Object Oriented Design

David Talby

Welcome!

- Introduction
- UML
  - Use Case Diagrams
  - Interaction Diagrams
  - Class Diagrams
- Design Patterns
  - Composite

UML

- Unified Modeling Language
  - Standard for describing designs
  - Visual: a set of diagrams
- Unifies entire design process:
  - Use Cases for requirements
  - Static class diagrams
  - Object & Interaction diagrams
  - Components, Packages, …

Use Cases

- A *Use case* is a narrative document that describes the sequence of events of an actor using a system to complete a process.
- A *use case diagram* visualizes relationships between a system's use cases and actors

Use Case Document

Name: Sell Item

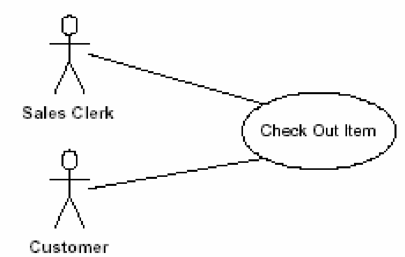
Initiator: Customer

Type: Primary, Required

Actions: 1. Customer asks for X
2. Sales clerk checks if X is in stock
3. …

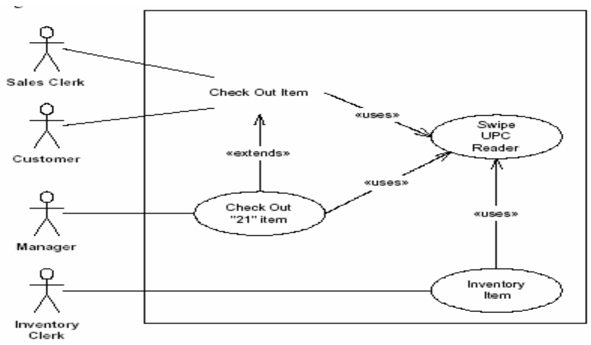
Error Case A: if … then …

Use Case Diagram



- Actors participate in use cases
- Use cases use or extend others

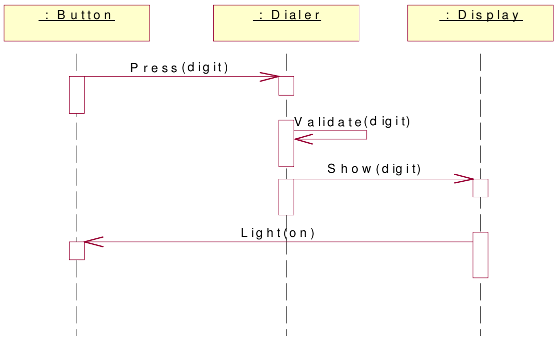
Use Case Diagram II



Sequence Diagrams

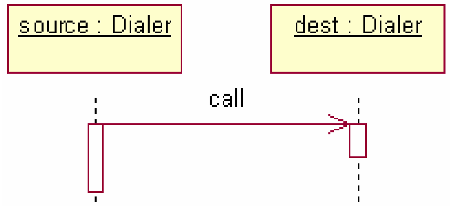
- A sequence diagram visualizes an ordered interaction between objects, by showing the messages sent between them.
- One way to start a design is:
  - Translating a UC to a sequence
  - Turn its actions to messages

A Sequence Diagram



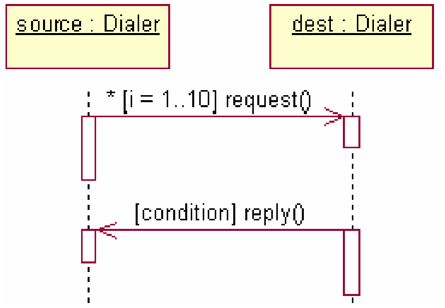
Sequence Diagrams II

- Good time-line visualization
- Supports messages to self
- Supports object of same class:



Sequence Diagrams III

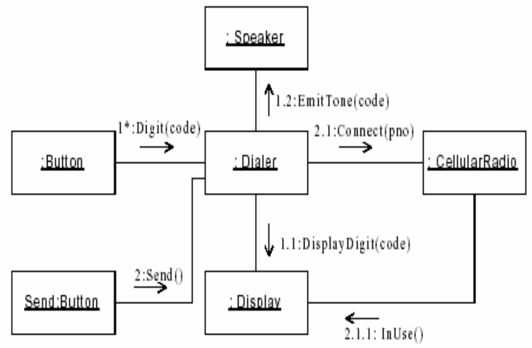
- Supports conditions and loops:



Collaboration Diagrams

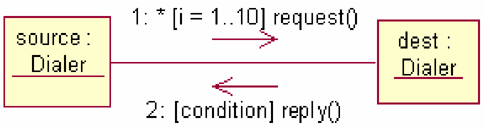
- Another visual way to show the same information that a sequence diagram shows
- Uses numbering of messages instead of a timeline
- Both diagrams are also called interaction diagrams

Collaboration Diagrams



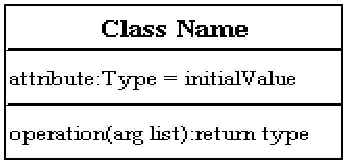
Collaboration Diagrams

- Good object-centric view
- Identical to Sequence diagrams
  - Loops, conditions, arguments
  - Automatic translation possible



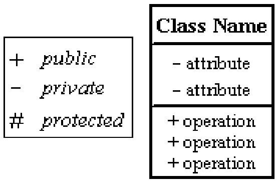
Class Diagrams

- Class diagrams show the static picture of the system’s classes
- And relationships between them


Class Name
attribute:Type = initialValue
operation(arg list):return type

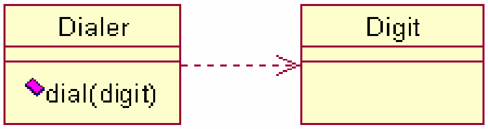
Diagramming a Class

- All Additions are optional
  - Types and argument lists
  - Initial values and constants



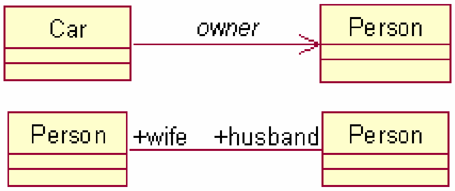
Dependency

- Class A requires B to compile
- Creates it (Instantiates)
  - Gets an argument



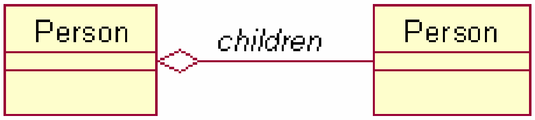
Association

- Class A points to a B object
  - Can be Uni- or Bi-Directional
  - Each role can be named



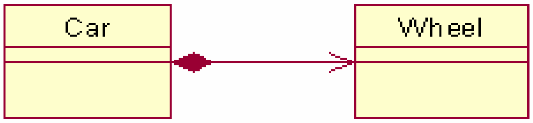
Aggregation

- Class A contains a list of B's
  - But B's can exist without A's
  - Can be Uni- or Bi-Directional
  - Can be numbered



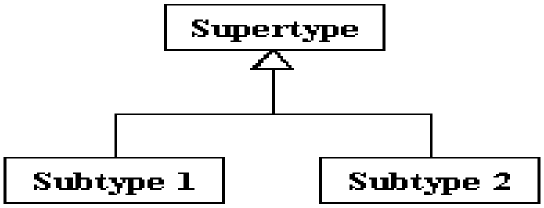
Composition

- Class A contains a list of B's
  - B's are destroyed with their container A is destroyed
  - Can be Uni/Bi-Di, Numbered



Inheritance

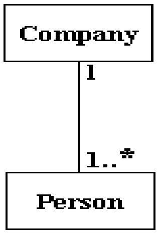
- Class A inherits from class B



Numbering

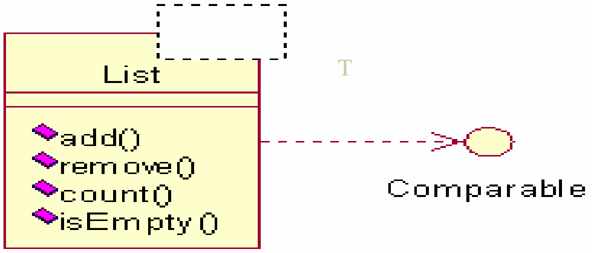
- Association, Aggregation and Composition can constraint lists

1	*no more than one*
0..1	*zero or one*
*	*many*
0..*	*zero or many*
1..*	*one or many*



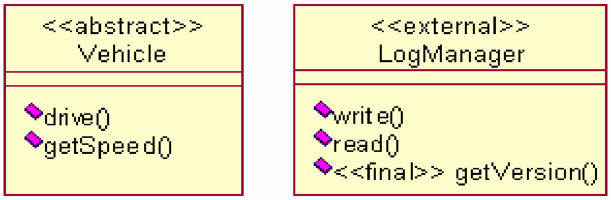
Templates & Interfaces

- Both are supported



Stereotypes

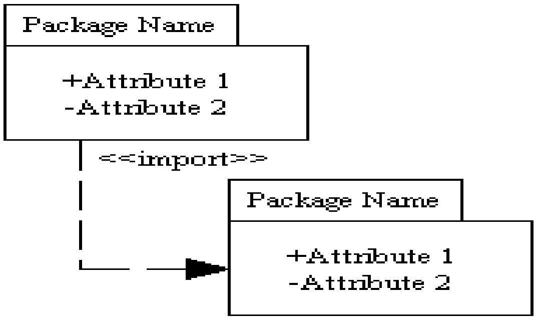
- Attributes of classes or methods
  - Standard: Interface, Abstract
  - Can be project-specific



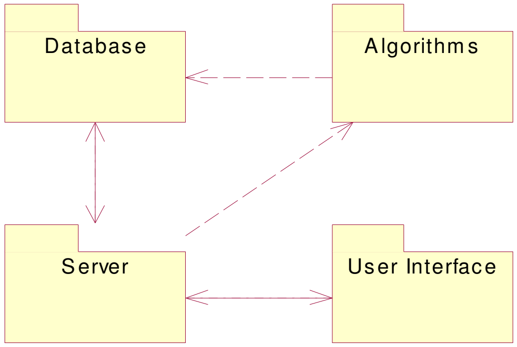
Package Diagrams

- Organize a system’s elements into related groups to minimize dependencies between them
- Provides a high-level view
- A UML package is analogous to
  - a Java package
  - a C++ namespace

Package Diagrams II

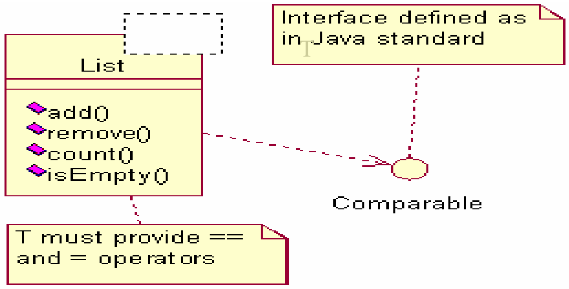


Package Diagrams III



UML Notes

- Can be attached to anything


List
add()
remove()
count()
isEmpty()
Interface defined as in Java standard
Comparable
T must provide == and = operators

Other UML Diagrams

- State diagrams illustrate the states of a system or an object, and events that cause state transitions
- Component diagrams show compiler and runtime dependencies between components.
- Deployment diagrams show the distribution of processes and components to processing nodes.
- UML is a large standard

Design Patterns

- O-O Design is Hard
- Errors are expensive
- Reuse experts’ designs

- Pattern = Documented experience

Expected Benefits

- Finding the right classes
- Finding them faster
- Common design jargon
- Consistent format
- Coded infrastructures

O-O Programming

- An interface is a contract to clients.
- A class implements interface(s).
- Objects are instances of classes.
- Objects are only accessed through their public interfaces.
- Only two relations between classes: Inheritance and composition

Object Relationships

- Inheritance: Static and efficient, but exposes and couples modules
- Composition: Hides more from client and can change dynamically
- *Gang of Four:*
  "Favor composition over inheritance"
- Dijkstra: "*Most problems in computer science can be solved by another level of indirection*"

Designing for Change

- The Open-Closed Principle
- The Single-Choice Principle
- Non-clairvoyance
- Key Issue: Prepare for change!

- Well, prepare for what?

Causes of Redesign

- Dependence on hardware or software platform
- Dependence on representation or implementation
- Specifying a class upon creation
- Algorithmic dependence
- Tight coupling
- Overuse of inheritance
- Inability to alter classes easily

Pattern Categories

- **Creational** - Replace explicit creation problems, prevent platform dependencies
- **Structural** - Handle unchangeable classes, lower coupling and offer alternatives to inheritance
- **Behavioral** - Hide implementation, hides algorithms, allows easy and dynamic configuration of objects

Pattern of Patterns

- Encapsulate the varying aspect
- Interfaces
- Inheritance describes variants
- Composition allows a dynamic choice between variants

Criteria for success:

Open-Closed Principle
Single Choice Principle

1. Composite

- A program must treat simple and complex objects uniformly
- For example, a painting program has simple objects (lines, circles and texts) as well as composite ones (wheel = circle + six lines).

The Requirements

- Treat simple and complex objects uniformly in code - move, erase, rotate and set color work on all
- Some composite objects are defined statically (wheels), while others dynamically (user selection)
- Composite objects can be made of other composite objects
- We need a smart **data structure**

The Solution

- All simple objects inherit from a common interface, say *Graphic*:

```
class Graphic {
   void move(int x, int y) = 0;
   void setColor(Color c) = 0;
   void rotate(double angle) = 0;
}
```

- The classes *Line*, *Circle* and others inherit *Graphic* and add specific features (radius, length, etc.)

The Solution II

- This new class inherits it as well:

```
class CompositeGraphic
  : public Graphic,
    public list<Graphic>
{
  void rotate(double angle) {
    for (int i=0; i<count(); i++)
      item(i)- >rotate();
  }
}
```

The Solution III

- Since a *CompositeGraphic* is a list, it had *add(), remove()* and *count()* methods
- Since it is also a *Graphic*, it *has rotate()*, *move()* and *setColor()* too
- Such operations on a composite object work using a ‘forall’ loop
- Works even when a composite holds other composites - results in a tree-like data structure

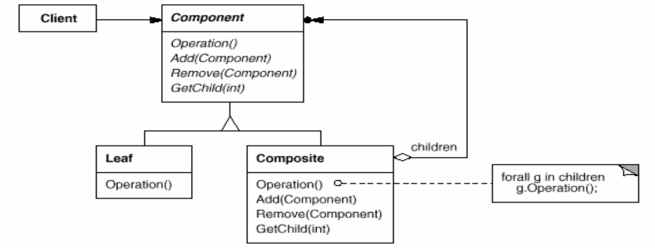
The Solution IV

- Example of creating a composite:

```
CompositeGraphic *cg;
cg = new CompositeGraphic();
cg->add(new Line(0,0,100,100));
cg->add(new Circle(50,50,100));
cg->add(t); // dynamic text graphic
cg->remove(2);
```

- Can keep order of inserted items if the program needs it

The GoF UML



- Single Inheritance
- Root has *add()*, *remove()* methods

The Fine Print

- Sometimes useful to let objects hold a pointer to their parent
- A composite may cache data about its children (count is an example)
- Make composites responsible for deleting their children
- Beware of circles in the graph!
- Any data structure to hold children will do (list, array, hashtable, etc.)

Known Uses

- In almost all O-O systems
- Document editing programs
- GUI (a form is a composite widget)
- Compiler parse trees (a function is composed of simpler statements or function calls, same for modules)
- Financial assets can be simple (stocks, options) or a composite portfolio

Pattern of Patterns

- Encapsulate the varying aspect
- Interfaces
- Inheritance describes variants
- Composition allows a dynamic choice between variants

Criteria for success:

Open-Closed Principle
Single Choice Principle