

USC - Center for Software Engineering

Unified Modeling Language

Based in parts on 'UML Distilled' from Martin Fowler

1

Why UML — OO Notation Standard Needed

- OO methodology evolved in parallel in different areas of development

Hardware	Programming Languages
Hand axe etc.	Simula 67/SmallTalk C++ Object-Oriented Languages
Design Methods	Database
"Program Design by Informal English Descriptions", Abbott Booch Ken Orr ERDs Chen ERDs	Flat-File Databases Hierarchical Databases Relational Databases Object-Oriented Databases

Many notations evolved

2

UML Goals

- Provide a visual modeling language that is
 - Ready to use
 - Expressive
 - Independent of particular programming languages
 - Only partially succeeded
 - Independent of development process
 - Extensible
 - Core concepts can be extended or specialized by users
 - Supportive of higher-level concepts
 - e.g. collaborations, frameworks, patterns, components
 - Provide a modeling language that
 - Integrates best practices
 - Encourages growth of OO tools market

3

UML History

- When UML started, architecture research still in infancy

4

UML Characteristics

- Large, useful set of predefined constructs
- Extensible
- Semi-formal definition of syntax & semantics
- Wide adoption
- Standardization
- Substantial tool support
- Basis in experience with mainstream development methods

Overview of UML

- UML model comprises several views & models addressing
 - Classes with attributes, operations, & relationships
 - States & behavior of individual classes
 - Packages of classes & their dependencies
 - Example scenarios of system usage
 - Object instances in scenario
 - Actual behavior of interacting instances in scenario
 - Distributed component deployment & communication
- UML syntax & semantics are defined via
 - metamodel
 - A UML model that specifies abstract syntax & semantics of language
 - Descriptive text
 - Constraints

6



Outline

- **Diagrams**
 - Use Case Diagrams
 - Class Diagrams
 - Interaction Diagrams
 - Package Diagrams
 - State Diagrams
 - Activity Diagrams
 - Deployment Diagrams
- **Properties (Pros/Cons)**

7



UML OMG Standard

UML is a language for specifying, constructing, and documenting software systems:

- General Purpose Modeling Language
- Merges Modeling Element from Booch, Rumbaugh, Jacobson and others
- Object-Oriented Analysis and Design
- Graphical Notation supporting numerous diagrams
- Extensible Notation (Stereotypes)
- Extensible Semantics (Object Constraint Language)

UML is backed by numerous software producers such as Digital, HP, IBM, Oracle, Microsoft, Unisys, and others

USC CSE Analysis, Design, Programming
 USC - Center for Software Engineering

Object-Oriented Analysis OOA
 -> create a domain model in a OO modeling notation
 -> problem/environment description

Object-Oriented Design OOD
 -> create a product model in a OO modeling notation
 -> solution/interface to environment description

Object-Oriented Programming OOP
 -> OOD implementation in a programming language
 -> filling in the 'blanks' not defined in the OOD

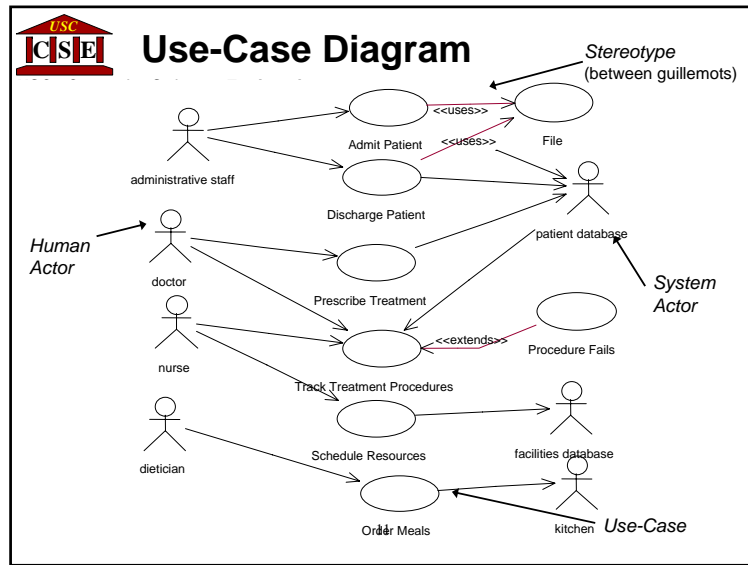
9

USC CSE Use-Case Diagrams
 USC - Center for Software Engineering

- Capture those parts of the system that are visible to the outside (e.g. users or other systems).
- A use case is usually about a concrete goal or task from the user's point of view
- Use-Cases give no feedback in terms of complexity or system decomposition

=> mainly an analysis method.

10



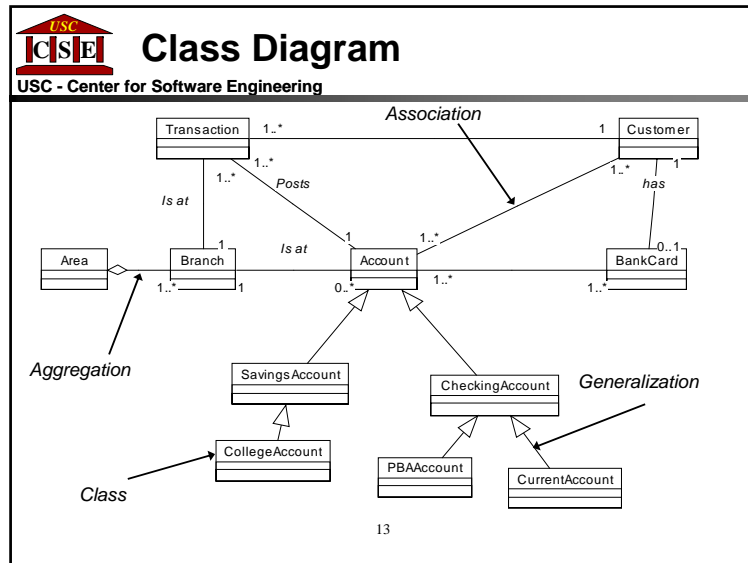
USC CSE Class Diagrams
 USC - Center for Software Engineering

- Describe the static relationship of classes in a system. These relationships must always be true.
- Classes often represent physical or otherwise tangible entities but this must not always be true.
- Classes are probably the most misunderstood and misused elements in OO design.

Connectors => Message Passing/Method Invocation:

- Associations
- Aggregation
- Generalization

12



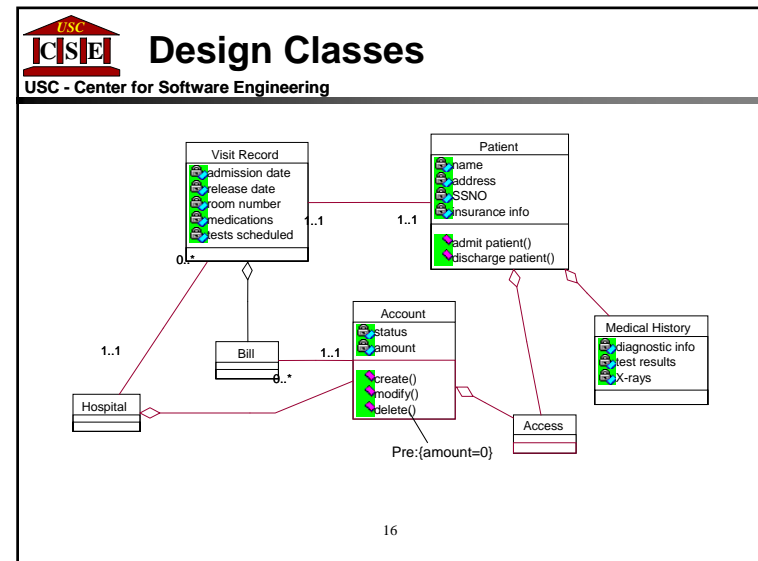
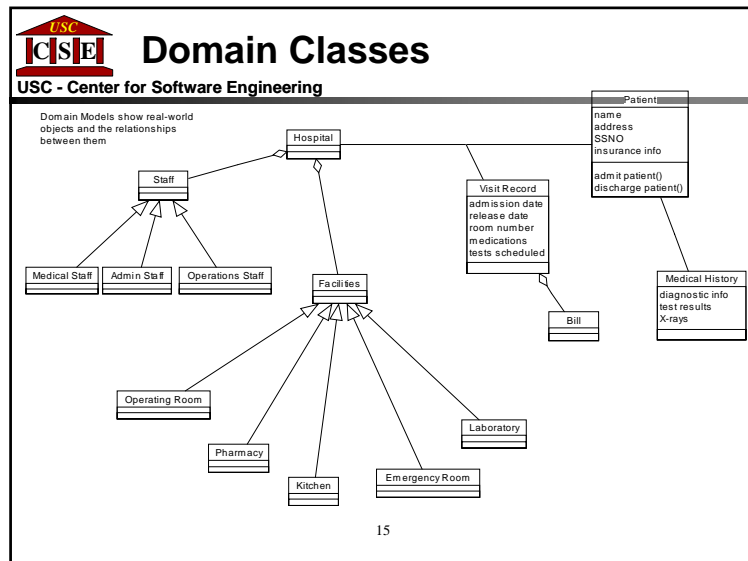
Class Diagrams

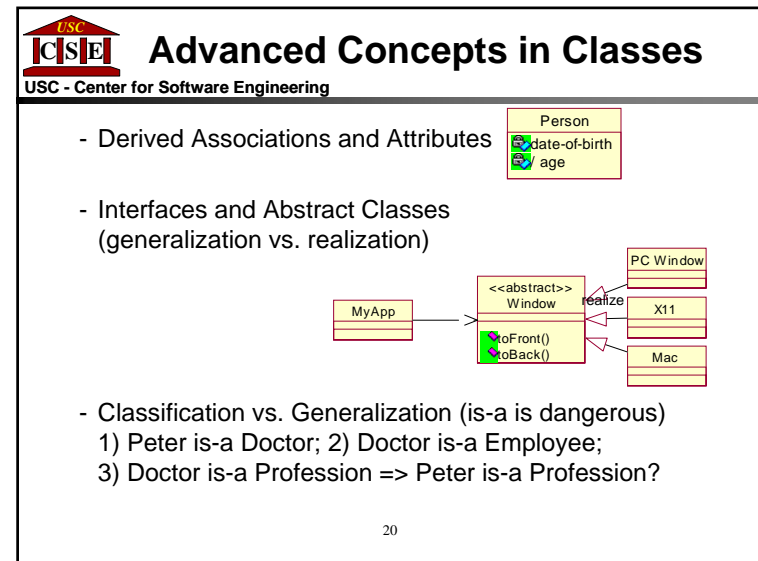
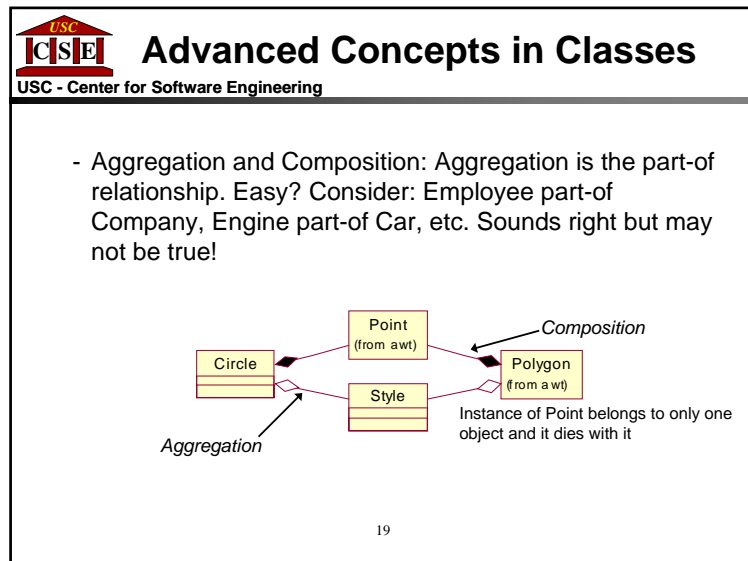
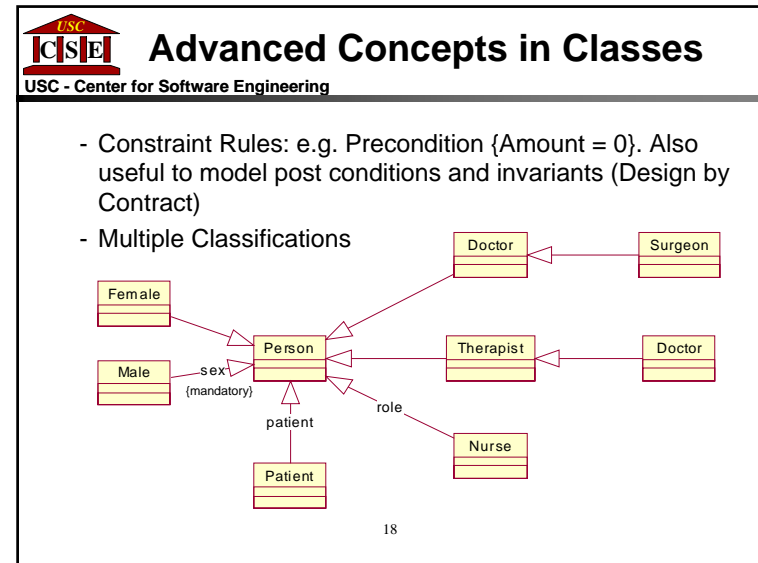
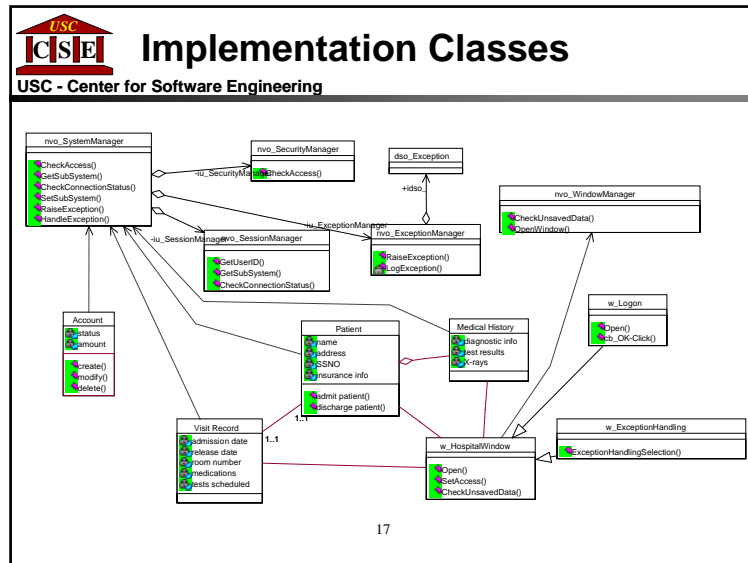
Classes are usually the most *central* OO development elements since they reflect the actual implementation the closest -> this may have negative side effects.

Need to distinguish three types of classes:

- Conceptual Classes: Should capture domain entities with no regard to the actual implementation.
- Design Classes: Should capture design entities without overly committing to implementation details
- Implementation Classes: Should capture the physical model (e.g. strongly influenced by programming language)

14





USC CSE Advanced Concepts in Classes
 USC - Center for Software Engineering

- Qualified Associations
 - Order uses Product to identify Order Line
 - Diagram: Order (0..*) -- line item --> Order Line (0..1) with Product as a qualified association.
- Association Classes
 - Diagram: Person (0..*) -- employer --> Company (0..1) with an association class Employment (period).
 - Diagram: Person (0..*) --> Skills (0..*) with an association class Competency.
 - Text: "Is categorized by" points to the Order Line class.
 - Text: "Implies only one instance of Employment per Association of Person and Company" points to the Employment class.
 - Text: "=> Person employed by same company at different times?"

21

USC CSE Advanced Concepts in Classes
 USC - Center for Software Engineering

- Parameterized Class
 - Diagram: Set class with methods insert() and remove().
 - Diagram: EmployeeSet class with a parameterized association to Employee (EmployeeSet --> Employee) and a binding <<bind>>.
 - Diagram: EmployeeSet class with a parameterized association to Set (EmployeeSet --> Set).
- Instantiated Class, Utility Class, Meta-Class, ...
- Visibility (public, private, protected)

22

USC CSE Interaction Diagrams
 USC - Center for Software Engineering

- Describe how groups of objects interact with one another. Interaction diagrams show the behavior of objects during a particular time or time frame.
- On a conceptual level, interaction diagrams may show the collaboration of use cases and conceptual classes -> things the user and customer can relate to.
- On a specification and implementation level the focus is on the collaboration of software system components.

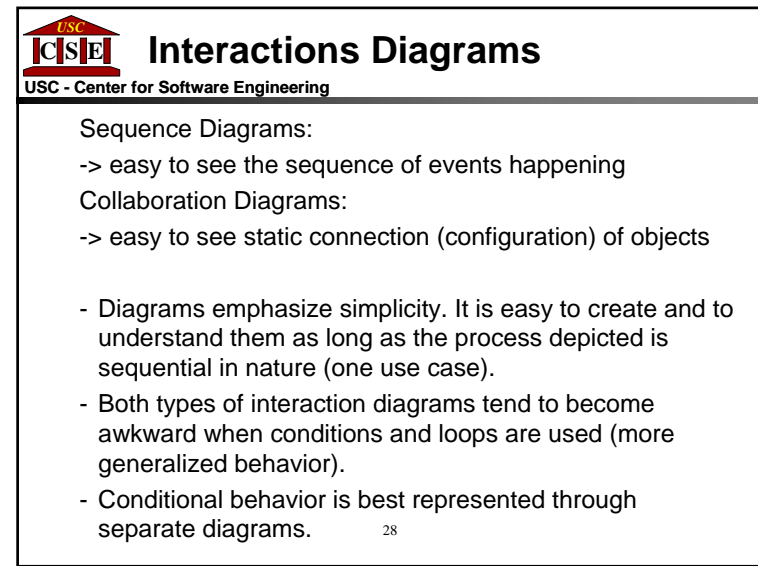
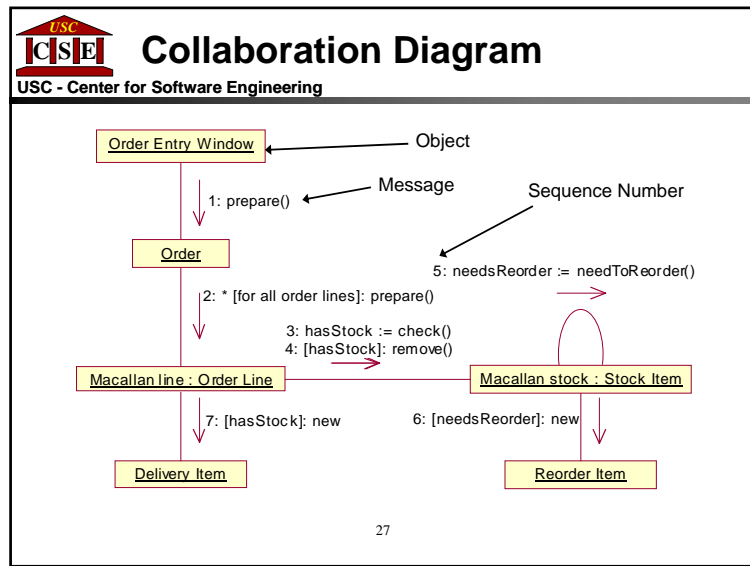
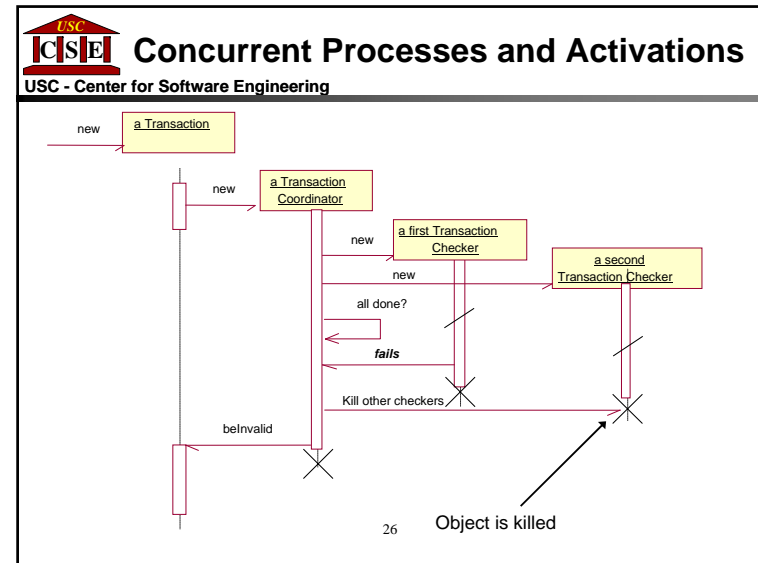
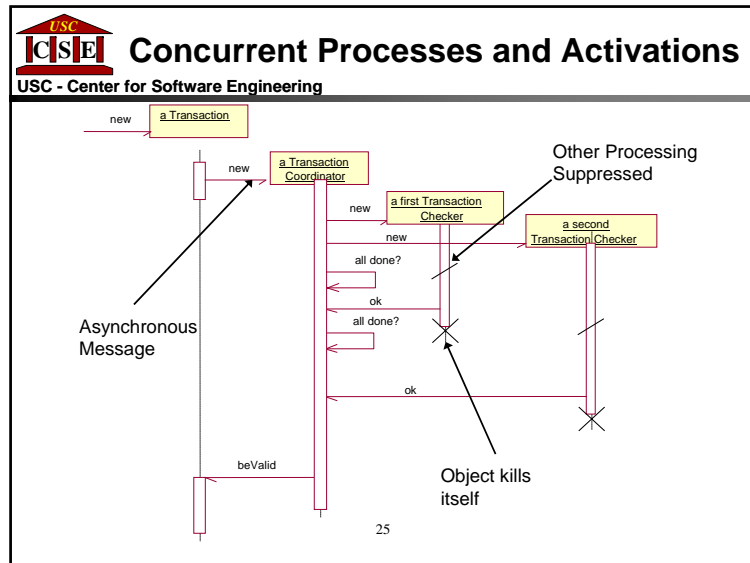
Two basic types

- Sequence Diagrams
- Collaboration Diagrams

23

USC CSE Sequence Diagram
 USC - Center for Software Engineering

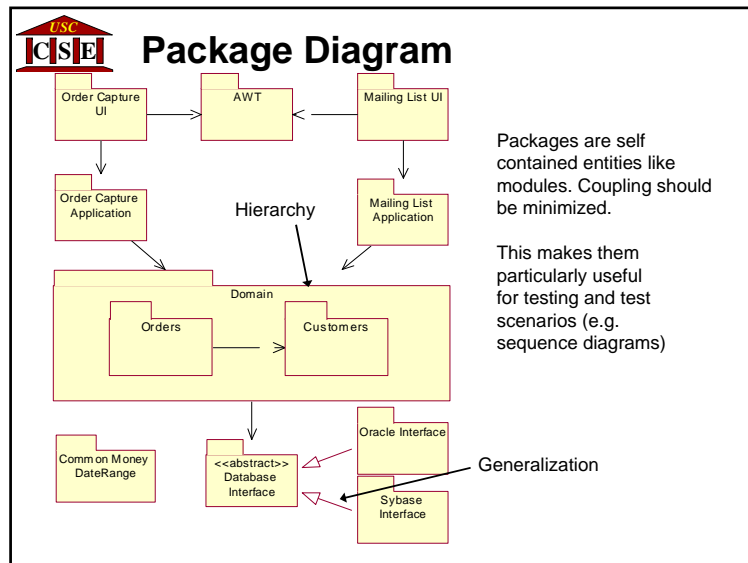
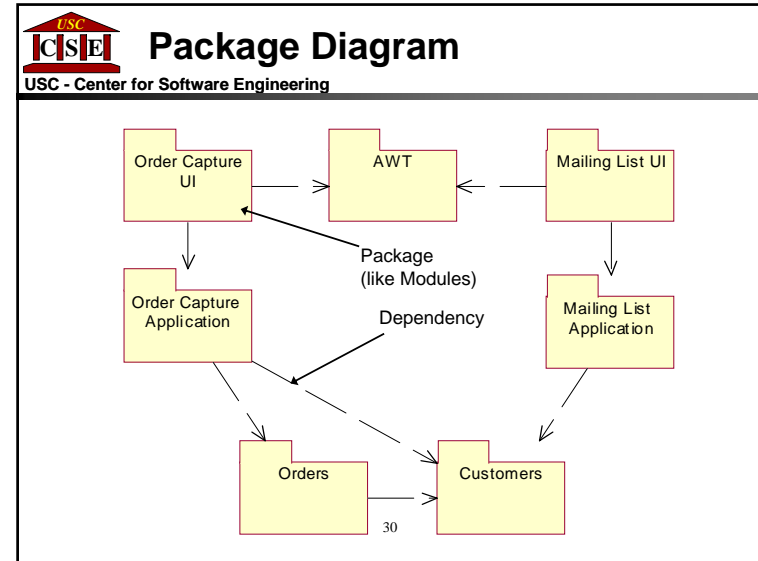
24



USC CSE Package Diagrams
 USC - Center for Software Engineering

- Help in breaking down large software systems into smaller pieces.
- Before OO, functional decomposition used to separate behavioral decomposition from data decomposition. This led to a system decomposition that was incompatible with OO since here behavior and data cannot be separated.
- With OO, both forms of decompositions are merged and, thus, packages are really just a grouping mechanism for classes.
- Relationships between packages are similar to those of classes, however, package interrelationships should decrease coupling and increase cohesion.

29



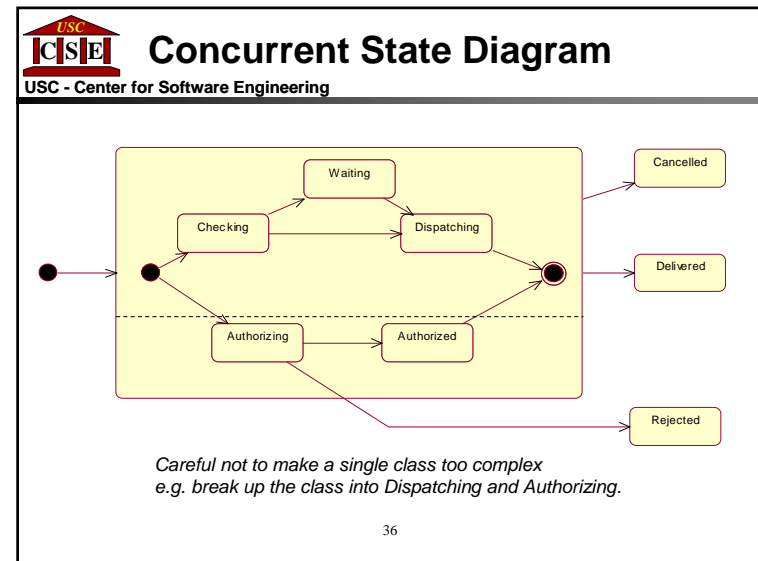
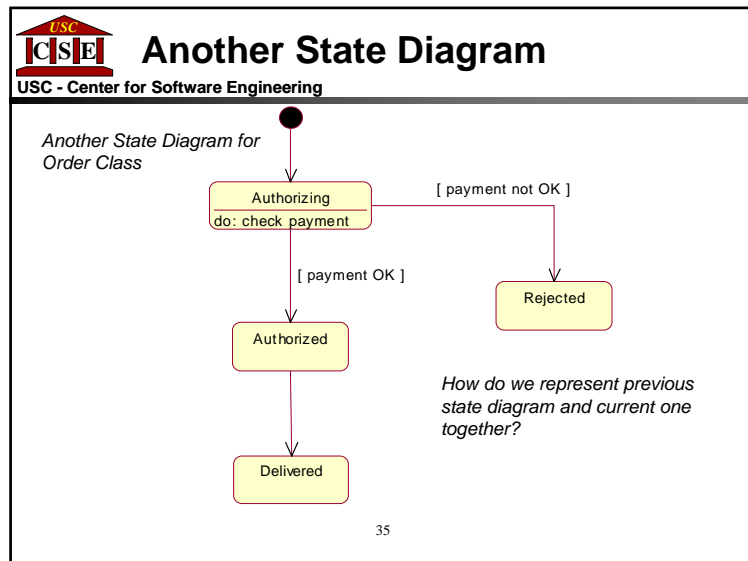
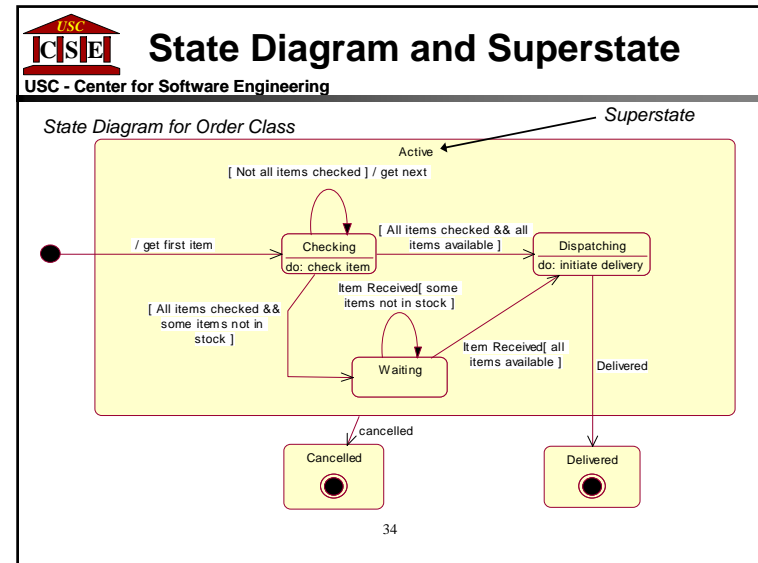
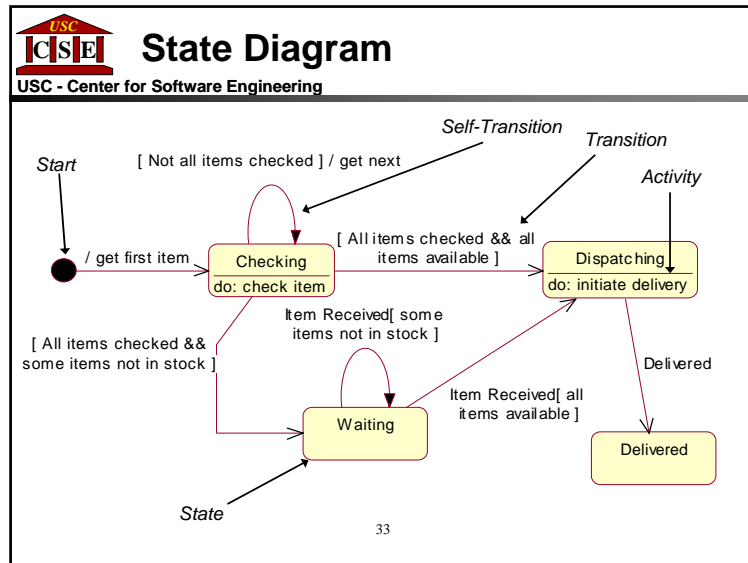
USC CSE State Diagrams
 USC - Center for Software Engineering

State Diagrams have been around for a long time, even before OO. They are very useful in presenting what states an object/class may go through in its lifetime.

In UML a state diagram represent the life of a single class or object. This is necessary because state diagrams are a non-OO design technique. Using state diagrams in OO on a system level may yield a functional system decomposition. Another problem is the state explosion problem.

- => use state diagrams only with classes
- => If used on a higher level (e.g. to describe use cases) be careful when using its decomposition during design.

32



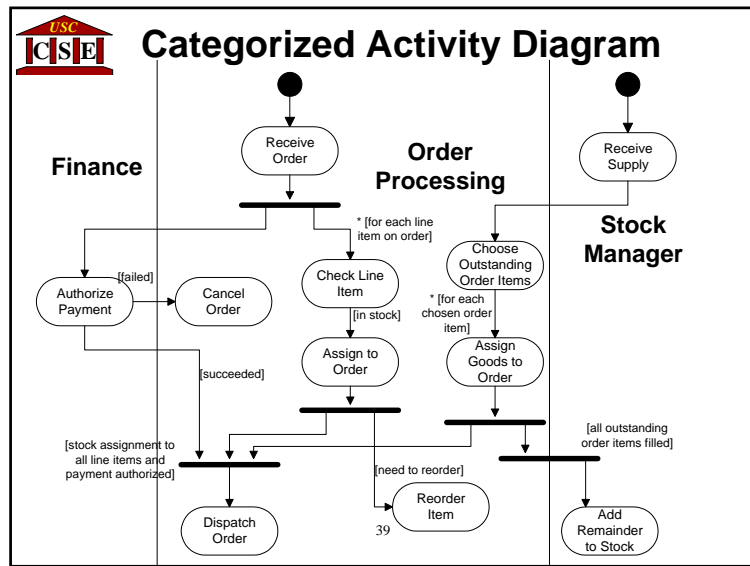
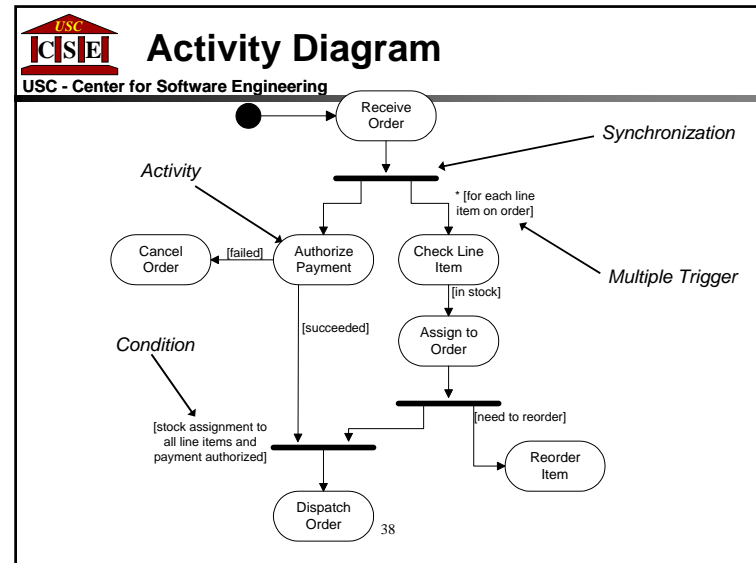
USC CSE Activity Diagrams
 USC - Center for Software Engineering

Activity Diagrams are the newest addition to UML and they are also unique in that they did not exist before. Activity Diagrams merge concepts from Event Diagrams, SDL state modeling, and Petri nets.

Activity Diagrams appear like enriched state diagrams, however, the meaning of components and connectors are not quite the same.

Activity diagrams cause a functional system breakdown and must be handled carefully. Nevertheless, through their emphasis on activities (like operators in classes), a OO interpretation of activity diagrams is much easier.

37

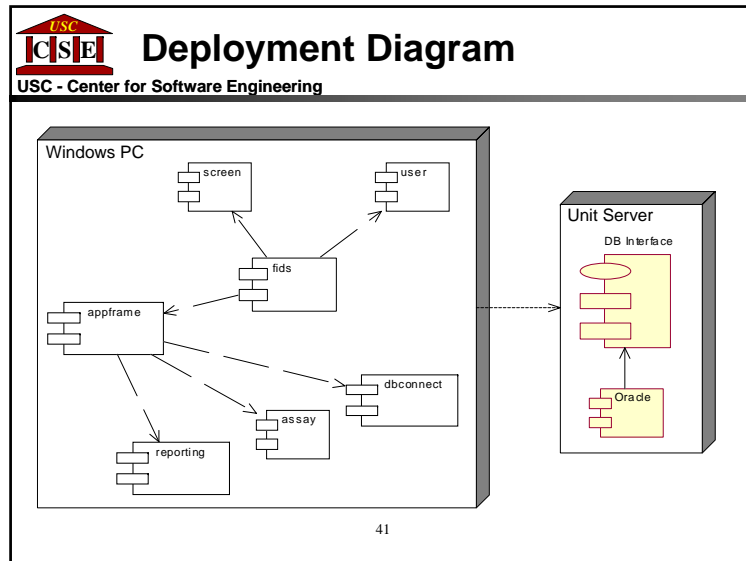


USC CSE Deployment Diagrams
 USC - Center for Software Engineering

Deployment Diagrams show the physical dependencies between software and hardware components.

- Nodes represent computational units
- Components represent packages (subsystems) and
- Connectors represent communication paths.

40



UML Properties

USC - Center for Software Engineering

- Components -> classes and objects
 - abstraction of the real world
 - self contained (method interfaces and data/state)
 - dependencies among objects are described explicitly
- Connectors -> "arrows"
 - Association/dependency/inheritance
 - Links (method invocation)
 - Shared state (accessed through method invocation)

=> Little information about distribution, parallelism
=> Object reuse/understanding

42

UML Properties

USC - Center for Software Engineering

However, ...

- UML semantics are very ambiguous
- Tools typically only support parts of UML but not entirety
- Uses both OO and functional development concepts

Nevertheless, ...

- Common meta model eases communication and interaction
- More precise meaning (semantics) can be added (customized)

43

Conclusions

USC - Center for Software Engineering

- UML notation is not very complex (easy to understand) but there are many subtleties.
- UML is not well suited for highly formal domains (too many ambiguities).
- UML supports all of software development (few restrictions)
- UML is not very analyzable

=> use UML during general development process
=> use UML as a refinement of ADLs

44