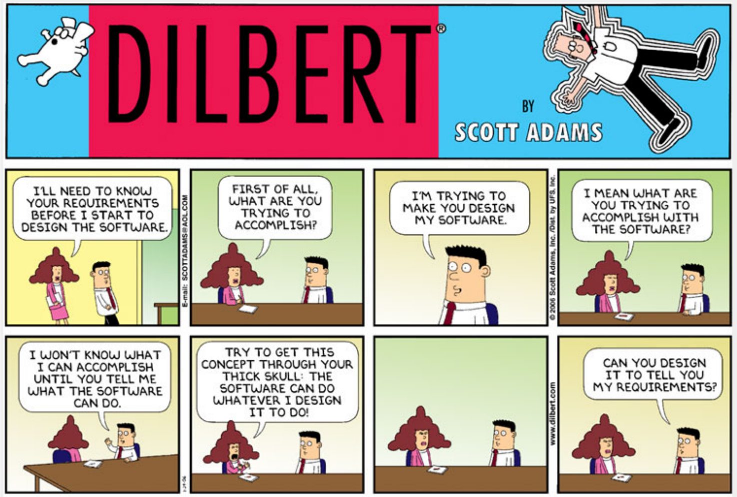
Project Iteration Artifacts

CSE3063

Majority of slides are taken from CSE 432: Object-Oriented Software Engineering class at Lehigh University (http://www.cse.lehigh.edu/~glennb/oose/oose.htm)


DILBERT
BY SCOTT ADAMS
I'LL NEED TO KNOW YOUR REQUIREMENTS BEFORE I START TO DESIGN THE SOFTWARE.
FIRST OF ALL, WHAT ARE YOU TRYING TO ACCOMPLISH?
I'M TRYING TO MAKE YOU DESIGN MY SOFTWARE.
I MEAN WHAT ARE YOU TRYING TO ACCOMPLISH WITH THE SOFTWARE?
I WON'T KNOW WHAT I CAN ACCOMPLISH UNTIL YOU TELL ME WHAT THE SOFTWARE CAN DO.
TRY TO GET THIS CONCEPT THROUGH YOUR THICK SKULL: THE SOFTWARE CAN DO WHATEVER I DESIGN IT TO DO!
CAN YOU DESIGN IT TO TELL YOU MY REQUIREMENTS?
E-mail: SCOTTADAMS@AOL.COM
© 2006 Scott Adams, Inc./Dist. by UFS, Inc.
www.dilbert.com

© Scott Adams, Inc./Dist. by UFS, Inc.

1- 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*

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,
  - remember

  what is really needed, in the form that clearly speaks to the

  1. client (customer)
  2. development team members

Functional (*what* behaviors it does) and non-functional (*how* it does them)

- **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)

Desiderata for a requirements specification

- Should say ***what***, not *how.* **Why?**
- **Correct**: does what the client wants, according to specification
  - Like motherhood and apple pie—how to accomplish it?
  - 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
  - Tension between understandability and other desiderata?
- **Modifiable**: requirements change!
  - Changes should be noted and agreed upon, in the spec!

2- Use cases

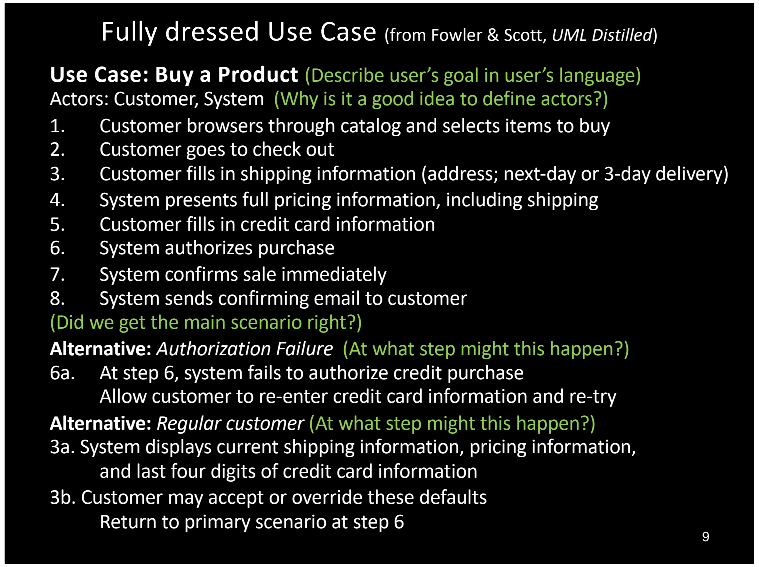
- First developed by Ivar Jacobson
  - Now part of the UML (though not necessarily object-oriented)
  - Emphasizes user’s point of view
  - **Explains everything in the user’s language**
- A "use *case*" is a set of cases or scenarios for using a system, tied together by a common user goal
  - Essentially 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 = the set of all use cases
- Use cases are good for brainstorming the requirements

Brief Use Case format

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

Rent Videos.

A Customer browses and selects videos to rent from the web site. System outputs the price information for each video ID. Customer fills in credit card information. System confirms sale and send confirmation message through email.



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?)

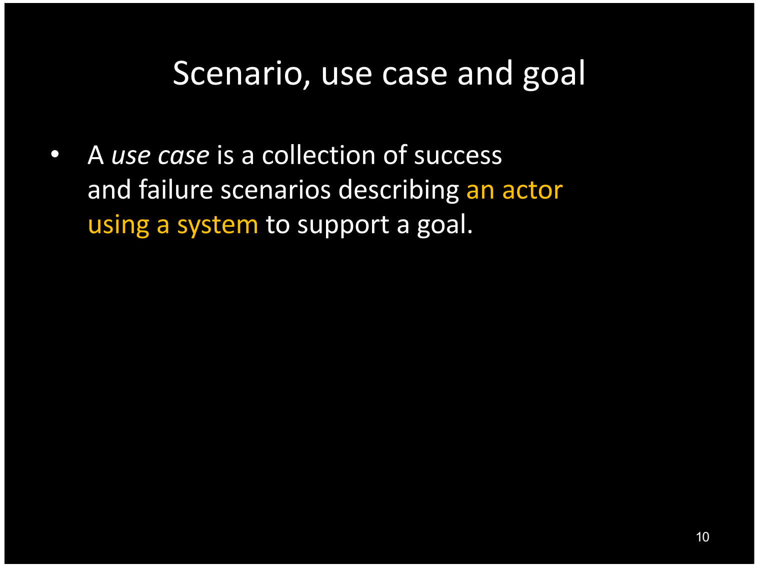
1. Customer browsers through catalog and selects items to buy
2. Customer goes to check out
3. Customer fills in shipping information (address; next-day or 3-day delivery)
4. System presents full pricing information, including shipping
5. Customer fills in credit card information
6. System authorizes 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

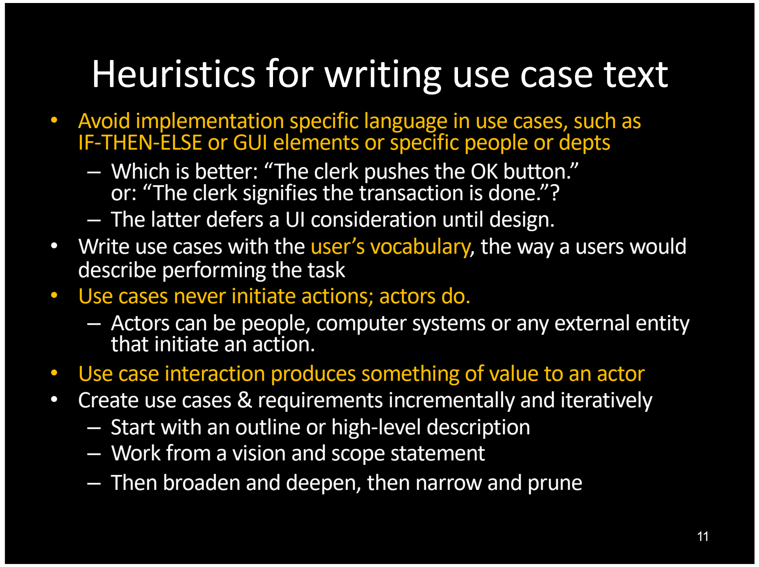
9



Scenario, use case and goal

- A *use case* is a collection of success and failure scenarios describing an actor using a system to support a goal.

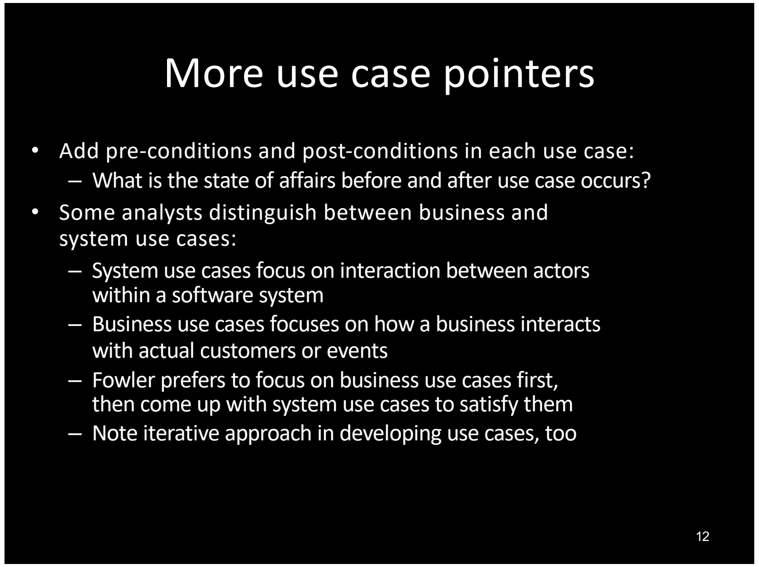
10



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

11



More use case pointers

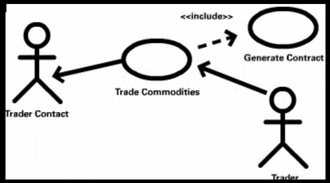
- Add pre-conditions and post-conditions in each use case:
  - What is the state of affairs before and after use case occurs?
- Some analysts distinguish between business and system use cases:
  - System use cases focus on interaction between actors within a software system
  - Business use cases focuses on how a business interacts with actual customers or events
  - Fowler prefers to focus on business use cases first, then come up with system use cases to satisfy them
  - Note iterative approach in developing use cases, too

12

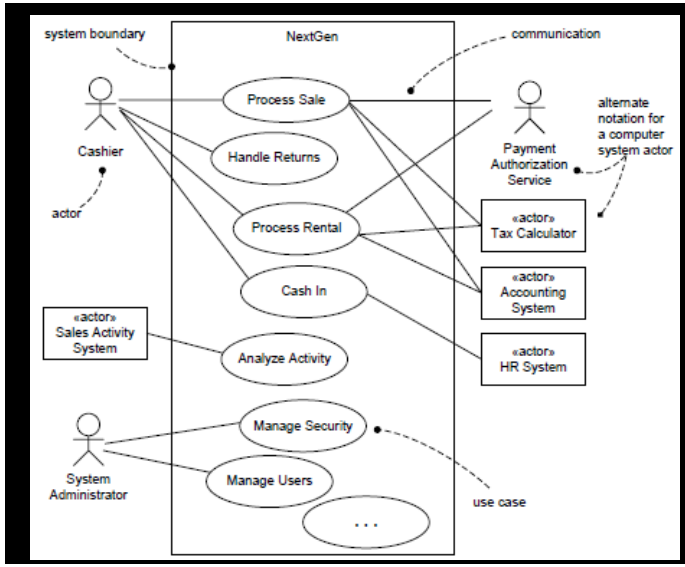
Text and Diagrams

- **Use cases are *text*, not diagrams**
- Use case *text* provides the detailed description of a particular use case
- UML Use case *diagram* provides an overview of interactions between actors and use cases

Use case diagram

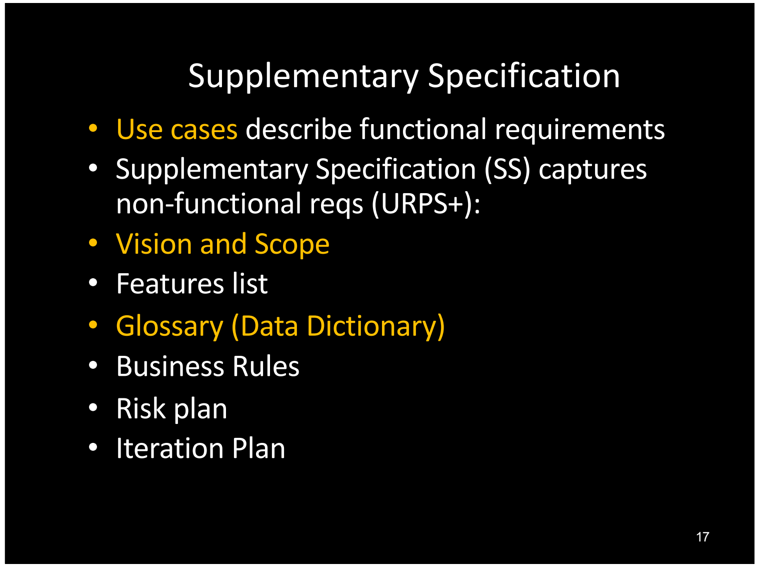


- Bird's eye view of use cases for a system
- Stick figures represent **actors** (human or computer in **roles**)
- Ellipses are **use cases** (behavior or functionality seen by users)
- What can user do with the system?
  - E.g., Trader interacts with Trader Contract via a Trade Commodities transaction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



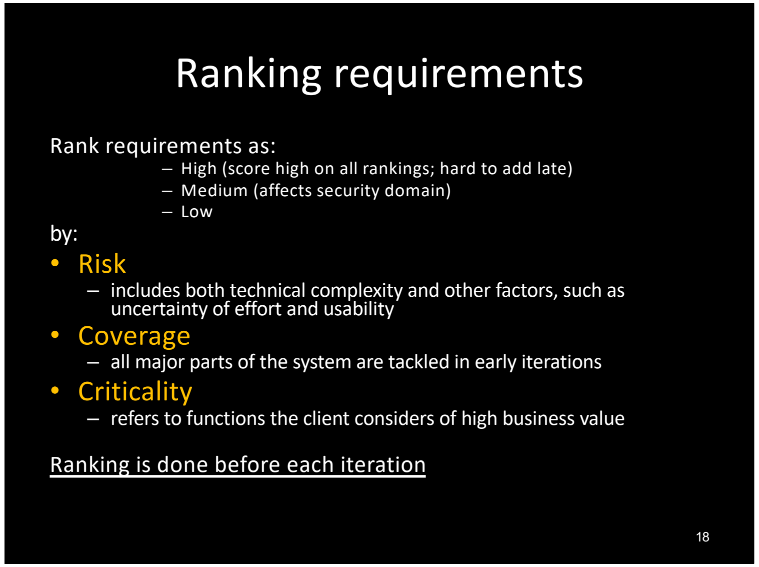
Advantages of use cases

- 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 Business Rules
  - When should we bother with diagrams?
- Use cases can drive the whole development process:
  - Analysis understand what user wants with use cases
  - Design and implementation realizes them
  - Help with early design of UI prototype
  - Help set up test plans
  - Help with writing a user manual



Supplementary Specification

- Use cases describe functional requirements
- Supplementary Specification (SS) captures non-functional reqs (URPS+):
- Vision and Scope
- Features list
- Glossary (Data Dictionary)
- Business Rules
- Risk plan
- Iteration Plan



Ranking requirements

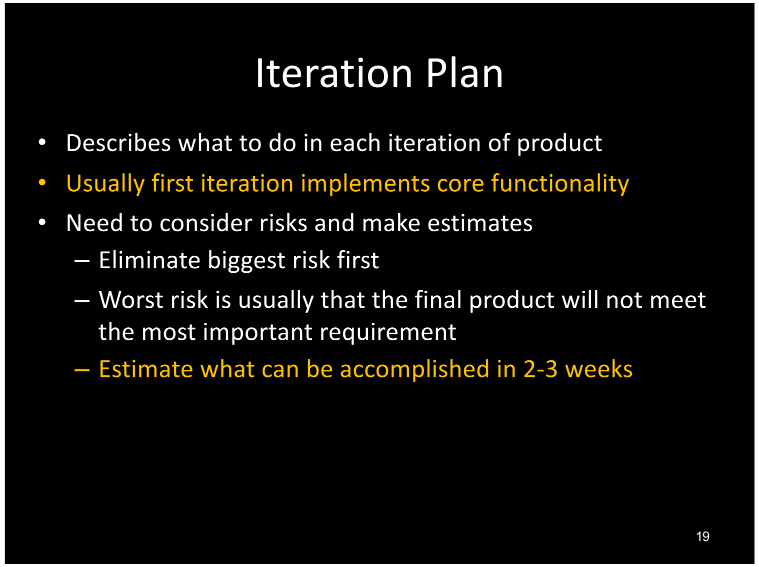
Rank requirements as:

- High (score high on all rankings; hard to add late)
- Medium (affects security domain)
- Low

by:

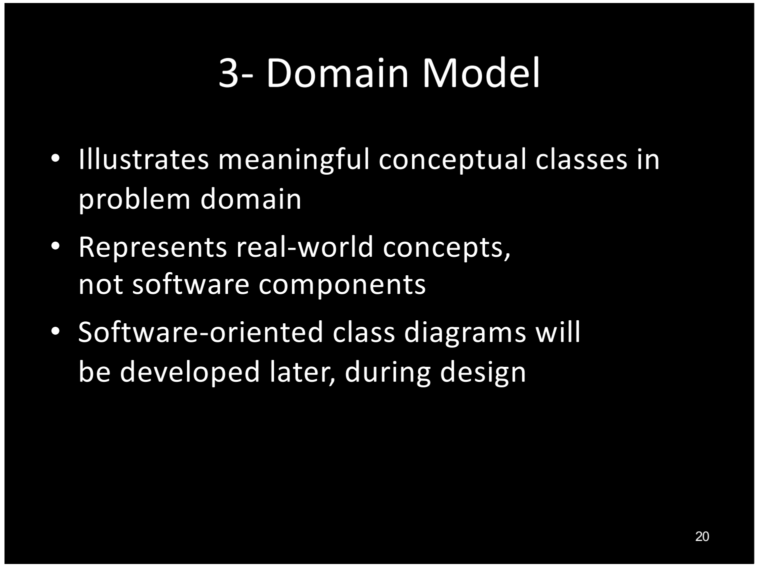
- Risk
  - includes both technical complexity and other factors, such as uncertainty of effort and usability
- Coverage
  - all major parts of the system are tackled in early iterations
- Criticality
  - refers to functions the client considers of high business value

Ranking is done before each iteration



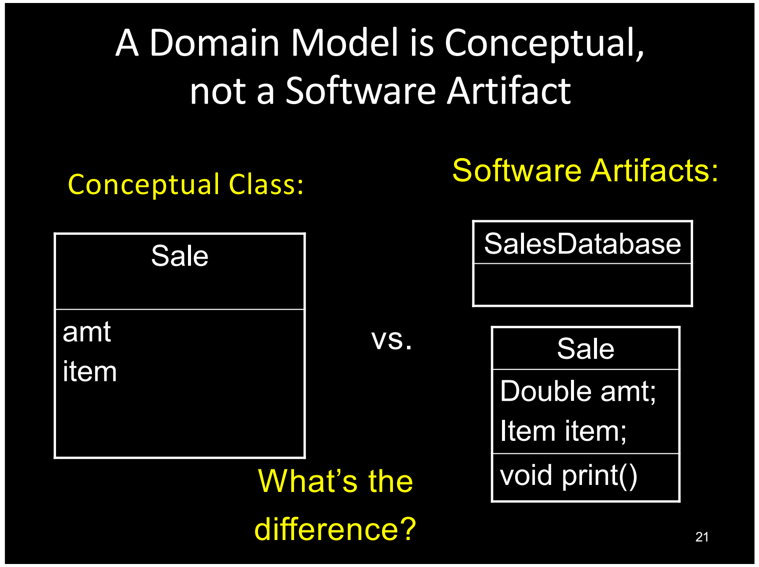
Iteration Plan

- Describes what to do in each iteration of product
- Usually first iteration implements core functionality
- Need to consider risks and make estimates
  - Eliminate biggest risk first
  - Worst risk is usually that the final product will not meet the most important requirement
  - Estimate what can be accomplished in 2-3 weeks



3- Domain Model

- Illustrates meaningful conceptual classes in problem domain
- Represents real-world concepts, not software components
- Software-oriented class diagrams will be developed later, during design



A Domain Model is Conceptual, not a Software Artifact

Conceptual Class:

Sale

amt
item

vs.

Software Artifacts:

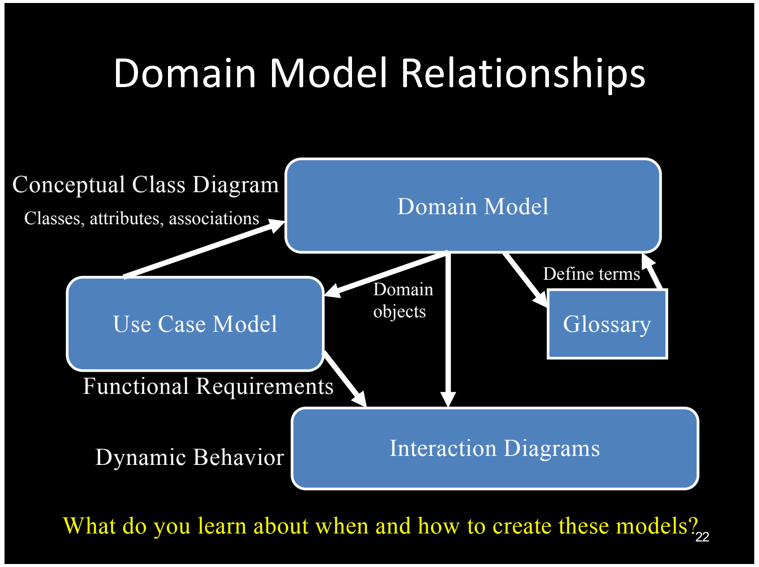
SalesDatabase

Sale

Double amt;
Item item;

void print()

What's the difference?



Domain Model Relationships

Conceptual Class Diagram

Classes, attributes, associations

Domain Model

Use Case Model

Domain objects

Define terms

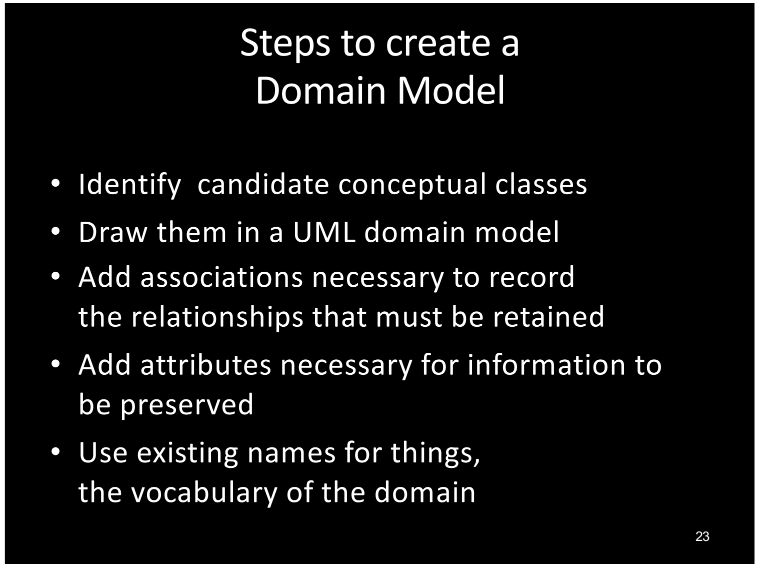
Glossary

Functional Requirements

Dynamic Behavior

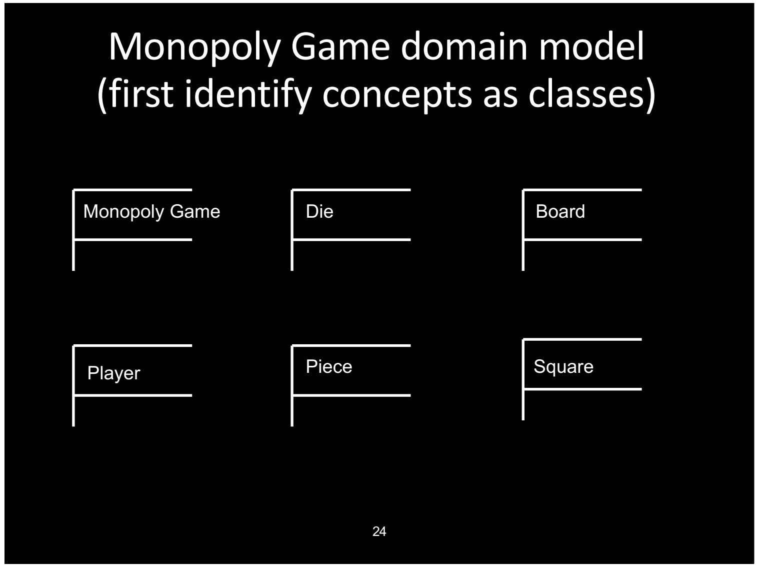
Interaction Diagrams

What do you learn about when and how to create these models?



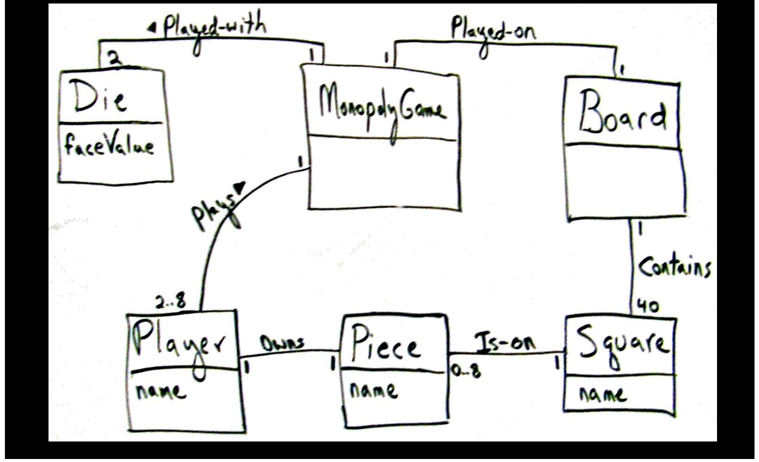
Steps to create a Domain Model

- Identify candidate conceptual classes
- Draw them in a UML domain model
- Add associations necessary to record the relationships that must be retained
- Add attributes necessary for information to be preserved
- Use existing names for things, the vocabulary of the domain



Monopoly Game domain model (first identify concepts as classes)

Monopoly Game domain model
Larman, Figure 9.28



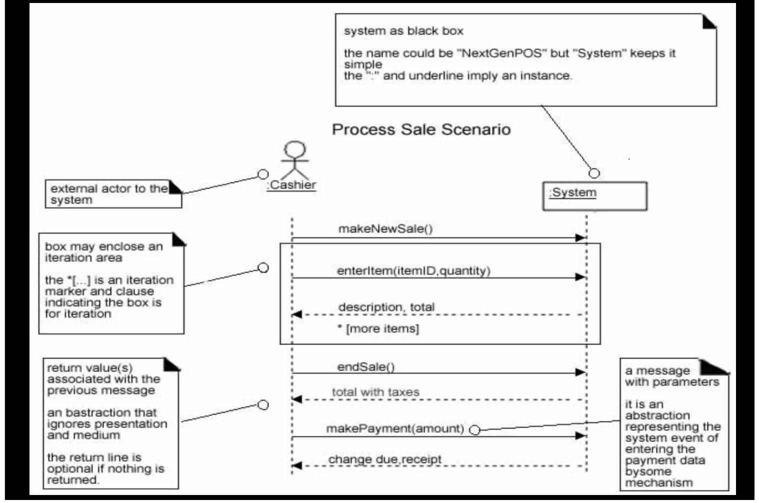
4- System Sequence Diagram (SSD)

For a use case scenario, an SSD shows:

- The System (as a black box) :System
- The external actors that interact with System
- The System events that the actors generate
- SSD shows operations of the System in response to events, in temporal order
- Develop SSDs for the main success scenario of a selected use case, then frequent and salient alternative scenarios

26

SSD for Process Sale scenario
(Larman, page 175)



system as black box

the name could be "NextGenPOS" but "System" keeps it simple
the ":" and underline imply an instance.

external actor to the system

box may enclose an iteration area

the *[...] is an iteration marker and clause indicating the box is for iteration

return value(s) associated with the previous message

an bastraction that ignores presentation and medium

the return line is optional if nothing is returned.

a message with parameters

it is an abstraction representing the system event of entering the payment data bysome mechanism

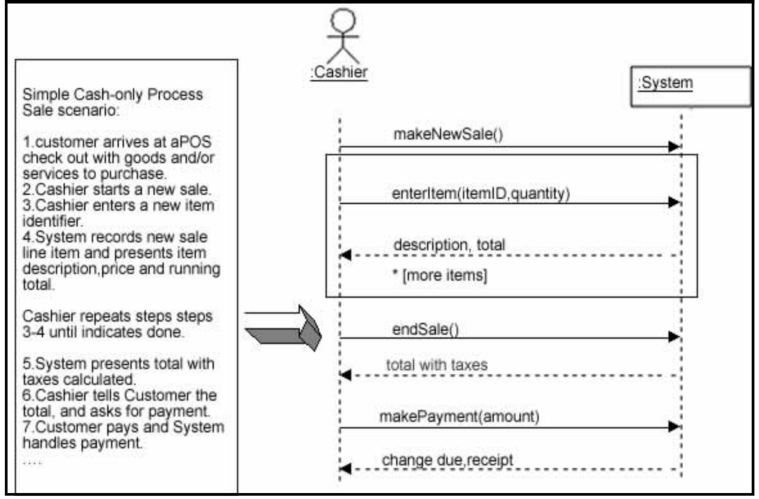
From Use Case to Sequence System Diagram

How to construct an SSD from a use case:

1. Draw System as black box on right side
2. For each actor that directly operates on the System, draw a stick figure and a lifeline.
3. For each System events that each actor generates in use case, draw a message.
4. Optionally, include use case text to left of diagram.

28

Example: use cases to SSD



Simple Cash-only Process Sale scenario:

1.customer arrives at aPOS check out with goods and/or services to purchase.
2.Cashier starts a new sale.
3.Cashier enters a new item identifier.
4.System records new sale line item and presents item description,price and running total.

Cashier repeats steps steps 3-4 until indicates done.

5.System presents total with taxes calculated.
6.Cashier tells Customer the total, and asks for payment.
7.Customer pays and System handles payment.
....

Identifying the right Actor

- In the process Sale example, does the customer interact directly with the POS system?
- Who does?
- Cashier interacts with the system directly
- Cashier is the generator of the system events
- Why is this an important observation?

30

Naming System events & operations

- System events and associated system operations should be expressed at the level of intent
- Rather than physical input medium or UI widget
- Start operation names with verb (from use case)
- Which is better, scanBarCode or enterItem?

31

SSDs and the Glossary in parallel

- Why is updating the glossary important when developing the SSD?
- New terms used in SSDs may need explanation, especially if they are not derived from use cases
- A glossary is less formal, easier to maintain and more intuitive to discuss with external parties such as customers

32

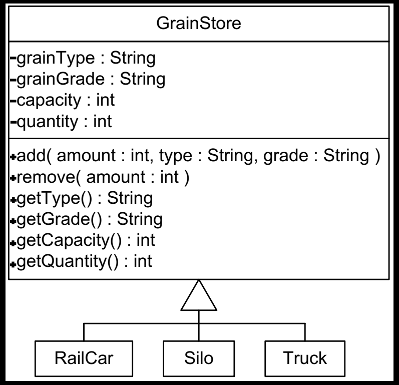
SSDs within the Unified Process

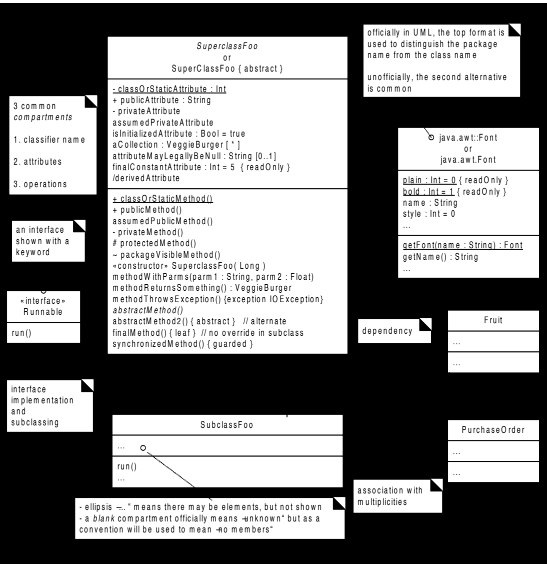
Create System Sequence Diagrams during Elaboration in order to:

- Identify System events and major operations
- Write System operation contracts (Contracts describe detailed system behavior)
- Support better estimates
- Remember, there is a season for everything: it is not necessary to create SSDs for all scenarios of all use cases, at least not at the same time

- ANALYSIS phase is finished for the iteration.
- Now let’s start DESIGN phase!

5- UML Class Diagram




SuperclassFoo
or
SuperClassFoo { abstract }
- classOrStaticAttribute : Int
+ publicAttribute : String
- privateAttribute
assumedPrivateAttribute
isInitializedAttribute : Bool = true
aCollection : VeggieBurger [*]
attributeMayLegallyBeNull : String [0..1]
finalConstantAttribute : Int = 5 { readOnly }
/derivedAttribute
+ classOrStaticMethod()
+ publicMethod()
assumedPublicMethod()
- privateMethod()
# protectedMethod()
~ packageVisibleMethod()
«constructor» SuperclassFoo(Long)
methodWithParms(parm1 : String, parm2 : Float)
methodReturnsSomething() : VeggieBurger
methodThrowsException() {exception IOException}
abstractMethod()
abstractMethod2() { abstract } // alternate
finalMethod() { leaf } // no override in subclass
synchronizedMethod() { guarded }
«interface»
Runnable
run()
SubclassFoo
...
run()
...
java.awt::Font
or
java.awt.Font
plain : Int = 0 { readOnly }
bold : Int = 1 { readOnly }
name : String
style : Int = 0
...
getFont(name : String) : Font
getName() : String
...
Fruit
...
...
PurchaseOrder
...
...

3 common *compartments*

1. classifier name

2. attributes

3. operations

an interface shown with a keyword

interface implementation and subclassing

officially in UML, the top format is used to distinguish the package name from the class name

unofficially, the second alternative is common

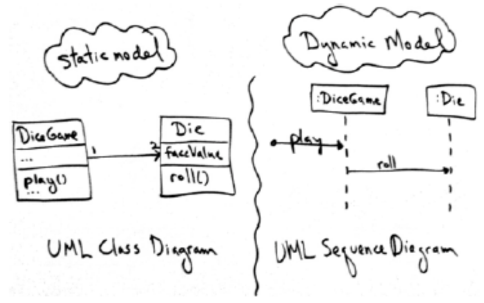
dependency

association with multiplicities

- ellipsis –... “ means there may be elements, but not shown
- a *blank* compartment officially means –unknown“ but as a convention will be used to mean –no members“

6- UML Sequence Diagram

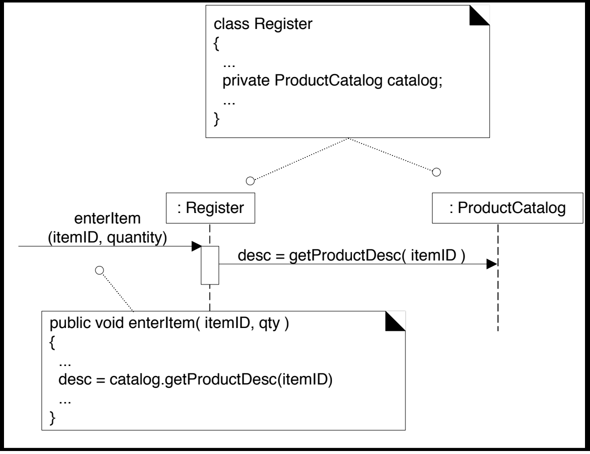
Figure 14.1. Static and dynamic UML diagrams for object modeling.



BONUS: Visibility

- Visibility is the ability of one object to “see” or have reference to another
- To send a message from one object to another, the receiver object must be “visible” to the sender, via a reference

What does the getProductDesc message imply about object visibility?



```
class Register
{
  ...
  private ProductCatalog catalog;
  ...
}
```

```
public void enterItem( itemID, qty )
{
  ...
  desc = catalog.getProductDesc(itemID)
  ...
}
```

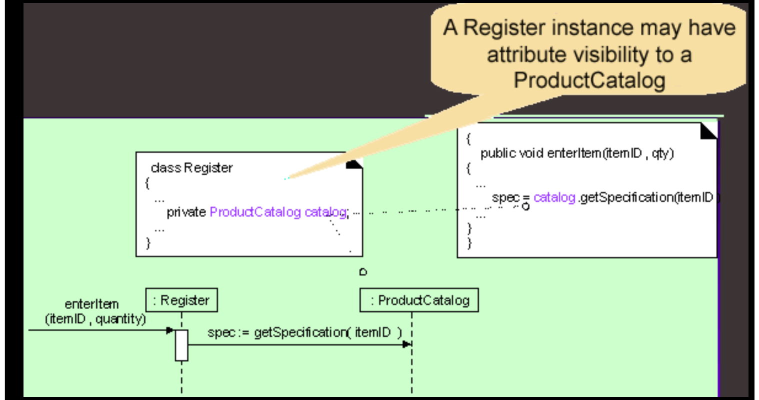
Fig. 19.1

Four Kinds of Visibility

OO programming languages may provide four levels of scope for names:

- Attribute visibility *(most common in OO systems)*
- Parameter visibility
- Local visibility
- Global visibility

Attribute Visibility



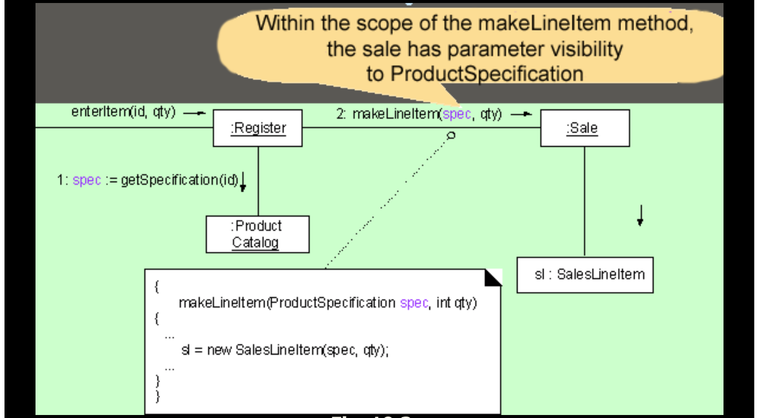
A Register instance may have attribute visibility to a ProductCatalog

```
class Register
{
 ...
  private ProductCatalog catalog;
 ...
}
```

```
{
  public void enterItem(itemID , qty)
{
 ...
   spec = catalog.getSpecification(itemID)
 ...
}
}
```

Fig. 19.2

Parameter Visibility



Within the scope of the makeLineItem method, the sale has parameter visibility to ProductSpecification

```
{
   makeLineItem(ProductSpecification spec, int qty)
{
 ...
   sl = new SalesLineItem(spec, qty);
 ...
}
}
```

Fig. 19.3

- Why is transforming parameters to attribute visibility common in OO design?

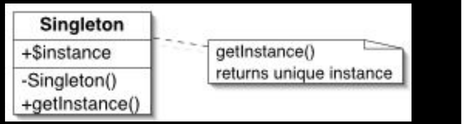
Global Visibility

- Object B has global scope relative to A
  - Relatively permanent visibility
  - Least common visibility in OO design
- Ways to achieve global visibility:
  - Assign an instance to a global variable.
  - Use the Singleton pattern

Singleton design pattern

Gamma, Helm, Johnson, and Vlissides (aka Gang of Four)

- Ensure that a class has only one instance and provide a global point of access to it
  - *Why not use a global variable?*



```
class Singleton
{ public:
     static Singleton* getInstance(); //accessor
  protected: //Why are the following protected?
     Singleton();
     Singleton(const Singleton&);
     Singleton& operator= (const Singleton&);
 private: static Singleton* instance; //unique
};
Singleton *p2 = p1->getInstance();
```