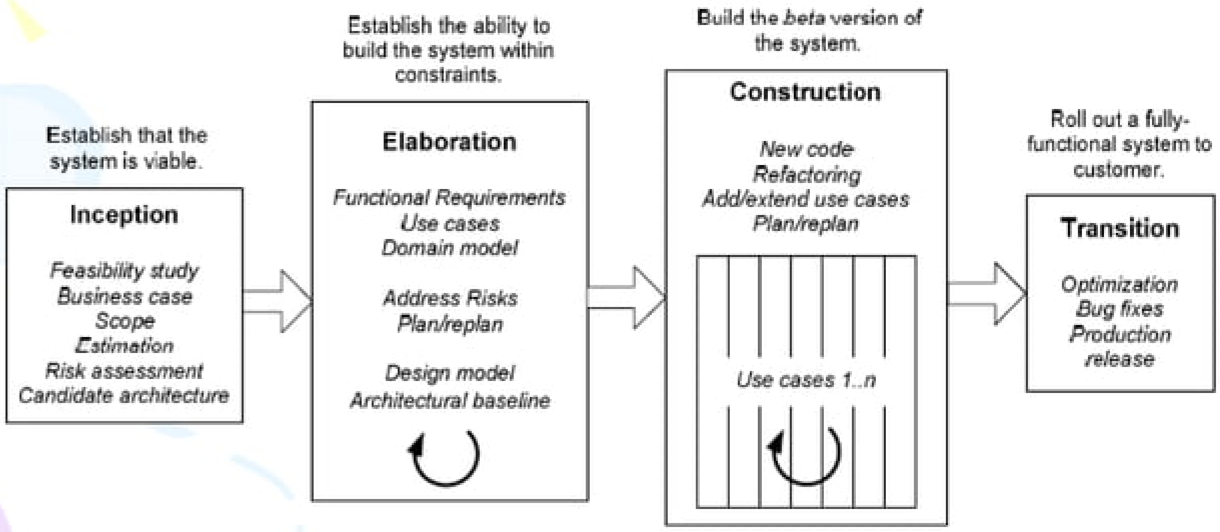
UNIT III CASE STUDY

Case study – the Next Gen POS system, Inception -Use case Modeling - Relating Use cases – include, extend and generalization - Elaboration - Domain Models - Finding conceptual classes and description classes – Associations – Attributes – Domain model refinement – Finding conceptual class Hierarchies - Aggregation and Composition

Phases in UP



Establish that the system is viable.
Inception
Feasibility study
Business case
Scope
Estimation
Risk assessment
Candidate architecture
Establish the ability to build the system within constraints.
Elaboration
Functional Requirements
Use cases
Domain model
Address Risks
Plan/replan
Design model
Architectural baseline
Build the *beta* version of the system.
Construction
New code
Refactoring
Add/extend use cases
Plan/replan
Use cases 1..n
Roll out a fully-functional system to customer.
Transition
Optimization
Bug fixes
Production release

Inception

A feasibility study

Is there a project in there?

What's the vision, scope & business case?

Remarks on Inception

- Supplementary Specifications, Vision, Glossary
- Use Case Model
- Most use cases written in brief format; 10-20% of cases written in fully dressed format
- Most influential and risky quality requirements identified
- First version of the Vision and SS documents

What is Inception?

- Inception is the initial short step to establish a common vision and basic scope for the project
  - Include analysis of perhaps 10% of the use cases,
  - Analysis of the critical non-functional requirement,
  - Creation of a business case, and
  - Preparation of development environment so that programming can start in the following elaboration phase

Vision: What do we want?

Scope: What do we include and not include?

Business case: Who wants it and why?

Determine primary scenarios as Use Cases

Questions in Inception

- Projects require a short initial step

  in which <u>the following kinds of questions are explored</u>

  - What is the vision and business case for this project – Feasible?
  - Buy and/or build this system?
  - Rough unreliable range of cost ? Is it $10K, $100K, millions?
  - Should we proceed or stop?
  - Do the stakeholders have basic agreement on the vision of the project, and is it worth investing in serious investigation?

What you get in Inception?

1. Defining the vision
2. Obtaining an order-of-magnitude (unreliable) estimate
   - It requires doing some requirements exploration.

- Purpose of the inception phase is
  - Not to define all the requirements or generate a believable estimate or project plan

- Most requirements analysis occurs during the elaboration phase,
  - In parallel with early production quality programming and testing

How Long is Inception?

- Inception phase should be relatively short for most projects
  - One or a few weeks long.

- On many projects, if it is more than a week, then the point of inception has been missed
  - It is to decide if the project is worth a serious investigation, not to do that investigation

- Inception phase may include
  - First requirements workshop,
  - Planning for the first iteration, and
  - Then quickly moving forward to elaboration

Requirements Organized in UP Artifacts

- UP offers several requirements artifacts
  - Use case Model: primarily functional requirements
  - Supplementary Specification: everything not in use cases (Non-functional requirements)
  - Glossary: defines noteworthy terms (encompassed the concept of the data dictionary)
  - Vision: summarizes high-level requirements
    - Elaborated in Use-Case Model and Supplementary Specification, and
    - Summarizes the business case for the project.
  - Business rules (also called Domain rules): (e.g., government tax laws)

Summary – A short executive overview document for quickly learning the project's big picture (summary).

Sample Inception Artifacts

Artifact	Comment
Vision and Business Case	Describes the high-level goals and constraints, the business case, and provides an executive summary.
Use-Case Model	Describes the functional requirements. During inception, the names of most use cases will be identified, and perhaps 10% of the use cases will be analyzed in detail.
Supplementary Specification (Non-functional requirements)	Describes other requirements, mostly non-functional. During inception, it is useful to have some idea of the key non-functional requirements that have will have a major impact on the architecture.
Glossary	Key domain terminology, and data dictionary.
Risk List & Risk Management Plan	Describes the risks (business, technical, resource, schedule) and ideas for their mitigation or response.
Prototypes and proof-of-concepts	To clarify the vision, and validate technical ideas.
Iteration Plan	Describes what to do in the first elaboration iteration.
Phase Plan & Software Development Plan	Low-precision guess for elaboration phase duration and effort. Tools, people, education, and other resources.
Development Case	A description of the customized UP steps and artifacts for this project. In the UP, one always customizes it for the project.

These artifacts are only partially completed in this phase.

They will be iteratively refined in subsequent iterations

Isn't That a Lot of Documentation?

- Recall that artifacts should be considered optional.

- Choose to create only those that really add value for the project,
  - Drop those if their worth is not proved

- Since this is evolutionary development,
  - Point is not to create complete specifications during this phase,
  - Create an initial, rough documents
  - **Those are refined during the elaboration iterations**, in response to invaluable feedback from early programming and testing

- **Most UML diagramming will occur in the next phase - elaboration**

Case study - Inception

Introduction

- NextGen POS, with the flexibility to support
  - Varying customer business rules,
  - Multiple terminal and user interface mechanisms, and
  - Integration with multiple third-party supporting systems

Case study - Inception

Business Case

- **Existing POS Products**
  - Not adaptable to customer's business, in terms of varying business rules.
  - They do not scale well as terminals and business increase.
  - None can work in either on-line or off-line mode, dynamically adapting depending on failures.
  - None easily integrate with many third-party systems.
  - None allow for new terminal technologies such as mobile PDAs.
  - There is marketplace dissatisfaction with this inflexible state of affairs, and demand for a POS that rectifies this.

Case study - Inception

Glossary

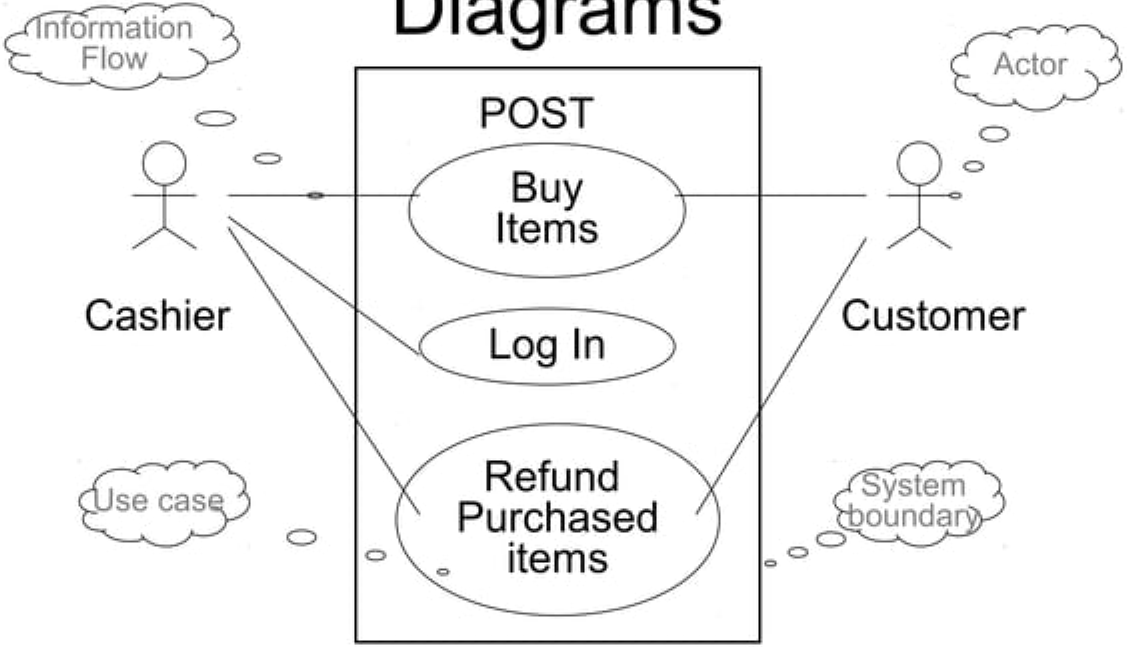
Term	Definition	Format	Aliases
item	A product or service for sale		
Payment authorization	Validation by an external payment authorization service that they will make or guarantee the payment to the seller.		
UPC	Numeric code that identifies a product. Usually symbolized with a bar code placed on products.	12-digit code of several subparts	Universal Product Code

Case study - Inception

Brief format Use Case

- A story of an actor using a system to meet a goal Process Sale:
  1. A customer arrives at a checkout with items to purchase.
  2. The cashier uses the POS system to record each purchased item.
  3. The system presents a running total and line-item details.
  4. The customer enters payment information, which the system validates and records.
  5. The system updates inventory.
  6. The customer receives a receipt from the system and then leaves with the items.

Use case Diagrams



Relating Use cases

-

Introduction

- Objective: **Relate uses cases with include and extend associations, in both text and diagram formats.**
- Use cases can be related to each other.
  - Ex : A subfunction use case "Handle Credit Payment" may be part of several regular use cases, such as "Process Sale" and "Process Rental".

Organizing use cases into relationships

- Has no impact on the behavior or requirements of the system.
- It is an organization mechanism
  - To improve communication & comprehension of the use cases,
  - To reduce duplication of text, and
  - To improve management of the use case documents

The include Relationship

-

The include Relationship

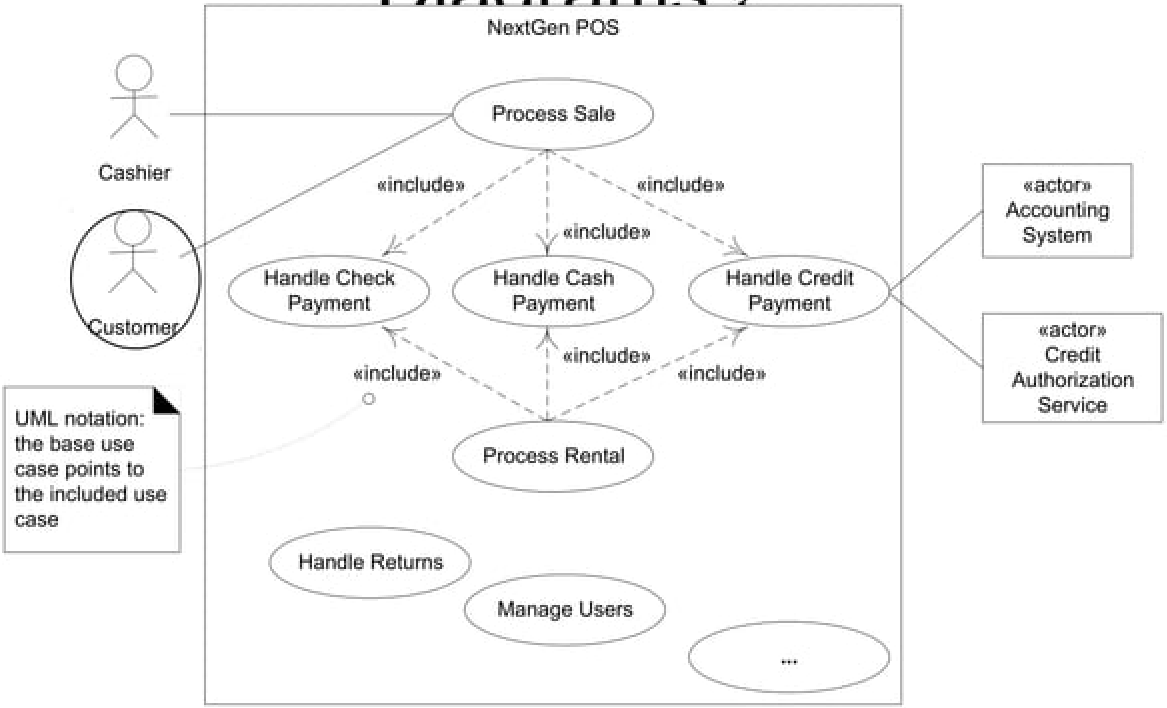
- Some partial behavior across several use cases. (most common)
  - E.g., Paying by credit occurs in several use cases,

    including Process Sale, Process Rental, Contribute to Lay-away Plan.
  - **Rather than duplicate this text, it is desirable**

    To separate it into its own subfunction use case, and indicate its inclusion.
  - This is simply refactoring and linking text to avoid duplication

- **Guideline: Use include**
  - When you are repeating yourself in two or more separate use cases and
  - When you want to avoid repetition

Applying UML: Use Case Diagrams 3



The include Relationship

UC1: Process Sale

Main Success Scenario:

1. Customer arrives at a POS checkout with goods and/or services to purchase

....

7. Customer pays and System handles payment....

Extensions:

7b. Paying by credit: **Include Handle Credit Payment**.

7c. Paying by check: **Include Handle Check Payment....**

UC7: Process Rental

Extensions:

6b. Paying by credit: **Include Handle Credit Payment.**

Underline indicates an included use case

UC12: Handle Credit Payment

Level: Subfunction

Main Success Scenario:

1. Customer enters their credit account information.
2. System sends payment authorization request to an external Payment Authorization Service System, and requests payment approval.
3. System receives payment approval and signals approval to Cashier.

Extensions:

2a. System detects failure to collaborate with external system:

System signals error to Cashier.

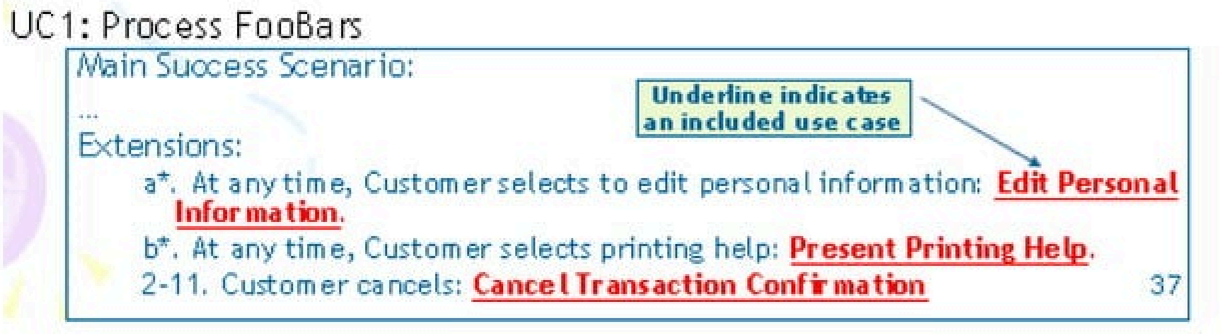
Cashier asks Customer for alternate payment.

Using include with Asynchronous Event Handling

- **Asychronous Events**
  - When a user is able to, at any time, select or branch to a particular window, function, or Web page, or within a range of steps.

- **Basic notation is to use a*, b*, ...** style labels in the Extensions section
  - To imply an extension or event that can happen at any time.

Using include with Asynchronous Event Handling



UC1: Process FooBars

Main Success Scenario:

...

Extensions:

a*. At any time, Customer selects to edit personal information: **Edit Personal Information**.

b*. At any time, Customer selects printing help: **Present Printing Help**.

2-11. Customer cancels: **Cancel Transaction Confirmation**

Underline indicates an included use case

37

Why to use include relationship?

- **There are other relationships**: Extend and Generalization

- Cockburn, an expert use-case modeler,
  - Advises to prefer the include relationship over extend and generalization.

- As a first rule of thumb
  - Always use the include relationship between use cases.

- People who follow “include” have less confusion with their writing than people who mix include with extend and generalizes [Cockburn01].

Concrete/Abstract Use Cases

- **Concrete use case**
  - Initiated by an actor and performs the entire behavior desired by the actor.
  - Are the elementary business process use cases
  - Process Sale is a concrete use case.

- **Abstract use case**
  - Never instantiated by itself; it is a subfunction use case that is part of another use case.
  - Handle Credit Payment is abstract; it doesn't stand on its own, but is always part of another story, such as Process Sale.

Base/Addition Use Cases

- **Base use case**
  - A use case that includes another use case, or is extended / specialized by another use case.
  - Process Sale is a base use case with respect to the included Handle Credit Payment.

- **Addition use case**
  - The use case that is an inclusion, extension, or specialization.
  - Handle Credit Payment is the addition use case in the include relationship to Process Sale.

- Addition use cases are usually abstract. Base use cases are usually concrete.

The extend Relationship

-

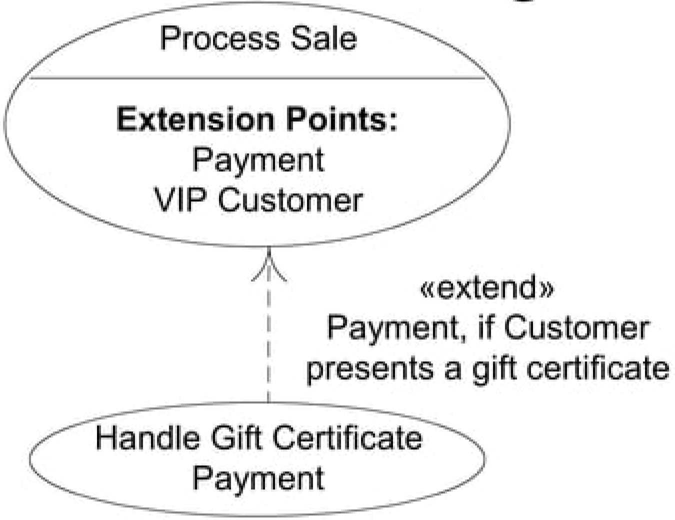
The extend Relationship

- Suppose a use case's text should not be modified
  - Continually modifying the use case with many new extensions and conditional steps is a maintenance headache
  - It has been base-lined as a stable artifact, and can't be touched.

  How to append to the use case?

- Extend relationship allow to create an extending or addition use case,
  - And within it, describe where and under what condition it extends the behavior of some base use case.

Applying UML: Use Case Diagrams 4


Process Sale
Extension Points:
Payment
VIP Customer
«extend»
Payment, if Customer presents a gift certificate
Handle Gift Certificate Payment

UML notation:
1. The extending use case points to the base use case.

2. The condition and extension point can be shown on the line.

The extend Relationship

Point of extension

UC1: Process Sale (the base use case)

Extension Points: VIP Customer, step 1. Payment, step 7.

Main Success Scenario:

1. Customer arrives at a POS checkout with goods and/or services to purchase

....

7. Customer pays and System handles payment....

UC15: Handle Gift Certificate Payment (the extending use case)

Trigger: Customer wants to pay with gift certificate.

Extension Points: Payment in Process Sale.

Level: Subfunction

Main Success Scenario:

1. Customer gives gift certificate to Cashier.
2. Cashier enters gift certificate ID.

A signature quality of extend relationship is that the base use case (Process Sale) has no reference to the extending use case (Handle Gift Certificate Payment)

Use of Extension Point

- Extending use case is triggered by some condition.

- Extension points are labels in the base use case
  - Here, extending use case references as the point of extension

- Step numbering of base use case can change without affecting the extending use case
  - Extension point may simply "At any point in use case X,“

- It is common in systems with many asynchronous events, such as
  - A word processor ("do a spell check now," "do a thesaurus lookup now"), or
  - Reactive control systems.

Prefer Extension Section

- **Practical motivation of using the extend technique** is
  - When it is undesirable for some reason to modify the base use case

Updating the Extensions section is usually the preferred solution, rather than creating complex use case relationships

Elaboration

Build the core architecture,

Resolve the high-risk elements,

Define most requirements, and

Estimate the overall schedule and resources

Introduction

- **Elaboration** is the initial series of iterations during project
  - Refined vision,
  - Core, risky software architecture is programmed and tested
  - **Major risks are mitigated or retired**
  - Majority of requirements are discovered and stabilized
  - More realistic estimates (overall schedule and resources)

- Elaboration often consists of b/w two and four iterations;
  - Each iteration is recommended to be 2~6 week

Not a Design Phase

- **Elaboration is not a design phase**
  - Also <u>not a phase when the models are fully developed</u> in preparation for implementation (Waterfall)

- **During this phase, one is not creating throw-away prototypes**
  - Rather, <u>Code and design are production-quality portions of the final system</u>
  - More commonly it is called the executable architecture or Architectural baseline

Key Ideas & Best Practices in Elaboration

- Do short time boxed risk-driven iterations
- Start programming early
- Adaptively design, implement, & test the core and risky parts of the architecture
- Test early, often, realistically
- Adapt based on feedback from tests, users, developers
- Write most of the use cases and other requirements in detail, through a series of workshops, once per elaboration iteration

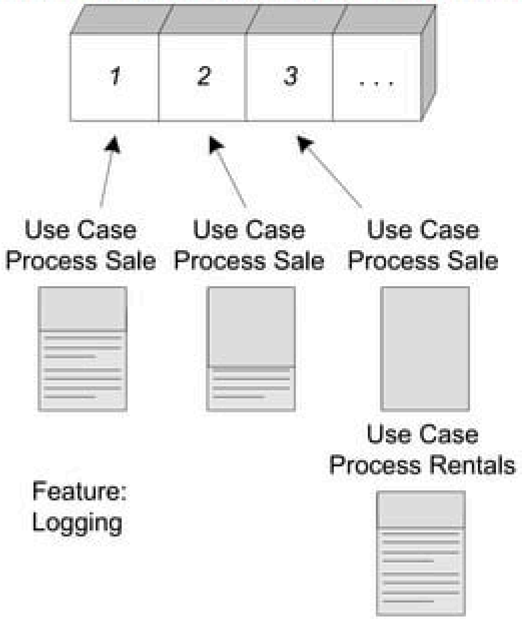
Artifacts

Artifact	Description
Domain Model	**Visual representation of the domain model**
Design Model	**Logical Design Diagrams**
Software Architecture Document	**Summary of key architectural issues and their resolution**
Data Model	**Database schemas, mapping strategies between object and non-object representation**
Use-Case Storyboards, UI Prototypes	**User interface description, usability models...**

Iterations in Elaboration

- In Iterative Development
  - Don't Implement All the Requirements at once
  - **Incremental Development for the Same Use Case Across Iterations**

Use case implementation may be spread across iterations



A use case or feature is often too complex to complete in one short iteration.

Therefore, different parts or scenarios must be allocated to different iterations.

Iteration 1 of Elaboration Phase

- Emphasizes a range of fundamental and core OOA/D skills used in building object systems,
  - Such as assigning responsibilities to objects.

- Iteration 1 is NOT architecture-centric and risk-driven

Iteration 1 of NextGen POS Application

- **Requirements for iteration 1 of the POS application**
  - Implement a basic, key scenario of the Process Sale use case:

    Entering items and receiving a cash payment.
  - Implement a Start Up use case as necessary

    To support the initialization needs of the iteration.
  - Nothing fancy or complex is handled,

    A simple happy path scenario, and the design and implementation to support it.
  - No collaboration with external services, such as a tax calculator or product DB.
  - No complex pricing rules are applied.
  - **Design and implementation of supporting UI, DB, and so forth, would also be done**

Planning the Next Iteration

- Organize requirements and iterations by risk, coverage, and criticality
  - **Risk** includes both technical complexity and other factors, such as uncertainty of effort or usability.
  - **Coverage** implies that all major parts of the system are at least touched on in early iterations perhaps a "wide and shallow" implementation across many components.
  - **Criticality** refers to functions the client considers of high business value.

Planning the Next Iteration

- Use cases or use case scenarios are ranked for implementation.
  - Early iterations implement high ranking scenarios.

Rank	Requirement (Use Case or Feature)	Comment
High	Process Sale Logging …	Scores high on all rankings. Pervasive. Hard to add late. …
Medium	Maintain Users …	Affects security subdomain. …
Low	…	…

Domain Model

A visual representation of
conceptual classes or real situation objects
in a domain

Definition

A domain model is a representation of real-world conceptual classes, not of software components. It is *not a set of diagrams describing software classes*, or software objects with responsibilities.

Introduction

- **A domain model**
  - Most important and classic model in OO analysis.
  - A visual representation of conceptual classes or real situation objects in a domain.
  - Also called conceptual models, domain object models, and analysis object models.
  - "focusing on explaining 'things' and products important to a business domain", such as POS related things.

- **Guidelines**
  - Avoid a waterfall-mindset (big-modeling effort to make "correct" domain model)

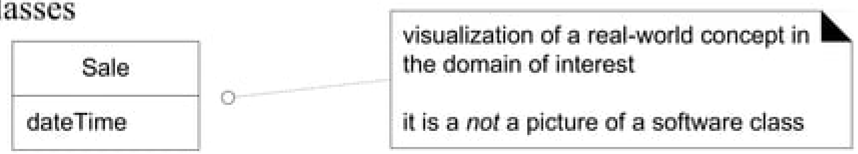
Provides a Conceptual Perspective

- Domain model is illustrated with UML class diagram.
- Domain model provides a conceptual perspective.
  - Domain objects or conceptual classes
  - Associations between conceptual classes
  - Attributes of conceptual classes
- Following elements are not suitable in a domain model
  - **Software artifacts**, such as a window or a DB,

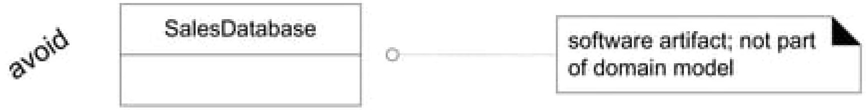
    Unless the domain being modeled is of software concepts, such as a model of graphical user interfaces.
  - **Responsibilities or methods**.

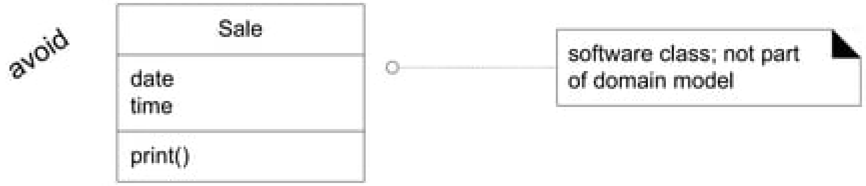
Domain Model 2

- A domain model shows real-situation conceptual classes, not software classes

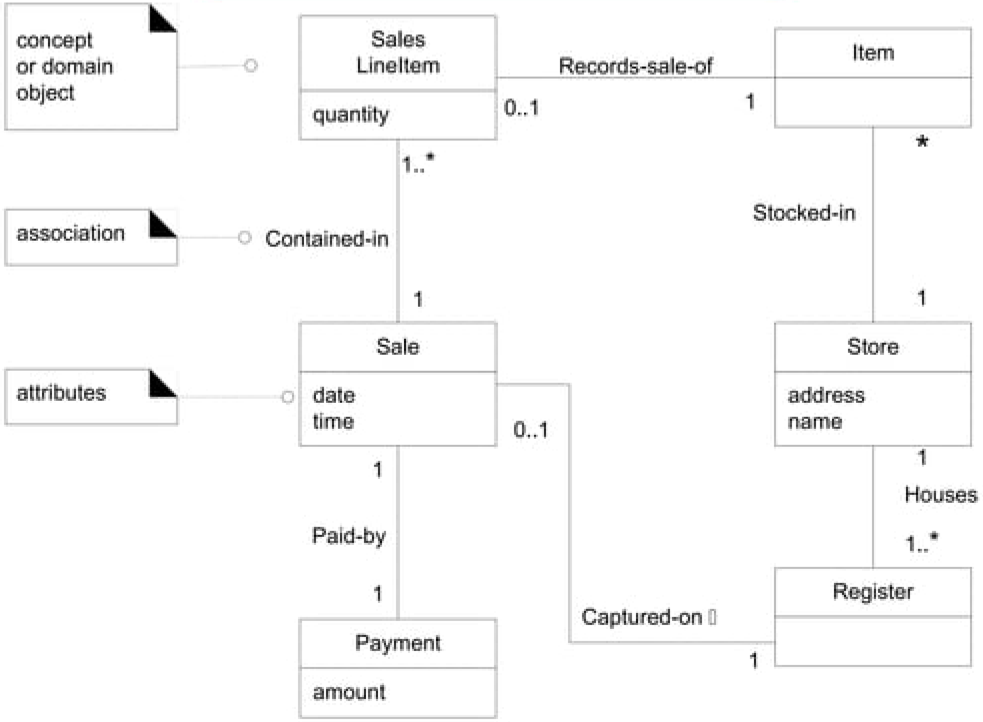

Sale
dateTime
visualization of a real-world concept in the domain of interest
it is a *not* a picture of a software class

- A domain model does not show software artifacts or classes





POS Domain Model



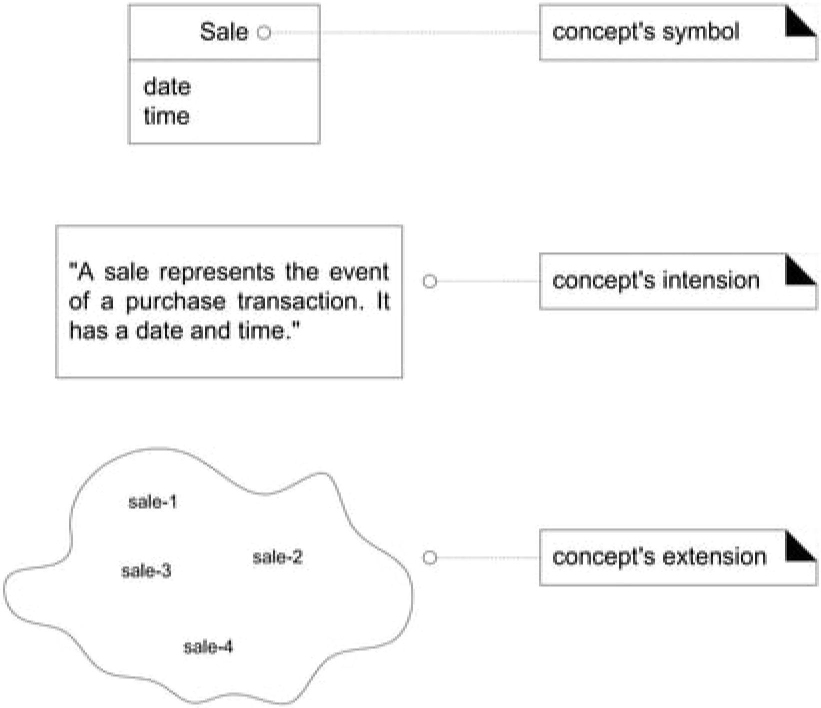
Conceptual Class

- A conceptual class is an idea, thing, or object.
  - Symbol words or images representing a conceptual class.
  - Intension -the definition of a conceptual class.
  - Extension -the set of examples to which the conceptual class applies

- Example of Intension:,
  - Customer may be a person or organization that purchases goods or services

- Example of Extension:
  - Set of all objects to which the concept applies, e.g. the Customer may be " John", Tom"

Domain Model 4



Domain Models and Decomposition

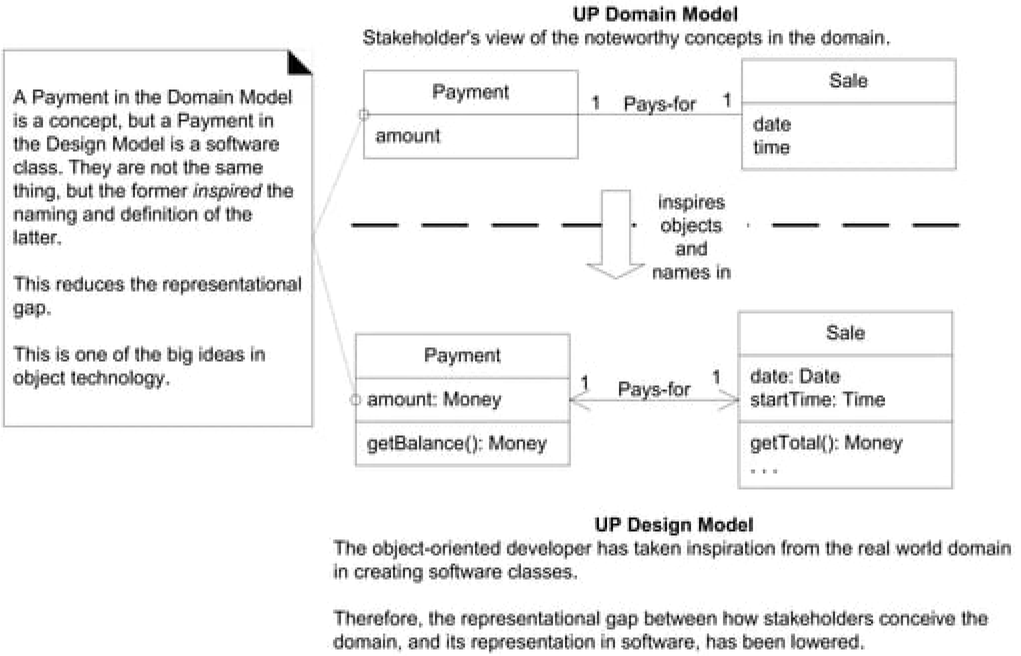
Software problems can be complex;

decomposition—divide-and-conquer—is a common strategy to deal with this complexity by division of the problem space into comprehensible units.

In **structured analysis, the dimension of decomposition** is by processes or *functions*.

However, *in* ***object-oriented analysis, the*** **dimension of decomposition is fundamentally by things or entities in the domain.**

Domain Model 5



A Payment in the Domain Model is a concept, but a Payment in the Design Model is a software class. They are not the same thing, but the former *inspired* the naming and definition of the latter.

This reduces the representational gap.

This is one of the big ideas in object technology.

UP Design Model

The object-oriented developer has taken inspiration from the real world domain in creating software classes.

Therefore, the representational gap between how stakeholders conceive the domain, and its representation in software, has been lowered.

- **Lower representational gap with OO modeling.**

Guideline: Create a Domain Model

- **Bounded by the current iteration requirements under design**

1. Find the conceptual classes.
2. Draw them as classes in a UML class diagram.
3. Add the association

   To record relationships for which there is a need to preserve some memory.

4. Add the attributes

   To fulfill the information requirements.

Aggregation and Composition

Aggregation

- **Definition :**
  - A special form of association that specifies a whole-part relationship between the aggregate (whole) and a component part.

- The whole is called the composite.

- Aggregation is also called "Has-a" relationship

- When an object 'has-a' another object,
  - Then you have got an aggregation between them.

Example:

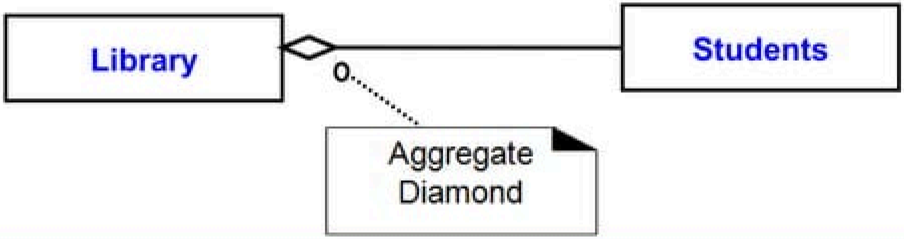
A Library contains students and books.

Relationship b/w library and student is aggregation.

Aggregation

- Aggregation is shown in UML with a hollow diamond symbol,
  - At the composite end of a whole-part association

- **Example:**
  - A Library contains students and books.
  - **Relationship b/w library and student is aggregation.**



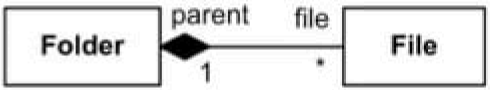
Association name is often excluded in aggregation relationships

since it is typically thought of as Has-part.

However, one may be used to provide more semantic detail.

Composite Aggregation (Composition)

- Composition is a special case of aggregation.
- Definition :
  - **When an object contains the other object, and if the contained object cannot exist without the existence of container object, then it is called composition.**



Folder could contain many files,

while each File has exactly one Folder parent.

If Folder is deleted, all contained Files are deleted as well.

Example:

A Library contains students and books.

Relationship between library and book is composition.

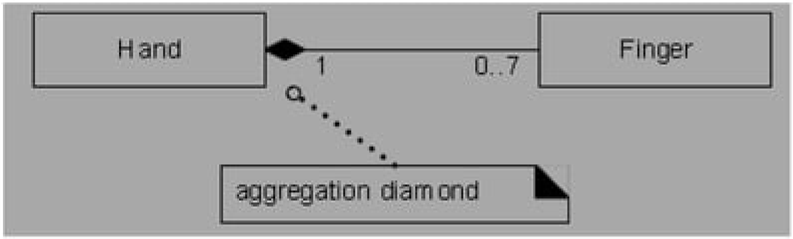
A student can exist without a library and therefore it is aggregation.

A book cannot exist without a library and therefore its a composition.

Composition

- Aggregation is shown in UML with a filled diamond symbol,
  - At the composite end of a whole-part association

- For instance,
  - Physical assemblies are organized in aggregation relationships,
    such as a Hand aggregates Fingers.



Aggregation Vs Composition

- Aggregation is a loosely suggests whole-part relationships
- Composition is a strong kind of whole-part aggregation
- Identifying and illustrating composition is not important;
  - It is quite reasonable to exclude it from a domain model.

Composite Aggregation (Composition)

- A composition relationship implies that

  1. An instance of the part (such as a Square) belongs to only one composite instance (such as one Board) at a time,

  2. The part must always belong to a composite (no free-floating Fingers) and

  3. The composite is responsible for the creation and deletion of its parts either by itself creating/deleting the parts, or by collaborating with other objects.

  – Related to this constraint is that if the composite is destroyed, its parts must either be destroyed, or attached to another composite (no free-floating Fingers allowed)

Composite Aggregation (Composition)

- **For example,**
  - If a physical paper Monopoly game board is destroyed, we think of the squares as being destroyed as well (a conceptual perspective).
  - Likewise, if a software Board object is destroyed, its software Square objects are destroyed, in software perspective.

How to identify Composition

- In some cases,
  - Presence of composition is obviously in physical assemblies.

- Guidelines:

  1. **On composition : If in doubt, leave it out**

  2. **Consider showing composition when:**

     1. Lifetime of the part is bound within the lifetime of the composite

        There is a create-delete dependency of the part on the whole.

     2. There is an obvious whole-part physical or logical assembly.
     3. Some properties of the composite propagate to the parts, such as the location.
     4. Operations applied to the composite propagate to the parts, such as destruction, movement, recording.

A Benefit of Showing Composition

- Most benefits of composition is relate to the design rather than the analysis
  - That is why its exclusion from the domain model is not very significant.

Benefits

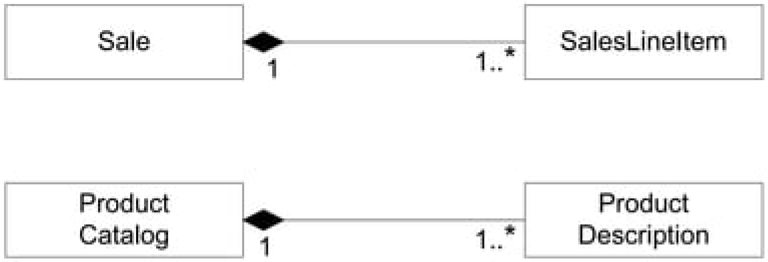
1. It clarifies the domain constraints regarding the eligible existence of the part independent of the whole.
   - During design work, this has an impact on the create-delete dependencies b/w the whole and part s/w classes and DB elements (in terms of referential integrity and cascading delete paths).
1. It assists in the identification of a creator (the composite)
   - Using the GRASP Creator pattern.
1. Operations such as copy & delete applied to whole often propagate to the parts.

Composition in NextGen Domain Model

- In the POS domain,
  - "SalesLineItems" may be considered a part of a complete "sale"
- In general,
  - Transaction line items are viewed as parts of an aggregate transaction.
- In addition to conformance to that pattern,
  - There is a create-delete dependency of the line items on the Sale

    (their lifetime is bound within the lifetime of the sale)

Aggregation in the POS Application



- By similar justification
  - “ProductCatalog” is a composite of “ProductDescriptions”

Associations

An association is a relationship between instances of types that indicates some meaningful and interesting connection

Associations

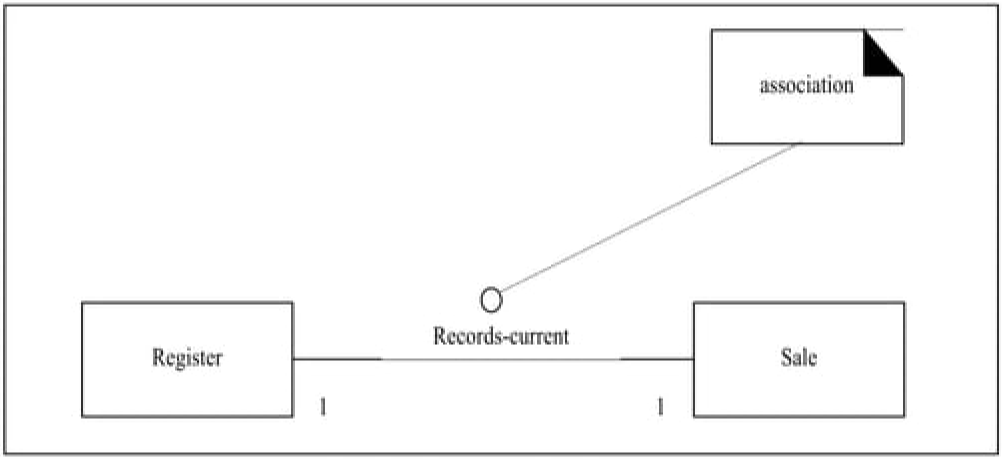


Fig 1.Associations

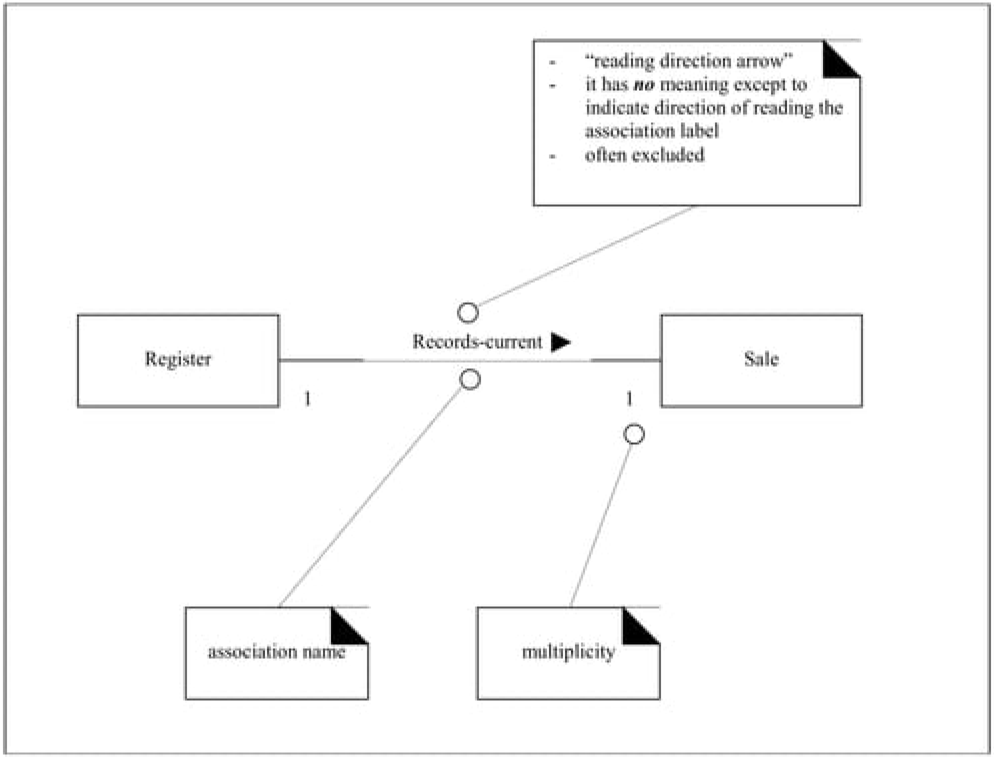
Useful Associations

- Associations for which knowledge of the relationship needs to be preserved for some duration.
- Associations derived from the Common Associations List.

UML Association Notation

- An association is represented as a line between classes with an association name.
- Associations are inherently bidirectional.
- Optional reading direction arrow is only an aid to the reader of the diagram.

UML Association Notation


- "reading direction arrow"
- it has *no* meaning except to indicate direction of reading the association label
- often excluded
Register
Records-current ▶
Sale
1
1
association name
multiplicity

Finding Associations- Common Associations List

The common categories that are worth considering are:

- A is a physical part of B . *Eg: Wing-Airplane*
- A is a logical part of B. *Eg: SalesLineItem-Sale.*
- A is physically contained in B . *Eg: Register-Store.*

Common Associations List 2

- A is logically contained in B. *Eg:ItemDescription-Catalog.*
- A is a description of B.*Eg:ItemDescription-Item.*
- A is a line item of a transaction or report B.*Eg:SalesLineItem-Sale*.
- A is a member of B .*Eg: Cashier-Store.*
- A uses or manages B.*Eg:Cashier-Register.*

Common Associations List 3

- A is known/logged/recorded/reported/captured in B.Eg: Sale-Register.
- A is an organizational subunit of B . *Eg:Department-Store.*
- A communicates with B. *Eg:Customer-Cashier.*
- A is next to B. *Eg:City-City.*

Common Associations List 4

- A is related to a transaction B. *Eg: Customer-Payment.*
- A is a transaction related to another transaction B. *Eg:Payment-Sale.*
- A is next to B. *Eg:City-City*.
- A is owned by B. *Eg:Register-Store.*
- A is an event related to B. *Eg:Sale-Customer.*

High-Priority Associations

- A is a physical or logical part of B.
- A is physically or logically contained in/on B.
- A is recorded in B.

Associations Guidelines

- The knowledge of the relationship needs to be preserved for some duration.
- Identifying conceptual classes is more important than identifying associations.
- Avoid showing redundant or derivable associations.

Roles

- Each end of an association is called a role.
- Roles may have
  - *name*
  - *multiplicity expression*
  - *Navigability(know about each other)*

Multiplicity

- Multiplicity defines the number of instances of a class A ,that can be associated with one instance of class B,at a particular moment.
- Eg: In countries with monogamy laws,a person can be married to 1 person at any particular time.But over a span of time one person can be married to many persons.

Multiplicity

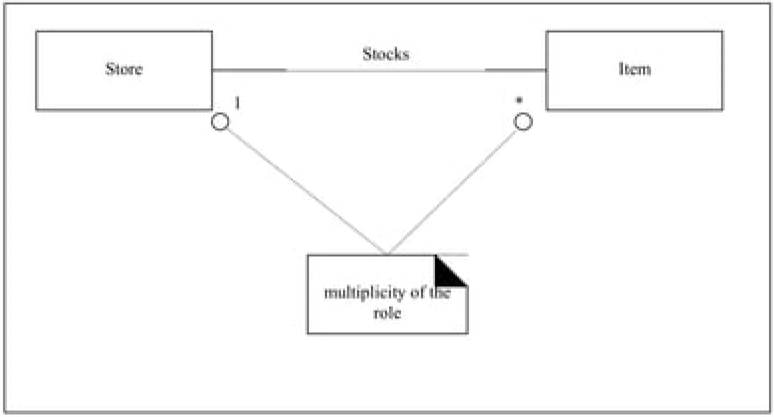


Fig 3 Multiplicity on an association.

Multiplicity

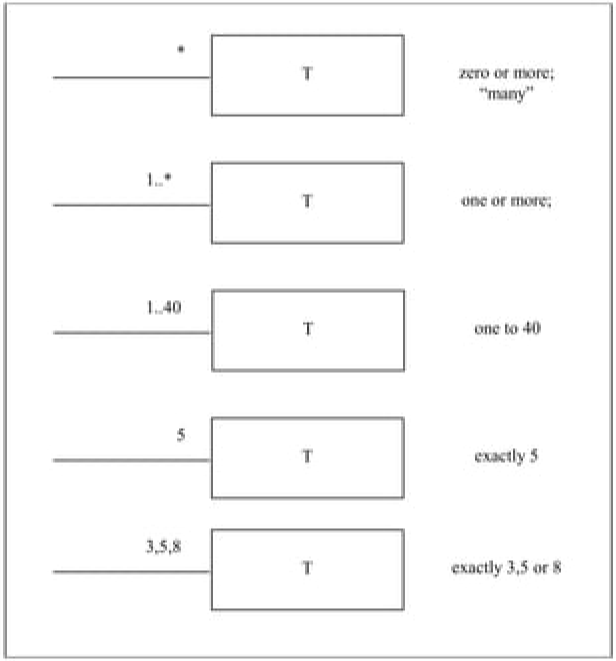


Fig 4. Multiplicity values.

Naming Associations

- Name an association based on TypeName-VerbPhrase-TypeName format.
- Names should start with a capital letter.
- Legal formats are:
  - Paid-by
  - PaidBy

Associations Names

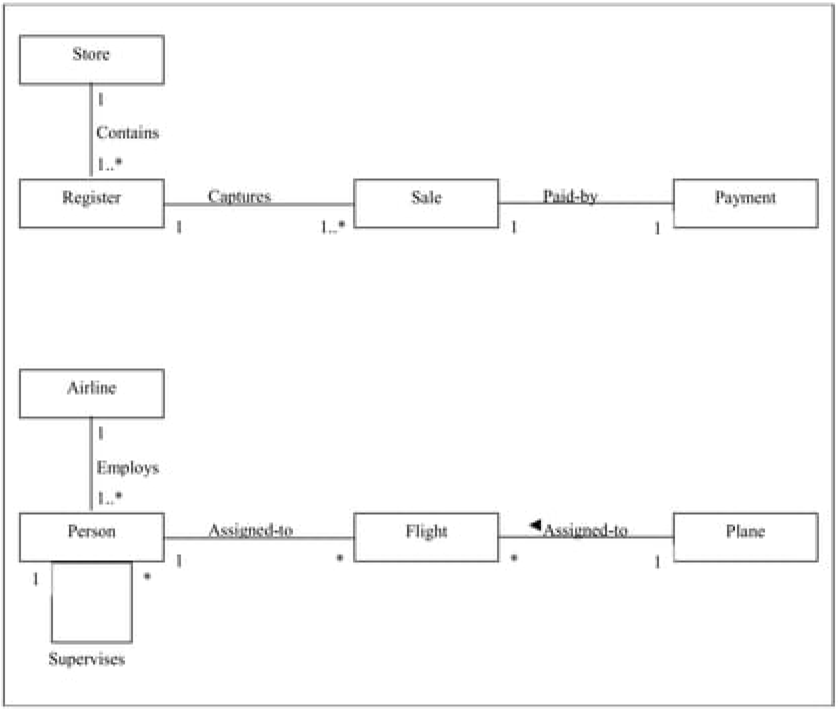


Fig 5.Association names.

Multiple Associations

- Two types may have multiple associations between them.

Multiple Associations

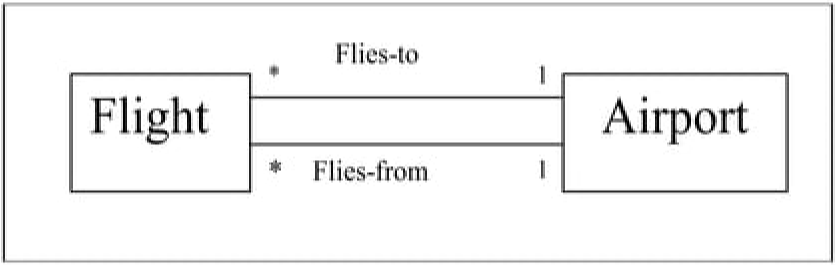


Fig 6. Multiple associations.

Implementation

- The domain model can be updated to reflect the newly discovered associations.
- But,avoid updating any documentation or model unless there is a concrete justification for future use.
- Defer design considerations so that extraneous information is not included and maximizing design options later on.

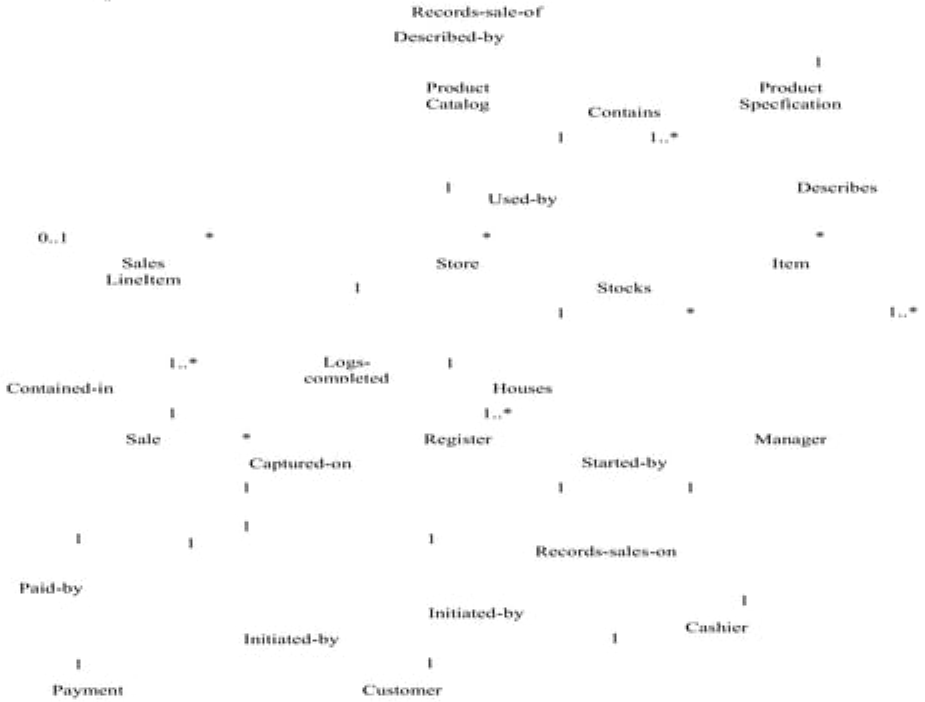
Cleaning Associations 1

- Do not overwhelm the domain model with associations that are not strongly required.
- Use need-to-know criterion for maintaining associations.
- Deleting associations that are not strictly demanded on a need-to-know basis can create a model that misses the point.

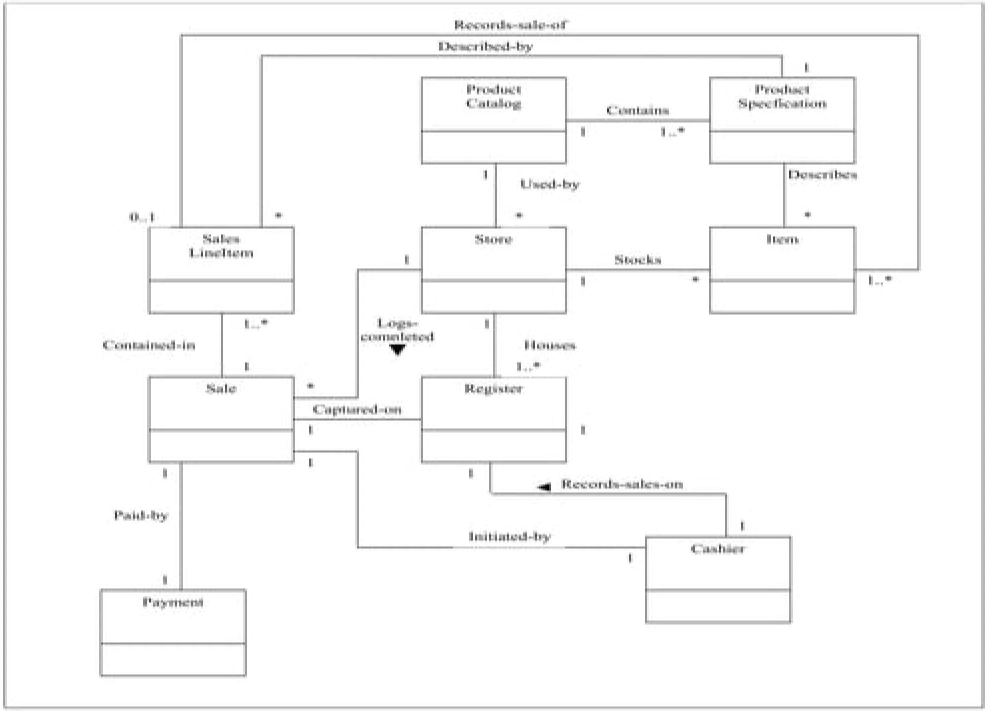
Cleaning Associations 2

- Add comprehension-only associations to enrich critical understanding of the domain.

A partial domain model



Without need-to-know associations



Conclusion

- When in doubt if the concept is required,keep the concept.
- When in doubt if the the association is required,drop it.
- Do not keep derivable association.