Requirements analysis and system specification

Fall 2012

Waterfall model – the traditional approach to analysis and design

Waterfall model is a sequential design process, often used in software development processes, in which progress is seen as flowing steadily downwards (like a waterfall) through the phases of Conception, Initiation, Analysis, Design, Construction, Testing, Production Implementation and Maintenance.

(See Wiki http://en.wikipedia.org/wiki/Waterfall_model)

The waterfall development model originates in the manufacturing and construction industries: highly structured physical environments in which after-the-fact changes are prohibitively costly, if not impossible. Since no formal software development methodologies existed at the time, this hardware-oriented model was simply adapted for software development.

The waterfall model maintains that one should move to a phase only when its preceding phase is completed and perfected.

The idea behind the waterfall model is "measure twice; cut once,"

Arguments for the Waterfall model

If a program design turns out to be impossible to implement, it is easier to fix the design at the design stage than to realize months later, when program components are being integrated, that all the work done so far has to be scrapped because of a broken design.

This is the central idea behind Big Design Up Front and the waterfall model: time spent early on making sure requirements and design are correct saves you much time and effort later. Thus, the thinking of those who follow the waterfall process goes, make sure each phase is 100% complete and absolutely correct before you proceed to the next phase.

Program requirements should be set in stone before design begins (otherwise work put into a design based on incorrect requirements is wasted).

The program's design should be perfect before people begin to implement the design (otherwise they implement the wrong design and their work is wasted), etc.

Arguments against the waterfall model

Bad idea in practice— impossible for any non-trivial project to finish a phase of a software product's lifecycle perfectly before moving to the next phases and learning from them.

For example, clients may not know exactly what requirements they need before reviewing a working prototype and commenting on it.

They may change their requirements constantly.

Designers and programmers may have little control over this. If clients change their requirements after the design is finalized, the design must be modified to accommodate the new requirements.

This effectively means invalidating a good deal of working hours, which means increased cost, especially if a large amount of the project's resources has already been invested in Big Design Up Front.

Arguments against the waterfall model

Designers may not be aware of future implementation difficulties when writing a design for an unimplemented software product.

It become clear in the implementation phase that a particular part is extraordinarily difficult to implement.

In this case, it is better to revise the design than persist in a design based on faulty predictions, and that does not account for the newly discovered problems.

Steve McConnell, refers to design as a "wicked problem"—a problem whose requirements and limitations cannot be entirely known before completion. The implication of this is that it is impossible to perfect one phase of software development, thus it is impossible if using the waterfall model to move on to the next phase.

Waterfall Problems

David Parnas, in *A Rational Design Process: How and Why to Fake It*, writes:
“Many of the system's details only become known to us as we progress in the system's implementation.

Some of the things that we learn invalidate our design and we must backtrack.”

Expanding the concept above, the project stakeholders (non-IT personnel) may not be fully aware of the capabilities of the technology being implemented. This can lead to what they "think is possible" defining expectations and requirements. This can lead to a design that does not use the full potential of what the new technology can deliver, or simply replicates the existing application or process with the new technology. This can cause substantial changes to the implementation requirements once the stakeholders become more aware of the functionality available from the new technology.

An example is where an organization migrates from a paper-based process to an electronic process. While key deliverables of the paper process must be maintained, benefits of real-time data input validation, traceability, and automated decision point routing may not be anticipated at the early planning stages of the project.

UP is iterative & incremental

- Development is organized in a series of short, fixed-length mini-projects called iterations
- Iterations are also incremental
- Successive enlargement and refinement of a system
- Feedback and adaptation evolve the specification, design and code
- How might iterative development be different from prototyping?
- Output of each iteration need not be experimental or a throw-away prototype
- Each iteration tries to be a production-grade subset of final system.
- Read chap 2 of your textbook

A motto for requirements

- Avoid *“Paralysis by Analysis”* – kills budget without significant benefit
- Classic mistake: Too much time and money wasted in the “fuzzy front end”

Early feedback is worth its weight in gold

- Each iteration involves choosing a small subset of requirements, and quickly designing, implementing and testing
- Early feedback (from users, developers and tests) drives development

Evolutionary requirements

- Requirements are capabilities and conditions to which the system and the project must conform

- A prime challenge of requirements analysis is to find, communicate, and remember what is really needed, in the form that clearly speaks to the client and development team members.

Requirements analysis and system specification

- Why is it one of first activities in software life cycle?
  - Need to understand what customer wants first!
  - Goal is to understand the customer's problem
  - Though customer may not fully understand it!
- Requirements analysis says: "Make a list of the guidelines we will use to know when the job is done and the customer is satisfied."
  - AKA *requirements gathering* or *requirements engineering*

System specification

- System specification says: “Here’s a description of *what* the program will do (not *how*) to satisfy the requirements.”
  - Distinguish requirements gathering & system analysis?
  - A top-level exploration into the problem and discovery
    of whether it can be done and how long it will take

Evolutionary requirements

- Requirements are capabilities and conditions to which the system and the project must conform

- A prime challenge of requirements analysis is to find, communicate, and remember what is really needed, in the form that clearly speaks to the client and development team members.

Functional and non-functional requirements

- **Functional** requirements describe system behaviors
  - **Priority:** rank order the features wanted in importance
  - **Criticality**: how essential is each requirement to the overall system?
  - **Risks**: when might a requirement not be satisfied? What can be done to reduce this risk?
- **Non-functional** requirements describe other desired attributes of overall system—
  - **Product cost** (how do measure cost?)
  - **Performance** (efficiency, response time? startup time?)
  - **Portability** (target platforms?), binary or byte-code compatibility?
  - **Availability** (how much down time is acceptable?)
  - **Security** (can it prevent intrusion?)
  - **Safety** (can it avoid damage to people or environment?)
  - **Maintainability** (in OO context: extensibility, reusability)

FURPS+ model

(Grady 1992)

FURPS is a checklist for requirements:

- Functional (features, capabilities, security)
- Usability (human factors, help, documentation)
- Reliability (frequency of failure, recoverability, predictability)
- Performance (response time, throughput, accuracy, availability, resource usage)
- Supportability (adaptability, maintainability, internationalization, configurability)

What’s with the + in FURPS+?

And don’t forget….

- Implementation (resource limitation, language and tools, hardware)
- Interface (constraints posed by interfacing with external systems)
- Operations (system management in its operational setting)
- Packaging (for example, a physical box)
- Legal (licensing)

What is a requirements specification

- Should say *what*, not *how.* **Why?**
- Correct: does what the client wants, according to specification
  - Ask the client: keep a list of questions for the client
  - Prototyping: explore risky aspects of the system with client
- Verifiable: can determine whether requirements have been met
  - But how do verify a requirement like “user-friendly” or “it should never crash”?
- Unambiguous: every requirement has only one interpretation
- Consistent: no internal conflicts
  - If you call an input "Start and Stop" in one place, don't call it "Start/Stop" in another
- Complete: has everything designers need to create the software
- Understandable: stakeholders understand enough to buy into it
- Modifiable: requirements change!
  - Changes should be noted and agreed upon, in the spec!

Use cases

- First developed by Ivar Jacobson
  - Now part of the UML (though not necessarily object-oriented)
  - Emphasizes user’s point of view
  - Explains everything in the user’s language
- A "use *case*" is a set of cases or scenarios for using a system, tied together by a common user goal
  - Essentially descriptive answers to questions that start with “What does the system do if …”
  - E.g., “What does the auto-teller do if a customer has just deposited a check within 24 hours and there’s not enough in the account without the check to provide the desired withdrawal?”
  - Use case describes what the auto-teller does in that situation
- Use case model = <u>the set of all use cases</u>
- Why are use cases good for brainstorming requirements?

Use Cases

- A use case is a description of a system's behaviour as it responds to a request that originates from outside of that system.
- Use case describes "who" can do "what" with the system in question. The use case technique is used to capture a system's behavioral requirements
- A use case is a description of steps or actions between a user (or "actor") and a software system which leads the user towards something useful.
- The user or actor might be a person or something more abstract, such as an external software system or manual process.

(Source: http://en.wikipedia.org/wiki/Use_case)

Overview of Use Case

- Use cases treat the system as a black box,
- Interactions with the system, including system responses, are perceived as from outside the system.
- Focus on what the system must do, not how it is to be done,
- Avoids making assumptions about how the functionality will be accomplished.

Features of use cases

- Describe what the system shall do to achieve a particular goal.
- Include no implementation-specific language.
- Be at the appropriate level of detail.
- Not include detail regarding user interfaces and screens

What are Use cases?

- Use cases are text stories to discover and record requirements.
- Use case diagram gives a bird’s eye view of the user requirements.
- It influences many aspects of a project, for example, the OOAD.
- It is the input of many other subsequent activities of OOAD.
- We will study how to write use cases, how to draw UML use case diagram with UML.

Use Cases

- The essence of use cases is discovering and recording functional requirements by writing stories of using a system to fulfill user goals; that is, cases of use.
- Use cases are not diagrams, they are text.
- UML use case diagram is only secondary.
- Use cases are ***text stories of some actor using a system to meet goals***.

Use Cases

- **Why use case is popular:**
  - **Easy for customers to contribute to the project**
  - **Lower the risk of project failure.**
  - **Emphasize the user goals and perspective.**
  - **Ability to scale both up and down in terms of sophistication and formality.**

Advantages of use cases

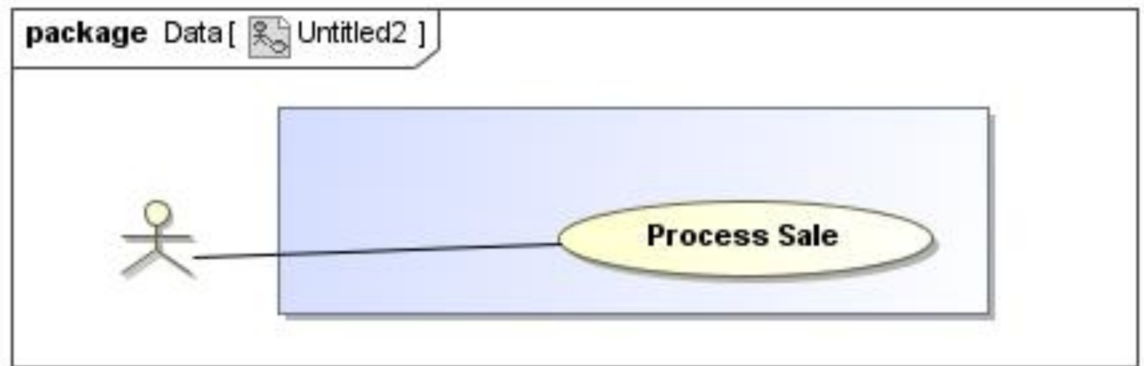
- Systematic and intuitive way to capture functional requirements.
- Facilitates communication between user and system analyst:
  - *Text* descriptions explain functional behavior in user's language
  - *Diagrams* can show relationship between use case behaviors
  - When should we bother with diagrams?
- Use cases can drive the whole development process:
  - Analysis to understand what user wants with use cases
  - Design and implementation to realize the use cases
  - Help with early design of UI prototype
  - Set up test plans
  - Help with writing a user manual

Process sale

…

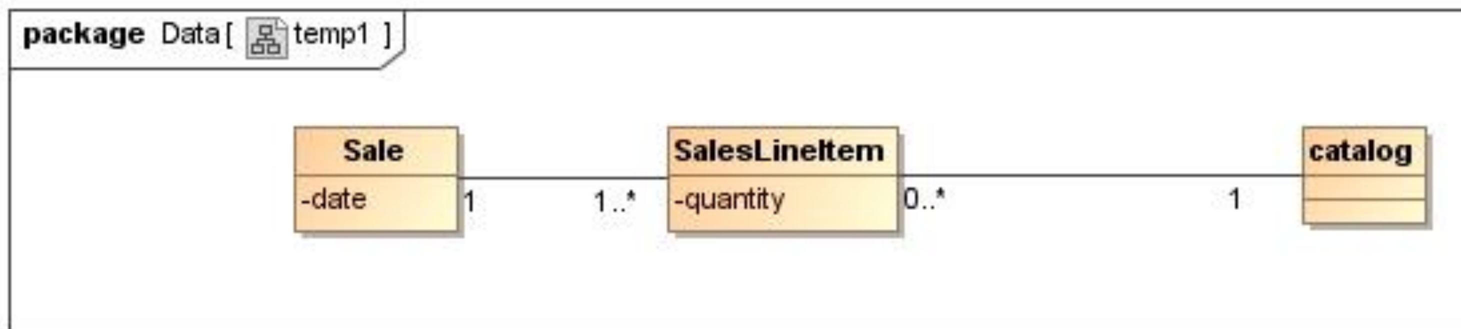
1) Customer arrives
2) cashier starts new sale
3) cashier enters item identifier and quantity
4) System records sale line item

……



Use Case diagram

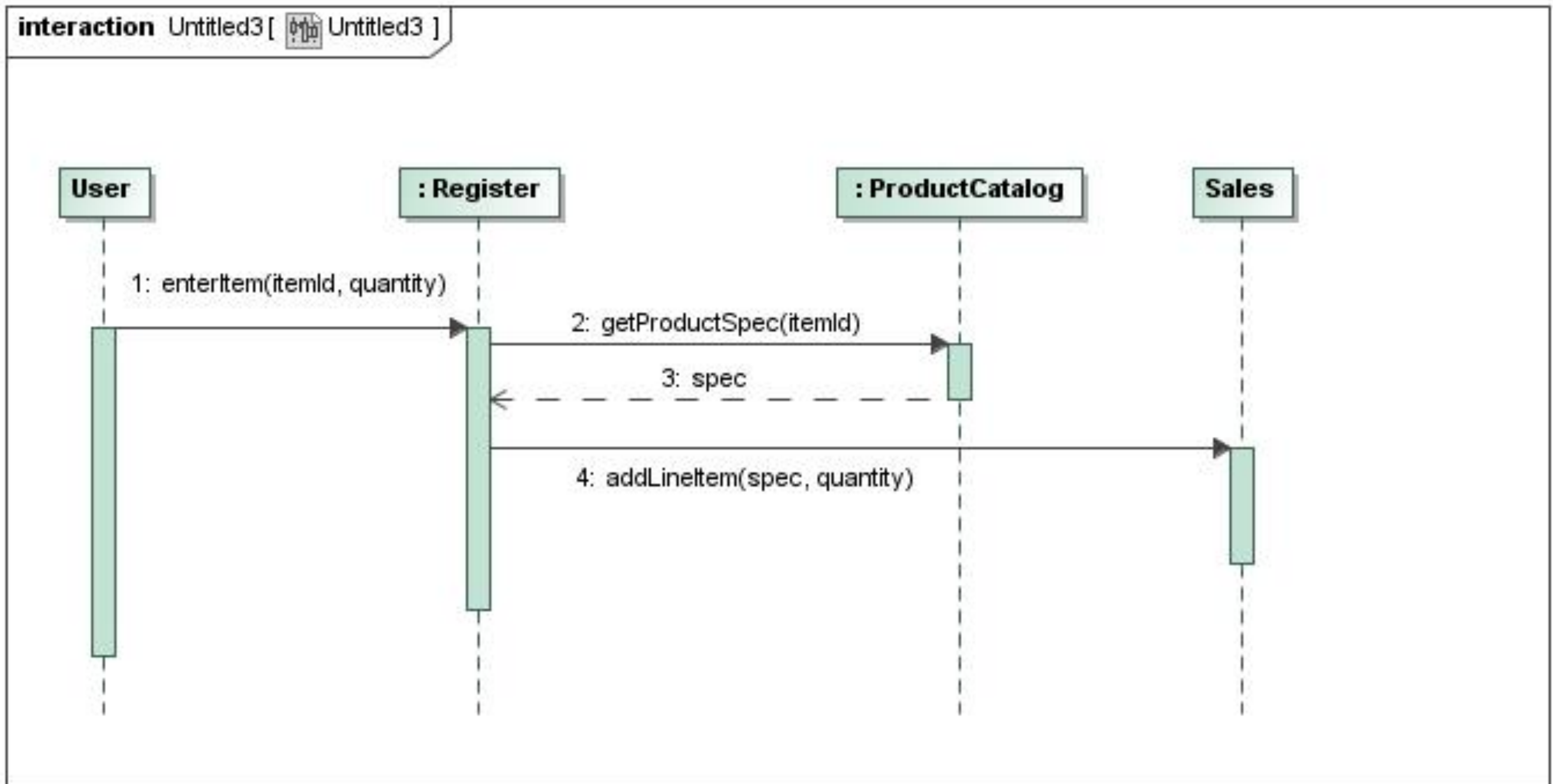
Use case



Domain model

Final result of design

A detailed class diagram with classes such as Register,
Product catalog, Sales, with properties and capabilities of classes as appropriate



Terms used in Use Cases

- An ***actor*** is something with behavior, such as a person (identified by role), computer system, or organization;
  - for example, a cashier.
- A ***scenario*** is a specific sequence of actions and interactions between actors and the system. It is one particular story of using a system, or one path through the use case.
  - For example, the scenario of successfully purchasing items with cash, or the scenario of failing to purchase items because of a credit payment denial.
- **A use case is a *collection of related success and failure scenarios* that describe an actor using a system to support a goal.**

More on Use Cases

- **For example, the "process sales" problem with alternate scenarios could be as follows:**

- **<u>Handle Returns</u>**
- **Main Success Scenario:**
  - **A customer arrives at a checkout with items to return. The cashier uses the POS system to record each returned item …**

- **<u>Alternate Scenarios:</u>**
  - **If the customer paid by credit, and the reimbursement transaction to their credit account is rejected, inform the customer and pay them with cash.**
  - **If the item identifier is not found in the system, notify the Cashier and suggest manual entry of the identifier code (perhaps it is corrupted).**
  - **If the system detects failure to communicate with the external accounting system, …**

Two popular levels of detail in use cases

- Brief use case
  - consists of a few sentences summarizing the use case.
  - It can be easily inserted in a spreadsheet cell, and allows the other columns in the spreadsheet to record priority, duration, a method of estimating duration, technical complexity, release number, and so on.
  - Used during early requirements analysis, to get a quick sense of subject and scope. May take only a few minutes to create.
- Fully dressed use case.
  - is a formal document
  - based on a detailed template with fields for various sections;
  - it is the most common understanding of the meaning of a use case.
  - Used after many use cases have been identified and written in a brief format, then during the first requirements workshop a few (such as 10%) of the architecturally significant and high-value use cases are written in detail.

Brief use case

- ***Process Sale:*** A customer arrives at a checkout with items to purchase. The cashier uses the POS system to record each purchased item. The system presents a running total and line-item details. The customer enters payment information, which the system validates and records. The system updates inventory. The customer receives a receipt from the system and then leaves with the items.

Fully-dressed use case

Typical structure

- Use case name
- Scope
- Level (user-goal or subfunction)
- Actors: Primary, Secondary
- Stakeholders and interests (who cares about this use case, and what do they want?)
- Preconditions (what must be true on start)
- Postconditions or Success guarantee (what must be true on successful completion)
- Main success scenario (typical path, happy path)
- Extensions (alternate scenarios of success and failure)
- Special requirements (related non-functional requirements)
- Technology and data variations list (varying I/O methods)
- Frequency of occurrence
- Miscellaneous

The Template of Fully dressed format

Use Case Section	Comment
Use Case Name	Start with a verb.
Scope	The system under design.
Level	"user-goal" or "subfunction"
Primary Actor	Calls on the system to deliver its services.
Stakeholders and Interests	Who cares about this use case, and what do they want?
Preconditions	What must be true on start, *and* worth telling the reader?
Success Guarantee	What must be true on successful completion, *and* worth telling the reader.
Main Success Scenario	A typical, unconditional happy path scenario of success.
Extensions	Alternate scenarios of success or failure.
Special Requirements	Related non-functional requirements.
Technology and Data Variations List	Varying I/O methods and data formats.
Frequency of Occurrence	Influences investigation, testing, and timing of implementation.
Miscellaneous	Such as open issues.

Fully Dressed Format Explained

- **Scope**
  - The scope bounds the system (or systems) under design. Typically, a use case describes use of one software (or hardware plus software) system; in this case it is known as a system use case.
- Example – NextGen Point of sale application

Fully Dressed Format Explained

- **Level**
  - Use cases are classified as at the ***user-goal level*** or the ***subfunction level***, among others.
  - A user-goal level use case describes the scenarios to fulfill the goals of a primary actor to get work done.
  - A subfunction-level use case describes substeps required to support a user goal, and is usually created to factor out duplicate substeps shared by several regular use cases (to avoid duplicating common text).
  - An example is the subfunction use case Pay by Credit, which could be shared by many regular use cases.
- In this case of Process Sale, it is user goal

Fully Dressed Format Explained

- **Primary Actor**
  - The principal actor that calls upon system services to fulfill a goal.
- For example in the Process sale, the principal actor is the cashier, not the customer (who calls upon the system?)

Fully Dressed Format Explained

- **Stakeholders and Interests List---Important!**
  - The [system] operates a contract between stakeholders, with the use cases detailing the behavioral parts of that contract…The use case, as the contract for behavior, captures the behaviors related to satisfying the stakeholders' interests.
  - The stakeholder interest viewpoint provides a thorough and methodical procedure for discovering and recording all the required behaviors.
- Example : Cashier wants fast entry, no payment error (see book for more)

Fully Dressed Format Explained

- **Preconditions and Success Guarantees (Postconditions)**
  - **Preconditions** state what must always be true before a scenario is begun in the use case.
  - Preconditions communicate noteworthy assumptions that the writer thinks readers should be alerted to.
  - Success guarantees (or **postconditions**) state what must be true on successful completion of the use case - either the main success scenario or some alternate path. The guarantee should meet the needs of all stakeholders.
- Example of precondition: Cashier is authenticated.
- Example of postcondition: Sale is saved, tax correctly recorded, inventory updated.

Fully Dressed Format Explained

- **Main Success Scenario and Steps (or Basic Flow)**
  - "happy path" scenario, or the more prosaic "Basic Flow" or "Typical Flow"
  - It describes a typical success path that satisfies the interests of the stakeholders.
- Guideline
  - defer all conditional handling to the Extensions section.
  - always capitalize the actors' names for ease of identification.

Example of basic flow

Step 1) Customer arrives with purchases and/or services needed.
Step 2) Cashier starts a new flow.
Step 3) Cashier scans a purchase (to get item id) and the quantity.
Step 4) System records sale line item and presents, description,
price, tax and running total.
Cashier repeats steps 3 – 4 until all purchases are recorded.

Step 5) System presents total with tax.
Etc, etc (see book)

Fully Dressed Format Explained

- **Extensions (or Alternate Flows)**
  - Normally comprise the majority of the text.
  - Indicate all the other scenarios or branches, both success and failure.
  - Extensions section was considerably longer and more complex than the Main Success Scenario section; this is common.
  - Extension scenarios are branches from the main success scenario, and so can be notated with respect to its steps 1…N.
  - For example, at Step 3 of the main success scenario there may be an invalid item identifier recorded by the scanner, either because it was incorrectly scanned or unknown to the system. An extension is labeled "3a"; it first identifies the condition and then the response. Alternate extensions at Step 3 are labeled "3b" and so forth (see next slide for a partial example).

Example of extension

- 3a) Invalid item id (the scanned item is not found in the system)
  - i) System signals error and rejects entry.
  - ii) Cashier responds to the error:
    - There is a item id that the cashier can see
      - Cashier manually enters the item ID (e.g., the bar code)
      - System displays description and price.
      - Etc etc

Fully Dressed Format Explained

- **An extension has two parts: the condition and the handling.**
- **Guideline: When possible, write the condition as something that can be detected by the system or an actor. To contrast:**
  - **5a. System detects failure to communicate with external tax calculation system service:**
  - **5a. External tax calculation system not working:**
  - **Which one you would prefer?**

Fully Dressed Format Explained

- **Extensions can include a sequence of steps, as in this example below, which also illustrates notation to indicate that a condition can arise within a range of steps (steps 3-6 in this case):**
  - **3-6a: Customer asks Cashier to remove an item from the purchase:**
    1. Cashier enters the item identifier for removal from the sale.
    2. System displays updated running total.

- **At the end of extension handling, by default the scenario merges back with the main success scenario, unless the extension indicates otherwise (such as by halting the system).**

Fully Dressed Format Explained

- Performing Another Use Case Scenario
  - 3a. Invalid item ID (not found in the example):
    - 1. System signals error and rejects entry.
    - 2. Cashier responds to the error:
      - 2a. …
      - 2c. Cashier performs ***Find Product Help*** to obtain true item ID and price.

(Here ***Find Product Help*** is to perform another use case scenario)

Fully Dressed Format Explained

- **Special Requirements**
  - If a non-functional requirement, quality attribute, or constraint relates specifically to a use case, record it with the use case. These include qualities such as performance, reliability, and usability, and design constraints (often in I/O devices) that have been mandated or considered likely
- Example: No sales must be lost if a database crashes.

Fully Dressed Format Explained

- **Technology and Data Variations List**
- **Examples:**
  - **3a. Item identifier entered by laser scanner or keyboard.**
  - **3b. Item identifier may be any UPC, EAN, JAN, or SKU coding scheme.**
  - **7a. Credit account information entered by card reader or keyboard.**
  - **7b. Credit payment signature captured on paper receipt. But within two years, we predict many customers will want digital signature capture.**

What behavior should we model with a use case?

- Cockburn: Elementary Business Process (EBP) guideline:
  - *“A task performed by one person in one place at one time, in response to a business event, which adds measurable business value and leaves the data in a consistent state.”*

- Naively, can you apply the “boss test” for an EBP?
  - Boss: “What do you do all day?”
  - Me: “I logged in!”
  - Is Boss happy?

- Size: An EBP-level use case *usually* is composed of several steps, not just one or two. Normally several pages of code

Use Case Levels: Applying the Guidelines

- ***Which of following meets EBP & size guidelines?***
  - Negotiate a Supplier Contract – too loosely focussed.
  - Rent Videos
  - Log In – too trivial
  - Start Up – too trivial
- The others *can* also be modeled as use cases
  - But focus first on essential cases (EBP level)

GUIDELINES: Use Case Modeling

- Keep use case names simple: Verb object
  - Deposit money.
- Accomplish a user's goal
  - Invalid PIN is not a use case. Why not?
- Include Secondary Actors (e.g., Bank)
- Avoid ambiguity
  - E.g., in the ATM problem, System could be the machine or the Bank's back-end server
- *Start Up* and *Shut Down* are use cases

Heuristics for writing use case text

- Avoid implementation specific language in use cases, such as IF-THEN-ELSE or GUI elements or specific people or depts
  - Which is better: “The clerk pushes the OK button.” or: “The clerk signifies the transaction is done.”?
  - The latter defers a UI consideration until design.
- Write use cases with the user’s vocabulary, the way a users would describe performing the task
- Use cases never initiate actions; actors do.
  - Actors can be people, computer systems or any external entity that initiate an action.
- Use case interaction produces something of value to an actor
- Create use cases & requirements incrementally and iteratively
  - Start with an outline or high-level description
  - Work from a vision and scope statement
  - Then broaden and deepen, then narrow and prune

More use case guidelines

- A use case diagram is not a flow chart
- Steps in the use case (such as enter PIN) are not necessarily use cases.
- Keep each step and alternative simple; e.g., don't validate PIN and balance in same step (and same alternative scenario)
- Transactions (such as deposit money and withdraw cash) are candidate use cases.

Example to be discussed

- A customer rents a video (or more than one video) from a catalog store

  (so that the videos are sent by post).

Brief Use Case format

Brief format narrates a story or scenario of use in prose form, e.g.:

Rent Videos. A Customer arrives with videos to rent. The Clerk enters their ID, and each video ID. The System outputs information on each. The Clerk requests the rental report. The System outputs it, which is given to the Customer with their videos.

Fully dressed Use Case (from Fowler & Scott, *UML Distilled*)

Use Case: Buy a Product (Describe user's goal in user's language)

Actors: Customer, System (Why is it a good idea to define actors?)

1. Customer browsers 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

(Did we get the main scenario right?)

Alternative: *Authorization Failure* (At what step might this happen?)

6a. At step 6, system fails to authorize credit purchase
Allow customer to re-enter credit card information and re-try

Alternative: *Regular customer* (At what step might this happen?)

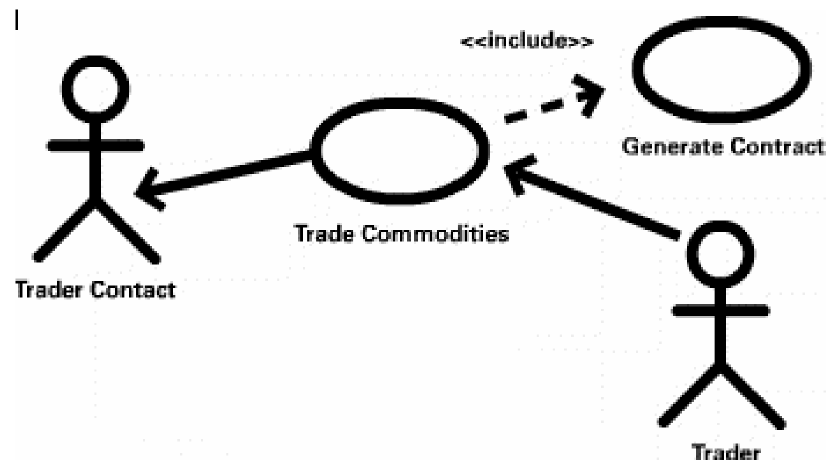
3a. System displays current shipping information, pricing information, and last four digits of credit card information

3b. Customer may accept or override these defaults
Return to primary scenario at step 6

Text and Diagrams

- Use case *text* provides the detailed description of a particular use case
- Use case *diagram* provides an overview of interactions between actors and use cases

Use case diagram



- Use case diagram gives a bird's eye view of use cases for a system
- Stick figures represent **actors** (human or computer in **roles**)
- Ellipses are **use cases** (behavior or functionality seen by users)
- What can user do with the system?
  - E.g., Trader interacts with Trader Contract via a Trade Commodities transaction
- **<<include>>** relationship inserts a chunk of behavior (another use case)
- <<extend>> adds to a more general use case

Use Case Diagrams

- UML has use case diagrams
- Use cases are *text*, not diagrams
- But a *short* time drawing a use case diagram provides a context for:
  - identifying use cases by name
- Use case diagram is not a flow chart!

UML Use Case Diagrams(UCD)

- used to describe the functionality of a system in a horizontal way.
- Only represents the details of individual features of system,
- UCDs are fundamentally different from sequence diagrams or flow charts
- does not represent the order or number of times that the systems actions and sub-actions need to be executed.

Elements of UCD

- UCDs have only 4 major elements:
- The **actors** that the system interacts with,
- the **system** itself,
- the **use cases**, or services, that the system knows how to perform, and
- the lines that represent **relationships** between these elements.

When to use UCD

- Represent the functionality of the system from a top-down perspective (that is, at a glance the system's functionality is obvious, but all descriptions are at a very high level. Further details can later be added to the diagram to elucidate interesting points in the system's behavior.)

- **Example:** A UCD is well suited to the task of describing all of the things that can be done with a database system, by all of the people who might use it (administrators, developers, data entry personnel.)

When not to use a USE case

- You should NOT use UCDs to represent exception behavior (when errors happen) or to try to illustrate the sequence of steps that must be performed in order to complete a task.
- UCDs are not a “blow-by-blow” description of behavior.
- Use Sequence diagrams to show these design features.

- **Example:** A UCD would be poorly suited to describing the TCP/IP network protocol, because there are many exception cases, branching behaviors, and conditional functionality (what happens when a packet is lost or late, what about when the connection dies?)

Who the actors are in a UCD?

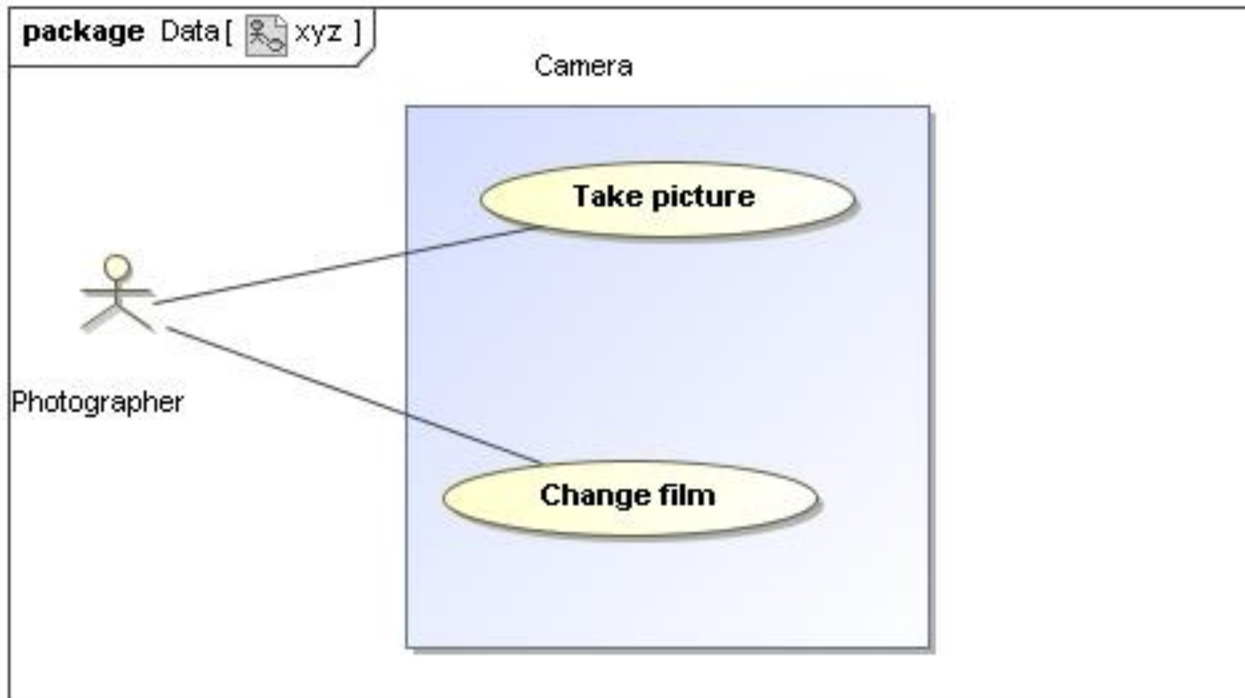
- The actors are typically those entities whose behavior you cannot control or change
- Agents that are not part of the system are actors
- the humans in the system;
- If your system interacts with other systems (databases, servers maintained by other people, legacy systems) treat these as actors, also, since it is not their behavior that you are interested in describing.

- **Example:** When adding a new database system to manage a company's finances, your system will probably have to interface with their existing inventory management software. Since you didn't write this software, don't intend to replace it, and only use the services that it provides, it makes sense for that legacy system to be an **actor**.

What to put in the "System" box?

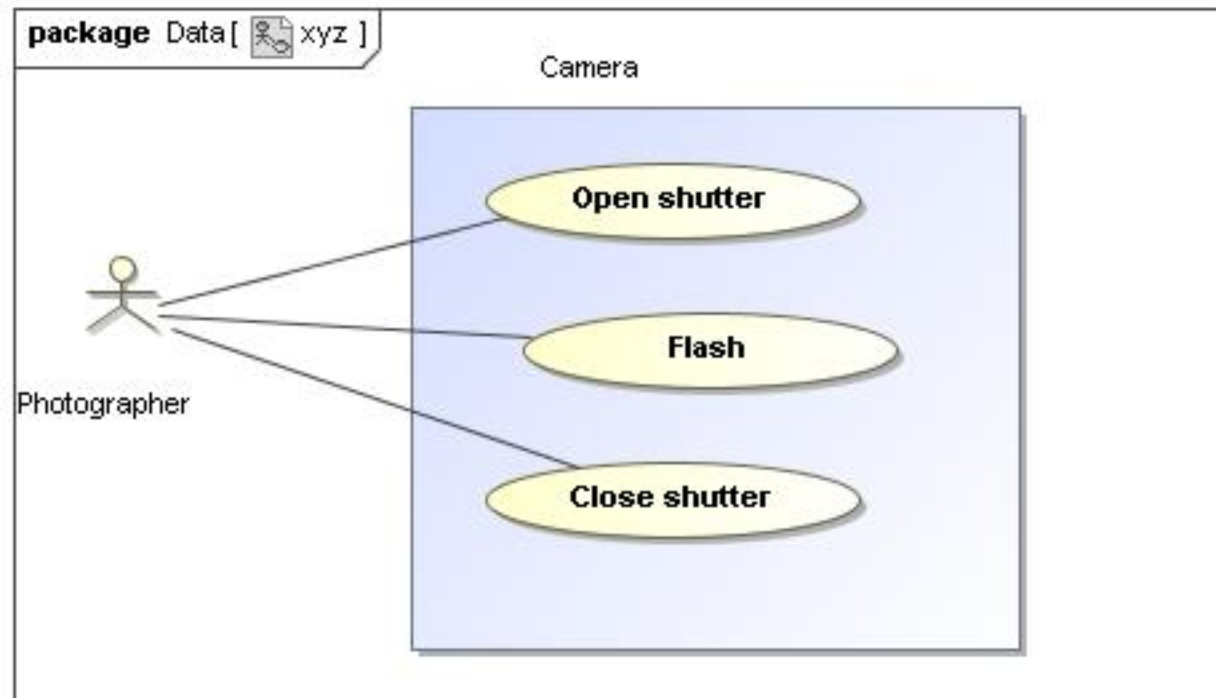
- The system box only appears on the top-level use case diagram (remember that a typical UML Use Case description will be composed of many diagrams and sub-diagrams),
- The system box should contain use case ovals, one for each **top-level** service that your system provides to its actors.
- Any kind of internal behavior that your system may have that is only used by other parts of the system should **not** appear in the system box.
- One useful way to think of these top-level services is as follows: if a use case represents a top-level service, then it should make sense for the actors who interact with it to request **only that service** of your system in a single session

Correct use case for a camera



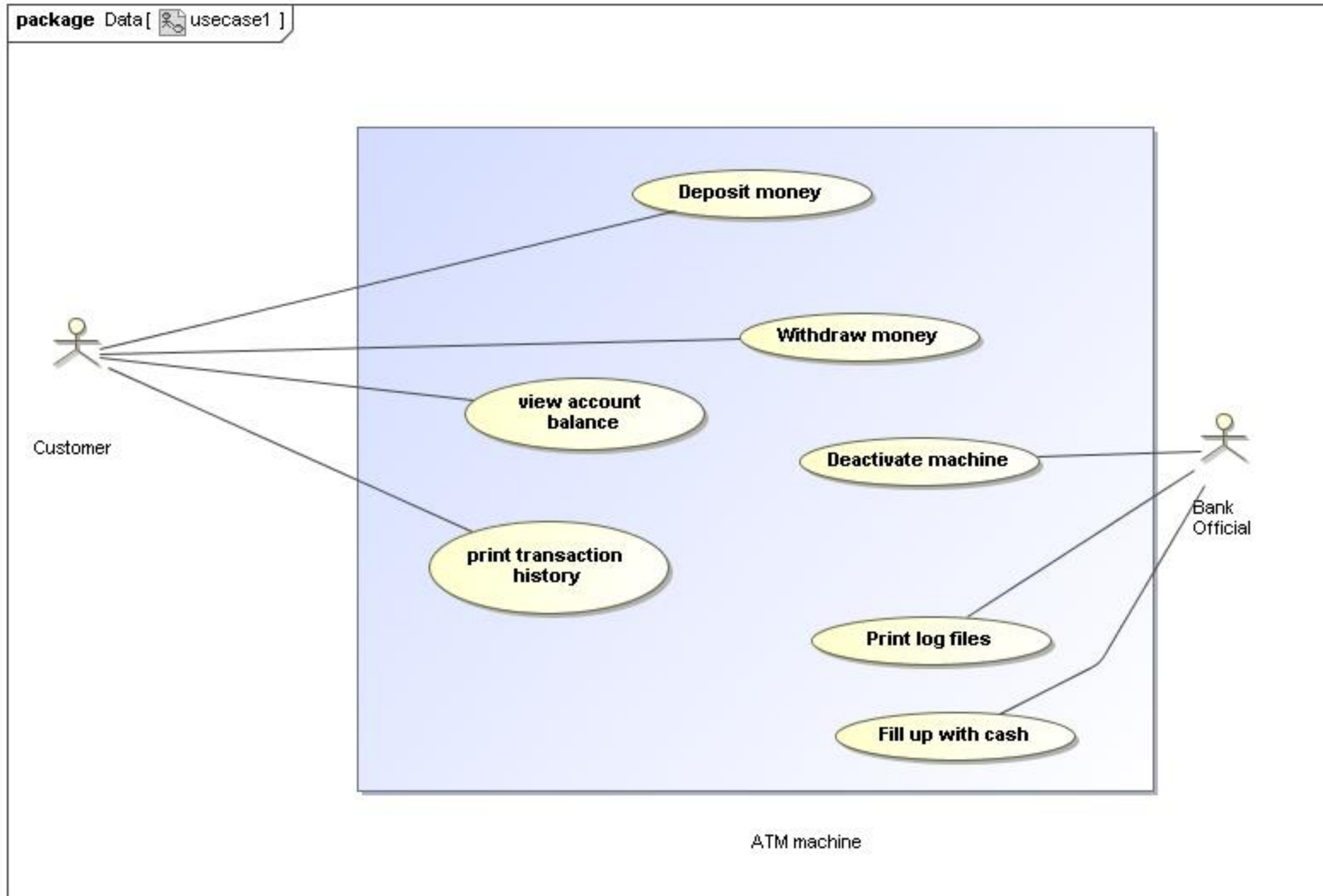
Photographer communicates with (uses) the camera to
-take pictures
- change film

Incorrect use case for a camera



Open shutter, flash etc are behaviors (capabilities) that a camera has – not the primary use by photographers.

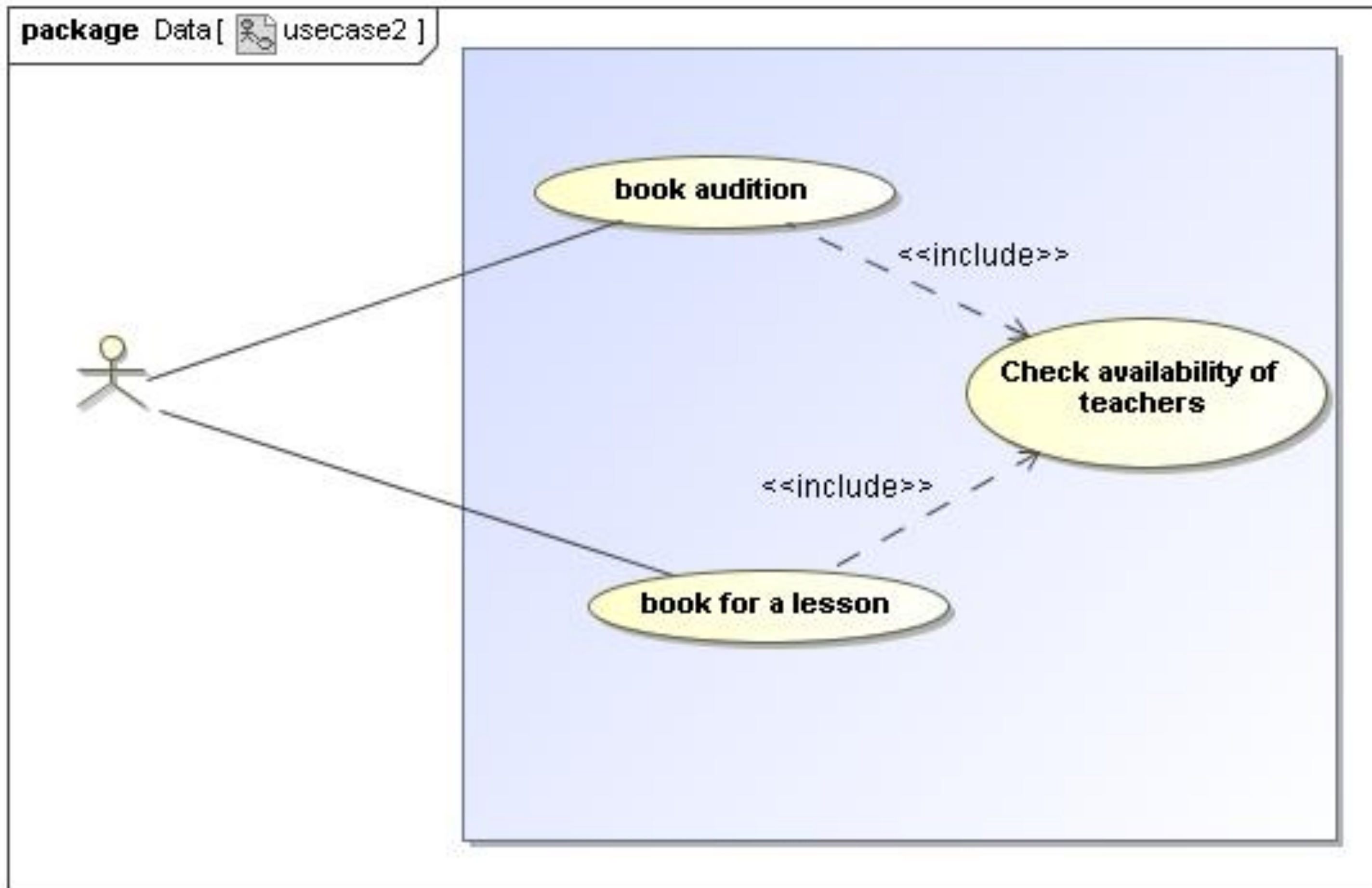
A use case with 2 actors



Including Use Cases

- When I make a cup of tea, I boil the water in a kettle. I also boil the water in a kettle when I make a cup of cocoa. When two or more use cases include the flow of another use case, they are said to include that use case's flow.
- We can illustrate this relationship in a use case diagram by simply drawing a dotted line with an arrow pointing towards the included use case from all the use cases that include it.
- The arrow should have the UML stereotype **<<include>>** to clearly show what kind of relationship it is.

An example of includes



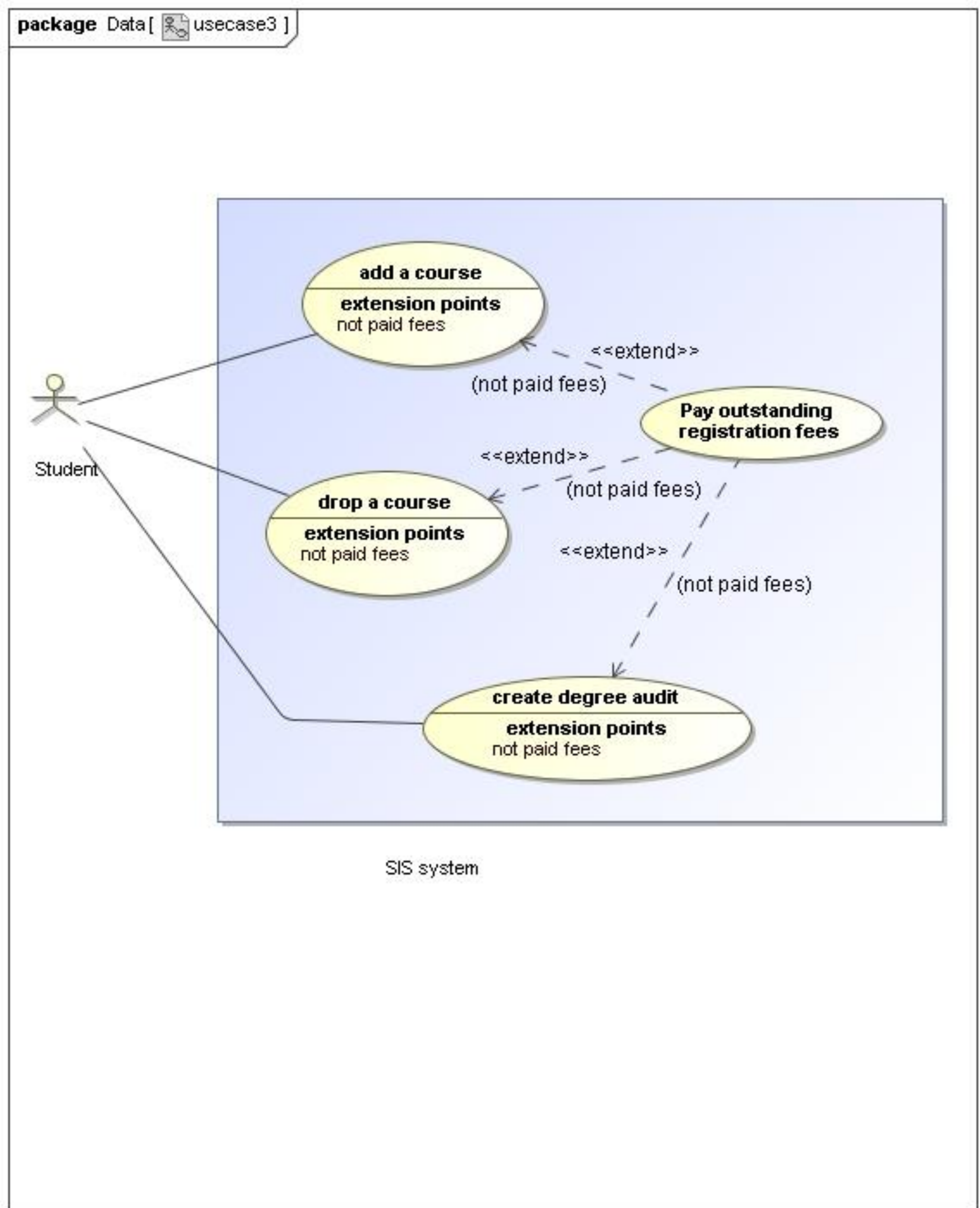
A Use case diagram for users getting an appointment with a music teacher

Extending Use Cases

- Sometimes, one or more use cases will include the flow of another use case, but only under certain conditions. For example, when I make a cup of tea or make a cup of cocoa, I might boil the kettle only if it has not recently been boiled.
- The <<include>> relationship means that the flow of that use case is always included.
- But a **<<extend>>** relationship means that the flow of the extending use case is only included under specific conditions, which must be specified as the **extension point** of the use case being extended.

Example of extends

- An **<<extend>>** relationship means that the flow of the extending use case is only included under specific conditions, which must be specified as the **extension point** of the use case being extended.



Using <<include>> and <<extend>> relationships

- <<include>> means always included.
- <<extend>> means conditionally included.
- <<include>> should point towards the use case being included.
- <<extend>> should point towards the use case(s) being extended.

Applying Use Cases - highlights

- Identify your actors: who will be using the system?
- Identify their goals: what will they be using the system to do?
- Identify key scenarios: in trying to achieve a specific goal, what distinct outcomes or workflows might we need to consider?
- Describe in business terms the interactions between the actor(s) and the system for a specific scenario