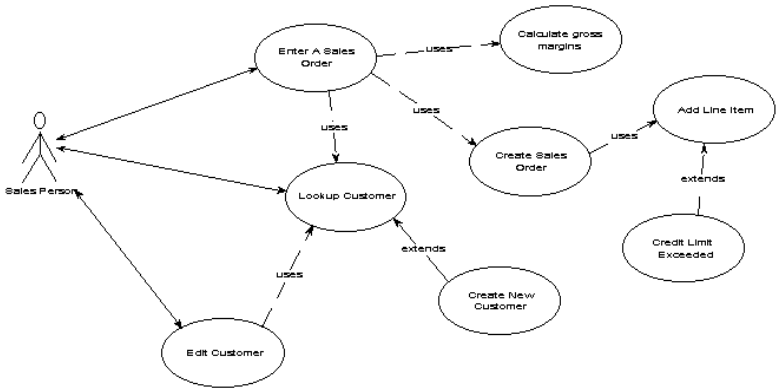
- Use case developers often find it difficult to determine the level of user interface (UI) dependency to incorporate in a use case. While use case theory suggests that UI not be reflected in use cases, it can be awkward to abstract out this aspect of design, as it makes the use cases difficult to visualise.

Drawbacks/Limitations of use-cases

- danger of building system which is not object oriented. Objects not primary – abuse by decomposition
  - in rush to deliver the use case in current iteration, developer may lose sight of the OO architecture, thus
  - could lead to functionality driven design
  - end up with top-down, function-oriented, unmaintainable, inflexible system

Use-case abuse example

Functional decomposition



Drawbacks of use-cases

- abuse by GUI
  - looks like nearly everything is done
  - may be significant gap between effort to put GUI together and the behind the scenes code
  - indications of false progress, makes negotiation difficult

Finally

- In Agile development, especially Extreme Programming, simpler user stories are preferred to use cases
- Craig Larman stresses that "use cases are not diagrams, they are text"

References

- https://en.wikipedia.org/wiki/Use_case
- http://alistair.cockburn.us/Use+cases