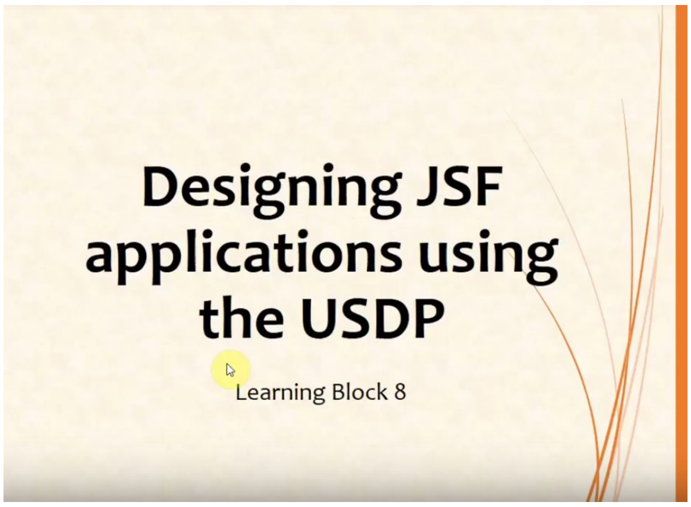
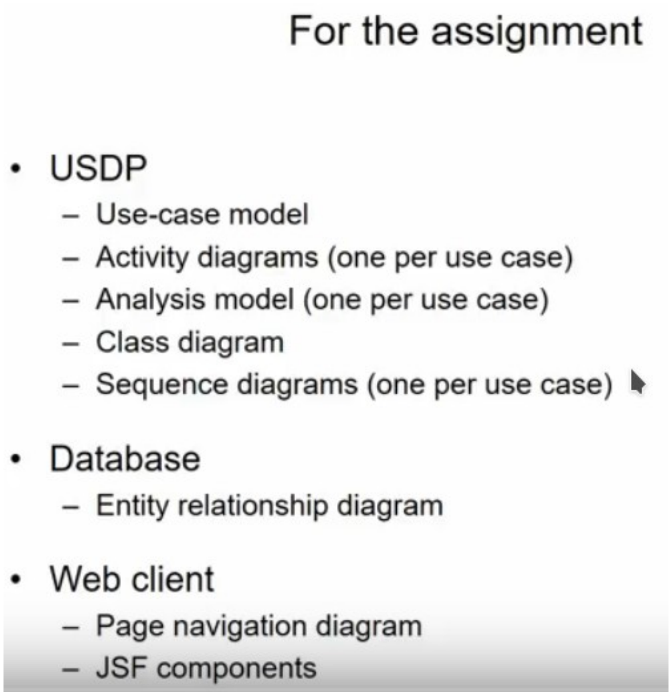
1. The USDP process is made of five stages.
2. Use case model (Problem domain).
3. Activity diagram (Problem domain).
4. Analysis models (Partial solution domain).
5. Class diagram (Solution domain).
6. Sequence diagrams (Solution domain).



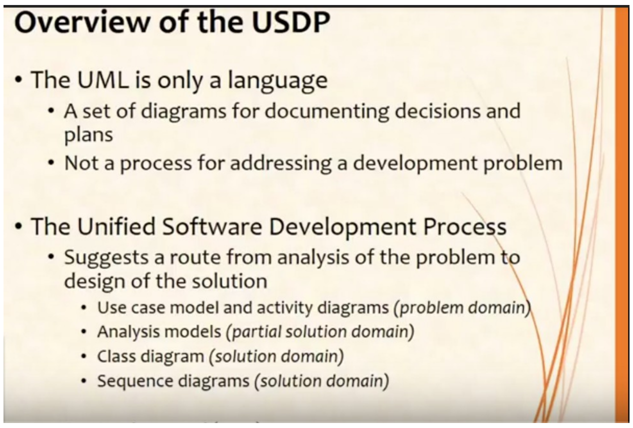
Designing JSF applications using the USDP

Learning Block 8



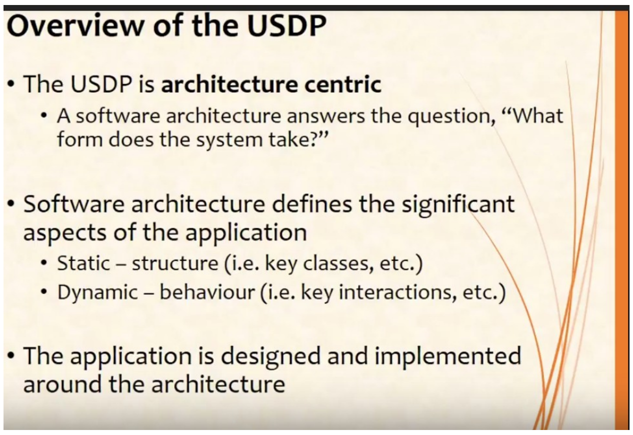
For the assignment

- USDP
  - Use-case model
  - Activity diagrams (one per use case)
  - Analysis model (one per use case)
  - Class diagram
  - Sequence diagrams (one per use case)
- Database
  - Entity relationship diagram
- Web client
  - Page navigation diagram
  - JSF components



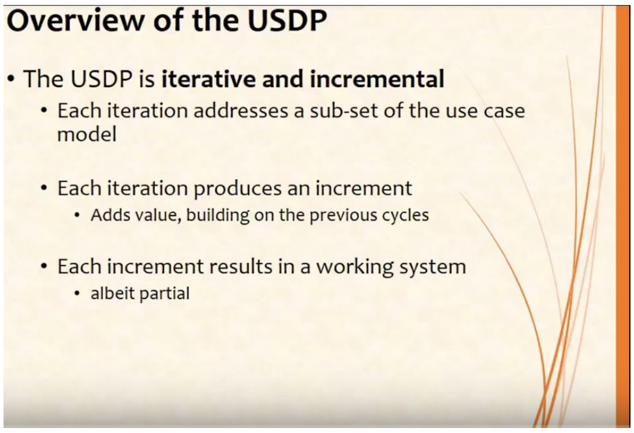
Overview of the USDP

- The UML is only a language
  - A set of diagrams for documenting decisions and plans
  - Not a process for addressing a development problem
- The Unified Software Development Process
  - Suggests a route from analysis of the problem to design of the solution
    - Use case model and activity diagrams *(problem domain)*
    - Analysis models *(partial solution domain)*
    - Class diagram *(solution domain)*
    - Sequence diagrams *(solution domain)*



Overview of the USDP

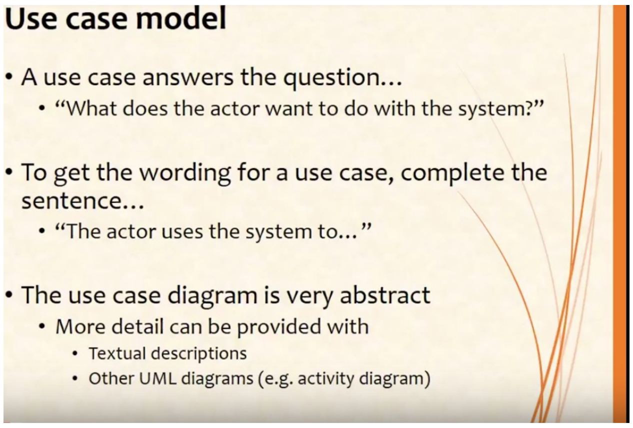
- The USDP is **architecture centric**
  - A software architecture answers the question, "What form does the system take?"
- Software architecture defines the significant aspects of the application
  - Static – structure (i.e. key classes, etc.)
  - Dynamic – behaviour (i.e. key interactions, etc.)
- The application is designed and implemented around the architecture



Overview of the USDP

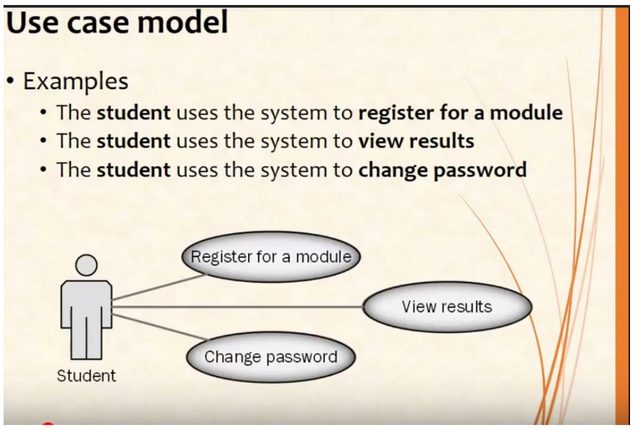
- The USDP is **iterative and incremental**
  - Each iteration addresses a sub-set of the use case model
  - Each iteration produces an increment
    - Adds value, building on the previous cycles
  - Each increment results in a working system
    - albeit partial

Think about the problem domain. The problem domain is defined by the real world situation in which that problem finds itself.
So if it's a problem about library books then we go to the library and we look at what we find in a library and we talk to the librarian we talk to other people who work in or use the library and we start to learn about their viewpoint the users viewpoint of the problem domain we also look for the terminology.



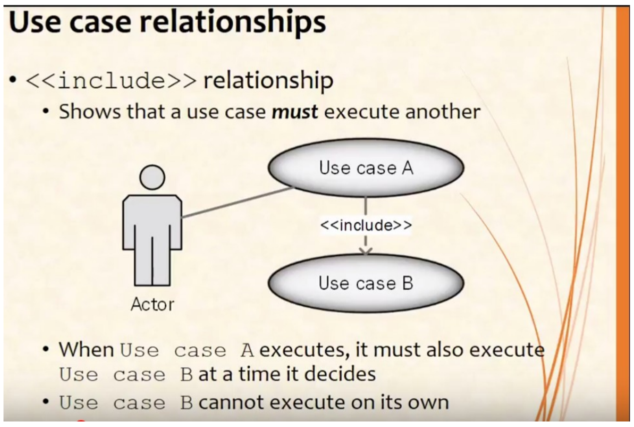
Use case model

- A use case answers the question…
  - "What does the actor want to do with the system?"
- To get the wording for a use case, complete the sentence…
  - "The actor uses the system to…"
- The use case diagram is very abstract
  - More detail can be provided with
    - Textual descriptions
    - Other UML diagrams (e.g. activity diagram)



Use case model

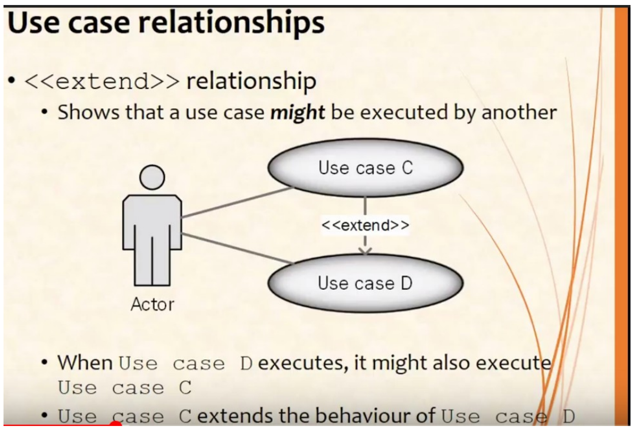
- Examples
  - The **student** uses the system to **register for a module**
  - The **student** uses the system to **view results**
  - The **student** uses the system to **change password**



Use case relationships

- `<<include>>` relationship
  - Shows that a use case ***must*** execute another

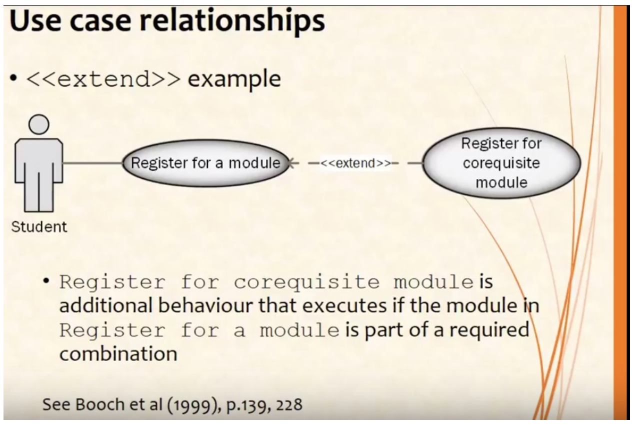
- When `Use case A` executes, it must also execute `Use case B` at a time it decides
- `Use case B` cannot execute on its own



Use case relationships

- `<<extend>>` relationship
  - Shows that a use case ***might*** be executed by another

- When `Use case D` executes, it might also execute `Use case C`
- `Use case C` extends the behaviour of `Use case D`

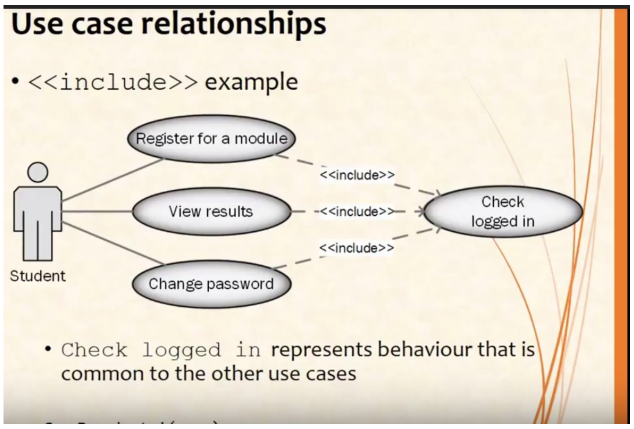


Use case relationships

- `<<extend>>` example

- `Register for corequisite module` is additional behaviour that executes if the module in `Register for a module` is part of a required combination

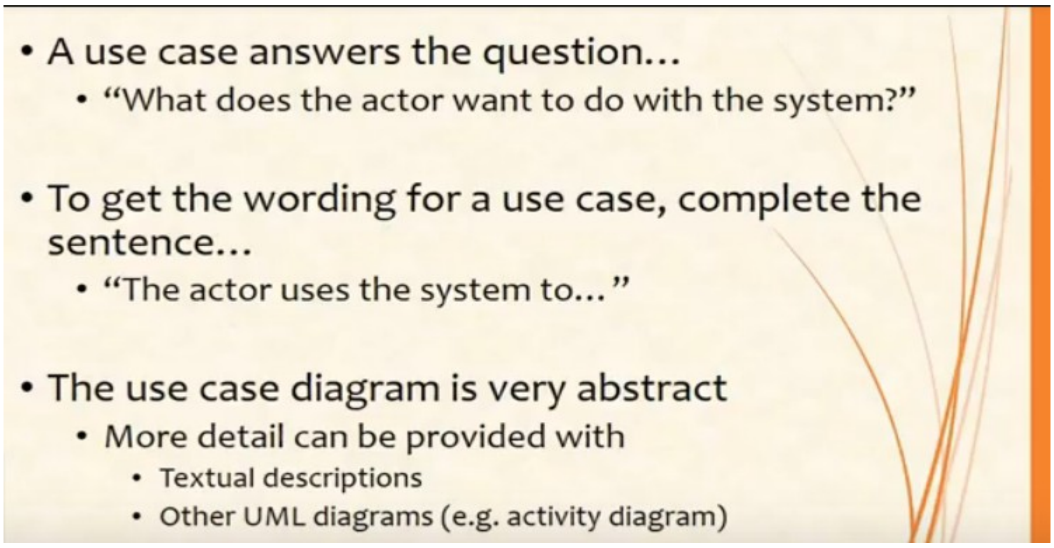
See Booch et al (1999), p.139, 228



Use case relationships

- `<<include>>` example

- `Check logged in` represents behaviour that is common to the other use cases



- A use case answers the question…
  - "What does the actor want to do with the system?"
- To get the wording for a use case, complete the sentence…
  - "The actor uses the system to…"
- The use case diagram is very abstract
  - More detail can be provided with
    - Textual descriptions
    - Other UML diagrams (e.g. activity diagram)

Activity diagrams Tips

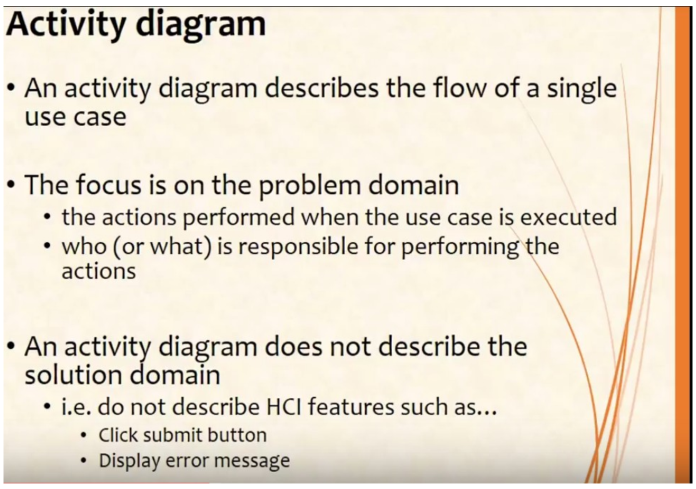
Tips on how to write a good activity diagrams.

Ask yourself What is it that the administrator needs to do to add, Find or Delete Medication from the system.

1. Well first we need to find that Medication if the medication does not exist then we can create the meditation we don't want to add a medication that already exists.

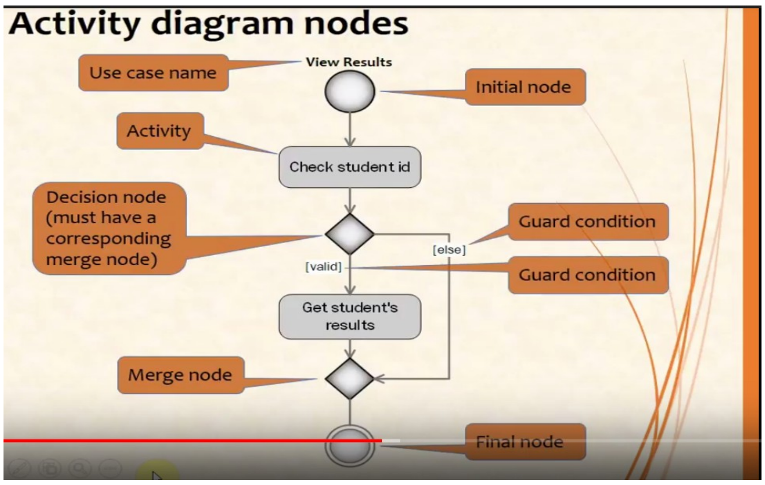
2. Activity diagram gives more detail about a use case.
It describes a given use case and it will represent a flow of activity for that use case that it describes,

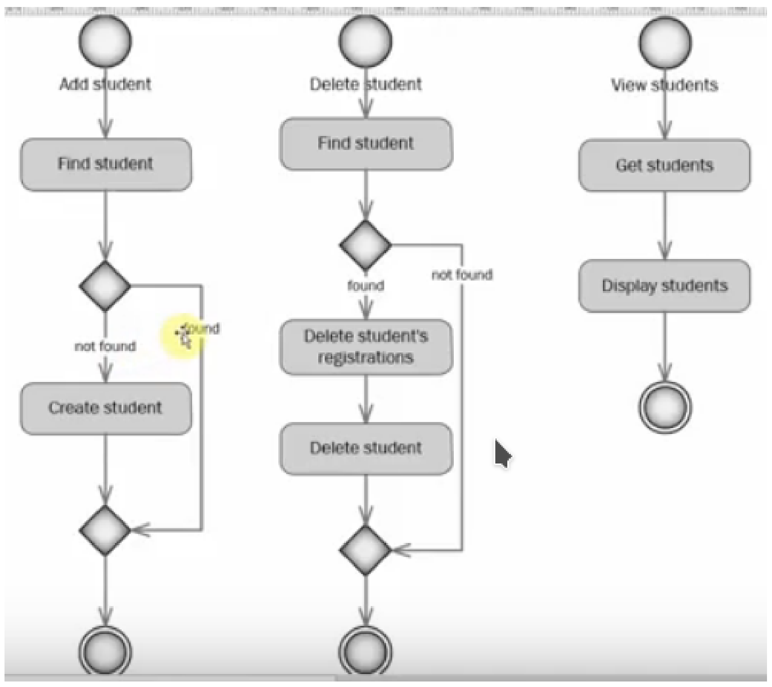
3. we are using the vocabulary and our observations of what is happening in the real world to define the activities that should take place within this use case



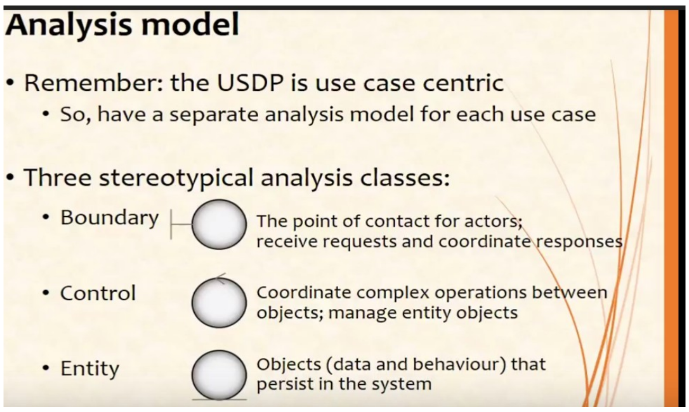
Activity diagram

- An activity diagram describes the flow of a single use case
- The focus is on the problem domain
  - the actions performed when the use case is executed
  - who (or what) is responsible for performing the actions
- An activity diagram does not describe the solution domain
  - i.e. do not describe HCI features such as…
    - Click submit button
    - Display error message





Analysis Model

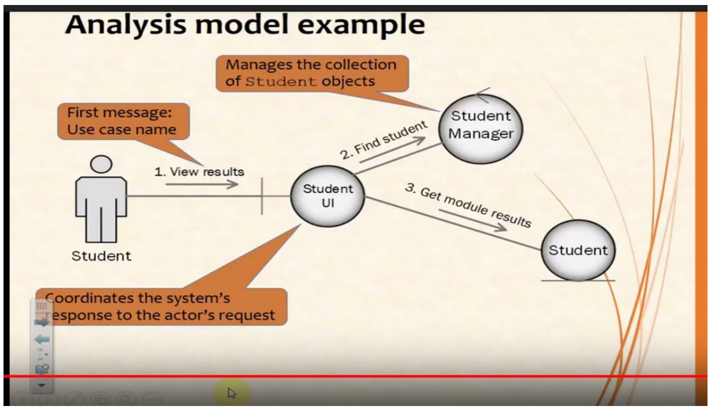


Analysis model

- Remember: the USDP is use case centric
  - So, have a separate analysis model for each use case
- Three stereotypical analysis classes:
  - Boundary — The point of contact for actors; receive requests and coordinate responses
  - Control — Coordinate complex operations between objects; manage entity objects
  - Entity — Objects (data and behaviour) that persist in the system

What we're focusing on with the analysis model are:

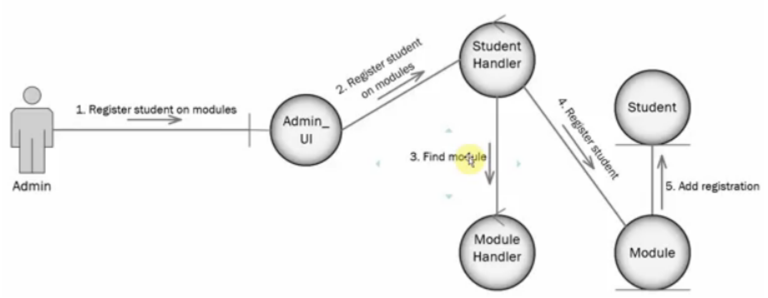
1. Types of classes that we will need for a given use case.
2. The names of those classes and the messages that will be sent between those classes

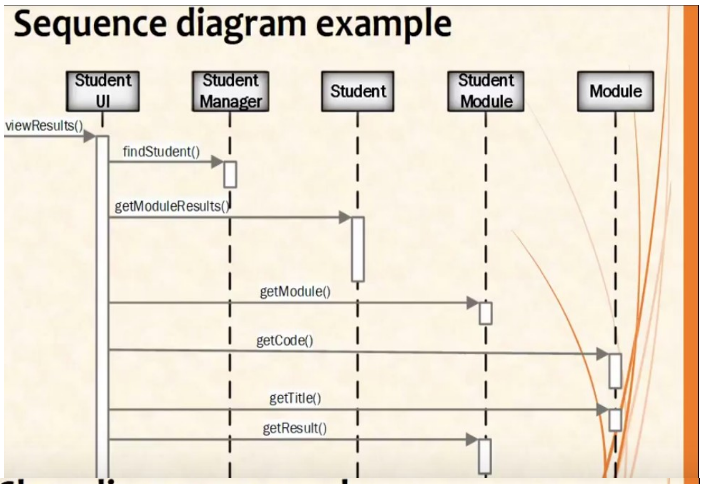


Analysis model example

Manages the collection of Student objects
First message: Use case name
1. View results
2. Find student
3. Get module results
Student Manager
Student UI
Student
Student
Coordinates the system's response to the actor's request

Example:



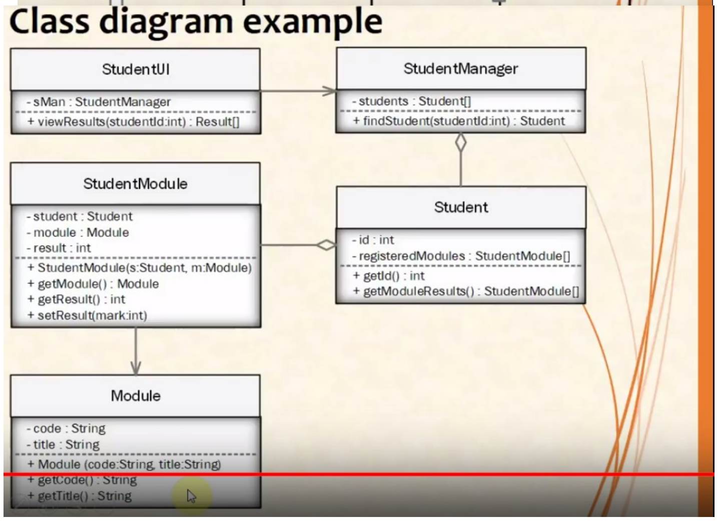


Sequence diagram example

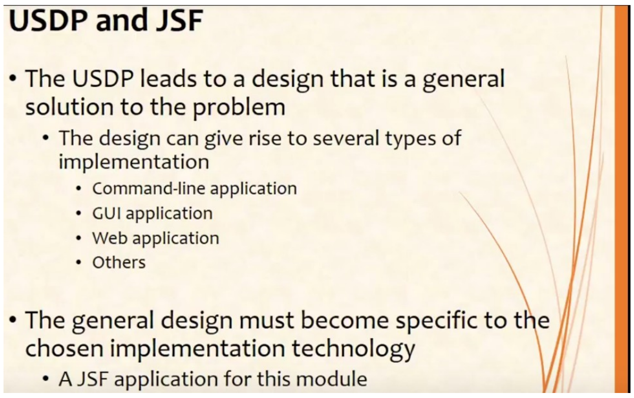
Sequence diagram is an imagined snapshot of what might happen at runtime when view results message comes in from the user to that boundary class .

Example:

when view results happens first we call find student in the student manager then we're going to call get module results in student and then we're going to call get module from the student module collection and then get code from module get title from module and get result from student module.
But that will only make sense after we've decided on what the class diagram structure will be what classes we've got so you can't do this earlier.

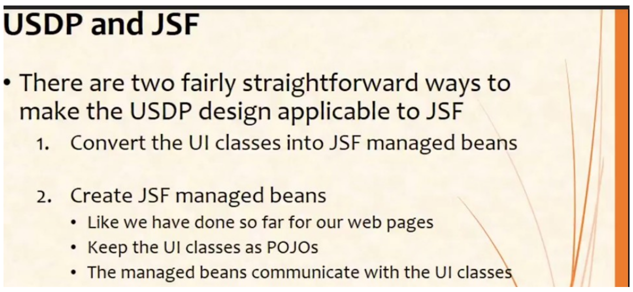


Class diagram example



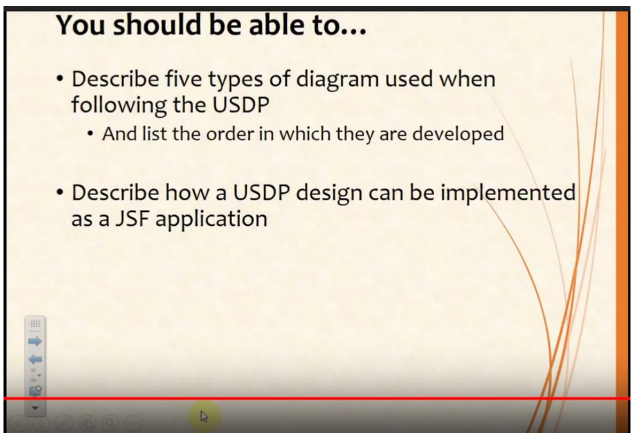
USDP and JSF

- The USDP leads to a design that is a general solution to the problem
  - The design can give rise to several types of implementation
    - Command-line application
    - GUI application
    - Web application
    - Others
- The general design must become specific to the chosen implementation technology
  - A JSF application for this module



USDP and JSF

- There are two fairly straightforward ways to make the USDP design applicable to JSF
  1. Convert the UI classes into JSF managed beans
  2. Create JSF managed beans
     - Like we have done so far for our web pages
     - Keep the UI classes as POJOs
     - The managed beans communicate with the UI classes



You should be able to...

- Describe five types of diagram used when following the USDP
  - And list the order in which they are developed
- Describe how a USDP design can be implemented as a JSF application

Step: Find Classes from Use-Case Behavior

Purpose

To identify a candidate set of model elements (analysis classes) which will be capable of performing the behavior described in use cases.

Substeps:

Finding a candidate set of analysis classes is the first step in the transformation of the system from a mere statement of required behavior to a description of how the system will work.

The technique described here uses three different perspectives of the system to drive the identification of candidate classes. The three perspectives are that of the **boundary**between the system and its actors, the information the system uses, and the control logic of the system. The corresponding class stereotypes, boundary, entity and control, are conveniences used during Analysis that disappear in Design.

Identification of classes means just that: they should be identified, named, and described briefly in a few sentences.

Find Boundary Classes

A boundary class intermediates the interface to something outside the system. Boundary objects insulate the system from changes in the surroundings (changes in interfaces to other systems, changes in user requirements, etc.), keeping these changes from affecting the rest of the system.

A system may have several types of boundary classes:

- **User interface classes**- classes which intermediate communication with human users of the system
- **System interface classes**- classes which intermediate communication with other system
- **Device interface classes**- classes which provide the interface to devices (such as sensors), which detect external events

Find User-Interface Classes

Boundary classes representing the user interface may exist from user-interface modeling activities; where appropriate, re-use these classes in this activity. In the event that user-interface modeling has not been done, the following discussion will aid in finding these classes.

There is at least one boundary object for each use-case actor-pair. This object can be viewed as having responsibility for coordinating the interaction with the actor. This boundary object may have **subsidiary** objects to which it delegates some of its responsibilities. This is particularly true for window-based GUI applications, where there is typically one boundary object for each window, or one for each form.

Make sketches, or use screen dumps from a user-interface prototype, that illustrate the behavior and appearance of the boundary objects.

Only model the key abstractions of the system; do not model every button, list and widget in the GUI. The goal of analysis is to form a good picture of how the system is composed, not to design every last detail. In other words, identify boundary classes only for phenomena in the system or for things mentioned in the **flow of events** of the use-case realization.

Find System-Interface Classes

A boundary class which communicates with an external system is responsible for managing the dialogue with the external system; it provides the interface to that system for the system being built.

Example

In an Automated Teller Machine, withdrawal of funds must be verified through the ATM Network, an actor (which in turn verifies the withdrawal with the bank accounting system). An object called ATM Network Interface can be identified to provide communication with the ATM Network.

The interface to an existing system may already be well-defined; if it is, the responsibilities should be derived directly from the interface definition. If a formal interface definition exists, it may be reverse engineered and we need not formally define it here; simply make note of the fact that the existing interface will be reused during design.

Find Device Interface Classes

The system may contain elements that act as if they were external (change value spontaneously without any object in the system affecting them), such as sensor equipment. Although it is possible to represent this type of external device using actors, users of the system may find doing so "confusing", as it tends to put devices and human actors on the same "level". Once we move away from gathering requirements, however, we need to consider the source for all external events and make sure we have a way for the system to detect these events.

If the device is represented as an actor in the use-case model, it is easy to justify using a boundary class to intermediate communication between the device and the system. If the use-case model does not include these "device-actors", now is the appropriate time to add them, updating the Supplementary Descriptions of the Use Cases where appropriate.

For each "device-actor", create a boundary class to capture the responsibilities of the device or sensor. If there is a well-defined interface already existing for the device, make note of it for later reference during design.

Find Entity Classes

Entity classes represent stores of information in the system; they are typically used to represent the key concepts the system manages. Entity objects are frequently passive and persistent. Their main responsibilities are to store and manage information in the system.

A frequent source of inspiration for entity classes are the Glossary (developed during requirements) and a business-domain model (developed during business modeling, if business modeling has been performed).

Find Control Classes

Control classes provide coordinating behavior in the system. The system can perform some use cases without control objects (just using entity and boundary objects)—particularly use cases that involve only the simple manipulation of stored information.

More complex use cases generally require one or more control classes to coordinate the behavior of other objects in the system. Examples of control objects include transaction managers, resource coordinators and error handlers.

Control classes effectively de-couple boundary and entity objects from one another, making the system more tolerant of changes in the system boundary. They also de-couple the use-case specific behavior from the entity objects, making them more reusable across use cases and systems.

Control classes provide behavior that:

- Is surroundings-independent (does not change when the surroundings change),
- Defines control logic (order between events) and transactions within a use case.
- Changes little if the internal structure or behavior of the entity classes changes,
- Uses or sets the contents of several entity classes, and therefore needs to coordinate the behavior of these entity classes.
- Is not performed in the same way every time it is activated (flow of events features several states).

Steps

- Determine whether a Control Class is needed
- Encapsulate the Main Flow of Events and Alternate/Exceptional Flows of Events in separate Control Classes
- Divide Control Classes where two Actors share the same Control Class

Determine whether a Control Class is needed

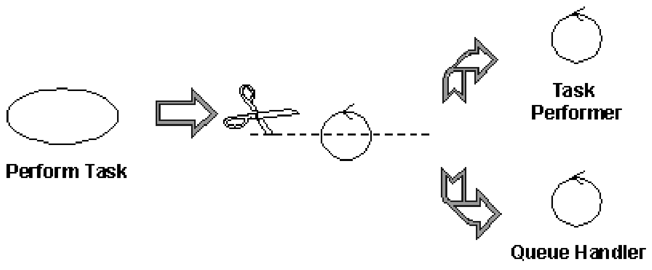
The flow of events of a use case defines the order in which different tasks are performed. Start by investigating if the flow can be handled by the already identified boundary and entity classes. For simple flows of events which primarily enter, retrieve and display, or modify information, a separate control class is not usually justified; the boundary classes will be responsible for coordinating the use case.

The flows of events should be encapsulated in a separate control class when it is complex and consists of dynamic behavior that may change independently from the interfaces (boundary classes) or information stores (entity classes) of the system. By encapsulating the flows of events, the same control class can potentially be re-used for a variety of systems which may have different interfaces and different information stores (or at least the underlying data structures).

Example: Managing a Queue of Tasks

You can identify a control class from the use case Perform Task in the Depot-Handling System. This control class handles a queue of Tasks, ensuring that Tasks are performed in the right order. It performs the next Task in the queue as soon as suitable transportation equipment is allocated. The system can therefore perform several Tasks at the same time.

The behavior defined by the corresponding control object is easier to describe if you split it into two control classes, Task Performer and Queue Handler. A Queue Handler object will handle only the queue order and the allocation of transportation equipment. One Queue Handler object is needed for the whole queue. As soon as the system performs a Task, it will create a new Task Performer object, which will perform the Task. We thus need one Task Performer object for each Task the system performs.



Complex classes should be divided along lines of similar responsibilities

The principal benefit of this split is that we have separated queue handling responsibilities (something generic to many use cases) from the specific activities of task management, which are specific to this use case. This makes the classes easier to understand and easier to adapt as the design matures. It also has benefits in balancing the load of the system, as as many Task Performers can be created as necessary to handle the workload.

Encapsulate the Main Flow of Events and Alternate/Exceptional Flows of Events in separate Control Classes

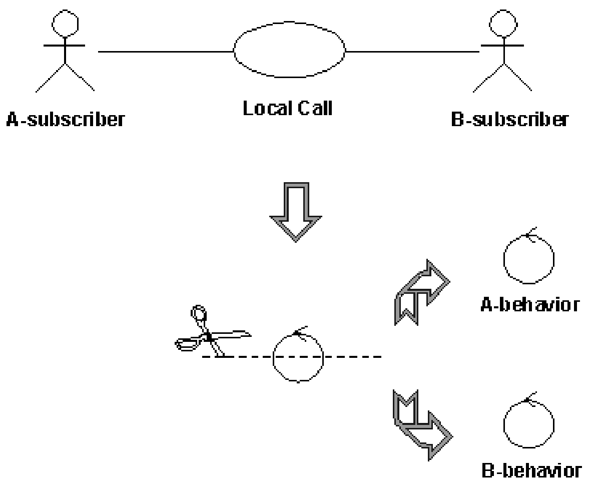
To simplify changes, encapsulate the main flow of events and alternate flows of events in different control classes. If alternate and exception flows are completely independent, separate them as well. This will make the system easier to extend and maintain over time.

Divide Control Classes where two Actors share the same Control Class

Control classes may also need to be divided when several actors use the same control class . By doing this, we isolate changes in the requirements of one actor from the rest of the system. In cases where the cost of change is high or the consequences dire, you should identify all control classes which are related to more than one actor and divide them. In the ideal case, each control class should interact (via some boundary object) with one actor or none at all.

Example: Call Management

Consider the use case Local Call. Initially, we can identify a control class to manage the call itself.



The control class handling local phone calls in a telephone system can quickly be divided into two control classes, A-behavior and B-behavior, one for each actor involved.

In a local phone call, there are two actors: A-subscriber who initiates the call, and B-subscriber who receives the call. The A-subscriber lifts the receiver, hears the dial tone, and then dials a number of digits, which the system stores and analyzes. When the system has received all the digits, it sends a ringing tone to A-subscriber, and a ringing signal to B-subscriber. When B-subscriber answers, the tone and the signal stop, and the conversation between the subscribers can begin. The call is finished when both subscribers hang up.

Two behaviors must be controlled: What happens at A-subscriber's place and what happens at B-subscriber's place. For this reason, the original control object was split into two control objects, A-behavior and B-behavior.

You do not have to divide a control class if:

- You can be reasonably sure that the behavior of the actors related to the objects of the control class will never change, or change very little.
- The behavior of an object of the control class toward one actor is very insignificant compared with its behavior toward another actor, a single object can hold all the behavior. Combining behavior in this way will have a negligible effect on changeability.

Enforce Consistency

- When a new behavior is identified, check to see if there is an existing class that has similar responsibilities, reusing classes where possible. Only when sure that there is not an existing object that can perform the behavior should you create new classes.
- As classes are identified, examine them to ensure they have consistent responsibilities. When an classes responsibilities are disjoint, split the object into two or more classes. Update the collaboration diagrams accordingly.
- If the a class is split because disjoint responsibilities are discovered, examine the collaborations in which the class plays a role to see if the collaboration needs to be updated. Update the collaboration if needed.
- A class with only one responsibility is not a problem, per se, but it should raise questions on why it is needed. Be prepared to challenge and justify the existence of all classes.

Handle Relationships between Use Cases

There are three possible relationships between use cases:

- One use case may include another use case
- One use case may extend another use case
- One use case may specialize another use case, using the generalization relationship.

The existence of these relationships does not require any particular relationships between model elements in either the analysis or design models. In fact, a frequent error in use-case modeling is to overly-structure the use-case model, expecting that it will save effort later. The use-model only describes what the system does, and its structure does not dictate how the use cases are realized.

A given use-case model structure, however, can be used to shape and guide certain decisions about the structure of the analysis and design models. So while we do not want to force the use-case model into considering design issues, we can use the structure of the model to inspire certain architectural decisions.

Handling Include-Relationships

An include-relationship between two use cases allows one use case (the base) to include behavior from another use case (the inclusion) - see Guidelines: Include-Relationship. The usual reasons for including behavior is that the behavior is re-used in several different use cases, or the behavioral details can be effectively encapsulated in the inclusion without a loss of meaning in the base.

The existence of an inclusion implies that there may be one or more objects which collaborate to provide the included behavior. These objects may be encapsulated into a separate package or subsystem.

If the collaborations can be encapsulated behind one or more interfaces, a subsystem is the preferred organizational tool. The subsystem is a kind of package which encapsulates the collaborations of its contained model elements behind one or more interfaces. The only publicly visible model elements of a subsystem are its interfaces - all other contents are private and may be re-placed or renamed without affecting any consumer of the subsystem's services. The interfaces provide the "contract" between the rest of the system and the package/subsystem which realizes the inclusion. This is especially useful in cases where a use case is used in several systems - the subsystem represents the design perspective of a re-usable component.

Note that at this point in object identification, the "subsystems" are still relatively imprecisely defined - they are somewhat loose coupling of collaborating analysis classes, represented as packages, with imprecisely defined interfaces represented as the public classes in these packages. After Activity: Architectural Design these packages will be more formally defined, cast into subsystems with formal interfaces defined.

Examine the **included** use-case to identify the implied objects. The inclusion use case should have **at least** one control object, which serves as the focal point for all other objects which re-use the behavior of the **included** use case. These objects will evolve into the interfaces for the subsystem.

> Example
>
> In many systems, authentication or access authorization represents a common set of behaviors that occur in many places in the system. It is natural to represent the "authentication" sequence of events as an included use case, **Authenticate User**. When identifying objects for this use case, a **control** object**UserAuthenticator** can be identified as having responsibility for carrying our the responsibilities of the use-case. This object can then be used as a place-holder for the behavior of the **Authenticate User** use-case wherever it is included.
>
> When analyzing the behavior of the **Authenticate User** use-case itself, the remainder of the objects needed to realize the use-case will be identified.

In order to help better organize the analysis/design model in which this package/subsystem resides, the realization for the inclusion should be placed within the package or subsystem which realizes the inclusion. This allows related elements to be kept together both logically and physically (when configuration management is considered).

Handling Extend-Relationships

The **extend-relationship** between use-cases allows for a use-case model to which additional functionality can be added without changing the original model - See Guidelines: Extend-Relationship. The extending behavior is most frequently optional or added behavior, or behavior which handles exceptional events outside the normal course of events.

The existence of an extension means that, under certain circumstances, the collaboration of objects which realizes the base use case has a significant alternative collaboration. Several alternatives are possible to handle this in use-case analysis:

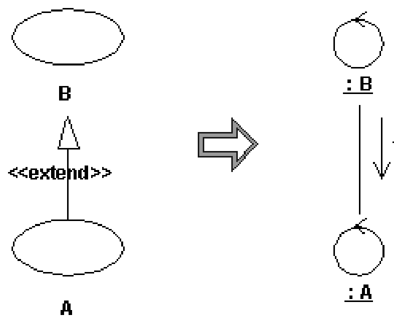
1. Substitute elements in the "base" use-case realization with new elements which provide the extending behavior.
2. Create an alternate use-case realization to represent the extension

In the first case, the extensions are localized to one or more model elements (classes or subsystems). By substituting alternative model elements which provide the same public operations (either directly, in the case of classes, or via interfaces for subsystems) but realize the behaviors differently internally, simple extensions can be realized. Taking this approach may lead to discovery of opportunities for generalization between the replaceable model elements. In the case of two or more subsystems which are substitutable, the subsystems will realize a set of common interfaces.

In the second case, where the extension changes the flow of events of the base use case, a separate use-case realization is needed. The realization need not recapitulate the entire flow of events of the base use; instead it should indicate where the extending use-case realization departs from the base use-case realization, and where it re-joins the base use-case realization. The departure and re-joining points should be expressed in terms of a set of extension points in the base use-case realization (captured as textual annotations in the margins of sequence diagrams).

Modeling Extending Behavior

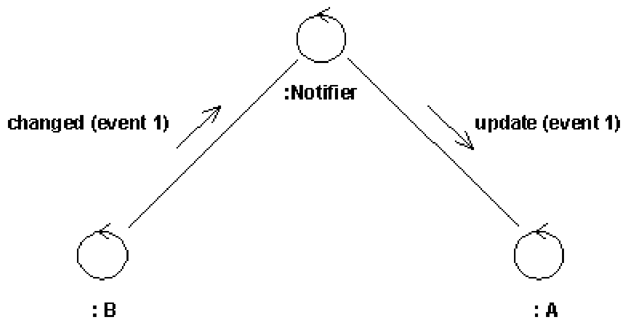
If use case A has an extend association to use case B, a simple solution is to have a link from the control object B to the control object handling A, and then use this link to send messages from B to A when the extension takes place. Unfortunately, this means that the object model will not mirror the use-case model (since in the use case model, use case B is not aware of use case A, but in the analysis model B must know of A in order to send it messages).



The use case A has an extend relation to use case B, handled by a link with messages in the opposite direction of the extend-relation.

There are more disadvantages with this solution. The extend-relation is particularly useful when modeling optional behavior that is inserted into a mandatory use case. If you handle the use-case extension as above, the mandatory control object B will depend on the optional control object A. This is not desirable.

An extend-relation can be represented using an event notification pattern. The event notifier keeps track of which objects must be notified when a particular event occurs. When an event occurs in B that other objects may need to know about, the it tells the notifier that the event has occurred. The notifier then tells all objects which have registered interest in the event that the event has occurred.

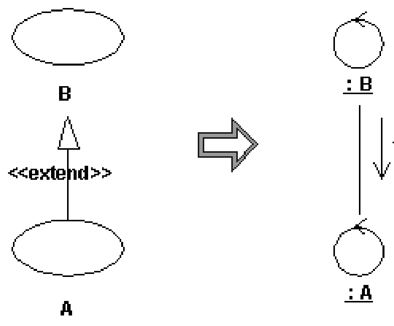


Event notification: object B object informs the support object when an event has occurred. The notifier object notifies the A object.

As a result, we need to describe two types of events for each object: the events the object can generate, and the events (and objects that generate those events) that the object needs to know about. Knowing this information, we can best determine the appropriate way to implement notification in design.

Modeling Extending Behavior

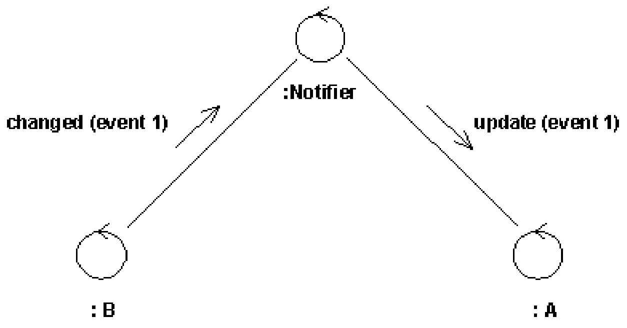
If use case A has an extend relation to use case B, a simple solution is to have a link from the control object B to the control object handling A, and then use this link to send messages from B to A when the extension takes place. Unfortunately, this means that the object model will not mirror the use-case model (since in the use case model, use case B is not aware of use case A, but in the analysis model B must know of A in order to send it messages).



The use case **A** has an extend-relation to use case **B**, handled by a link with messages in the opposite direction of the extend-relation.

There are more disadvantages with this solution. The extend-relation is particularly useful when modeling optional behavior that is inserted into a mandatory use case. If you handle the use-case extension as above, the mandatory control object B will depend on the optional control object A. This is not desirable.

An extend-relation can be represented using an **event notification** pattern. The event notifier keeps track of which objects must be notified when a particular event occurs. When an event occurs in B that other objects may need to know about, the it tells the **notifier** that the event has occurred. The **notifier** then tells all objects which have **registered interest** in the event that the event has occurred.



Event notification: object **B** object informs the support object when an event has occurred. The notifier object notifies the **A** object.

As a result, we need to describe two types of events for each object: the events the object can generate, and the events (and objects that generate those events) that the object needs to know about. Knowing this information, we can best determine the appropriate way to implement notification in design.

Handling Use-Case Generalization

The existence of a generalization between two use cases indicates that one use case (the child) is a specialization of another (the parent - see Guidelines: Use-Case Generalization.   Generalization is a powerful technique for recognizing patterns of behavior which are in turn adapted to specific contexts by the specialized use case.

As with extension, there are two general alternatives for representing use case generalization-specialization in a set of use-case realizations:

1. Substitute elements in the "base" use-case realization with new elements which provide the specializing behavior.
2. Create an alternate use-case realization to represent the specialization

In the first case, the specializations are localized to one or more model elements (classes or subsystems). By substituting alternative model elements which provide the same public operations (either directly, in the case of classes, or via interfaces for subsystems) but realize the behaviors differently internally, simple specializations; the underlying collaborations of the use-case realization remains unchanged. Taking this approach may lead to discovery of opportunities for generalization between the replaceable model elements. In the case of two or more subsystems which are substitutable, the subsystems will realize a set of common interfaces.

As noted above, the subsystems will be represented at this point as packages containing analysis objects. The public classes in these packages will evolve into the interfaces of the subsystems when we consider them in the Activity: Architectural Design.

In the second case, where the specialization adds to or refines the flow of events of the base use case, a separate use-case realization is needed. The realization need not recapitulate the entire flow of events of the base use; instead it should indicate at which points the additional or refining behavior is inserted, and where it re-joins the base use-case realization. The insertion and re-joining points should be expressed in terms of a set of extension points in the base use-case realization (captured as textual annotations in the margins of sequence diagrams).