CSC301: Introduction to Software Engineering

Lecture 1

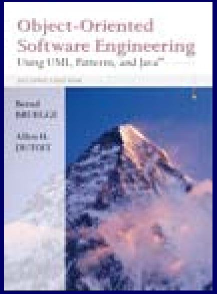
Wael Aboulsaadat

Course Info…

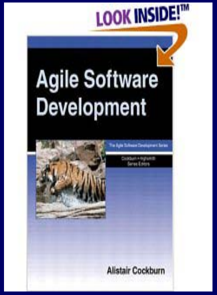
- Instructor: Wael Aboulsaadat
- Lecture: Wednesday 6:10-8:00 @ BA 1180
- Office Hour: Wednesday 4:00 → 5:00pm @ BA 4261
- Email: wael@cs.toronto.edu
- Web page: Blackboard http://portal.utoronto.ca/
- Course Software: check External Links in Blackboard (next week)
- Tutorial: Wednesday 8:10pm → 9:00pm @ BA 2135 && BA 2139
- TA: Atalay Ozgovde <atalay@cs.toronto.edu>
- TA: Jennifer Horkoff < jenhork@cs.toronto.edu>

Recommended Text Book(s)

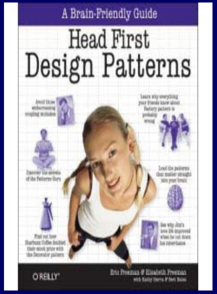
1. “Object-Oriented Software Engineering: Using UML, Patterns and Java”. Bernd Bruegge and Allen H. Dutoit
   ISBN: 978-0072465631


Object-Oriented Software Engineering
Using UML, Patterns, and Java
Bernd Bruegge
Allen H. Dutoit

2. “Agile Software Development”. Alistair Cockburn
   ISBN: 978-0201699692



3. “Head First Design Patterns”. (Head First). Elisabeth Freeman, Eric Freeman, Bert Bates, and Kathy Sierra
   ISBN-13: 978-0596007126



Grading

Project	65 %
Exam	35 %

Course Topics

- Introduction to Soft Eng
- Object Oriented Analysis
- Object Oriented Design
- Design Patterns
- Software Development Life Cycle
- Agile Software Development
- Refactoring
- Test Driven Development
- Software Project Management

Software Engineering: definition

Software Engineering is a collection of techniques, methodologies and tools that help
with the production of

- a high quality software system
- with a given budget
- before a given deadline

while change occurs.

Software Engineering: problem solving approach

- Methodologies:
  - Collection of techniques applied across software development and unified by a philosophical approach
- Tools:
  - Instrument or automated systems to accomplish a technique

Software Engineering: activities

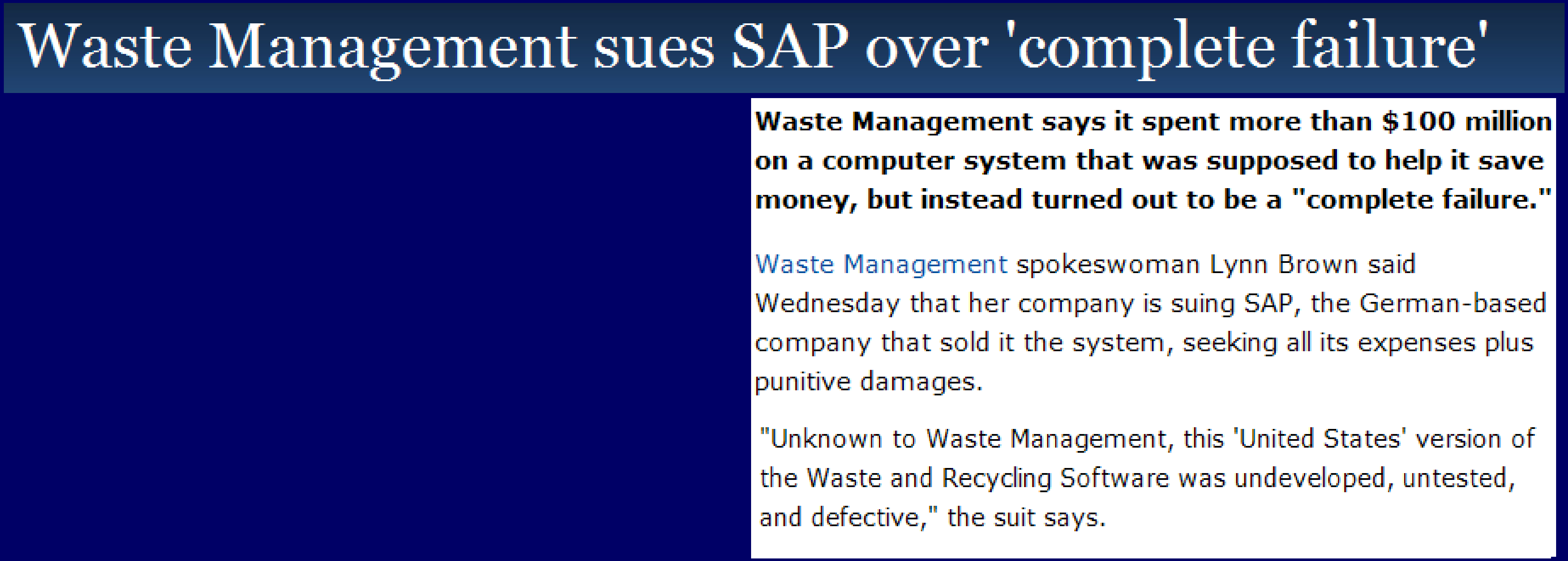
1. Analysis:
   - Understand the nature of the problem and break the problem into pieces
2. Synthesis:
   - Make components and put them together into a large structure

Software Engineering: the crisis

- What’s the problem?

Software Engineering: the crisis

- Example



Waste Management sues SAP over 'complete failure'

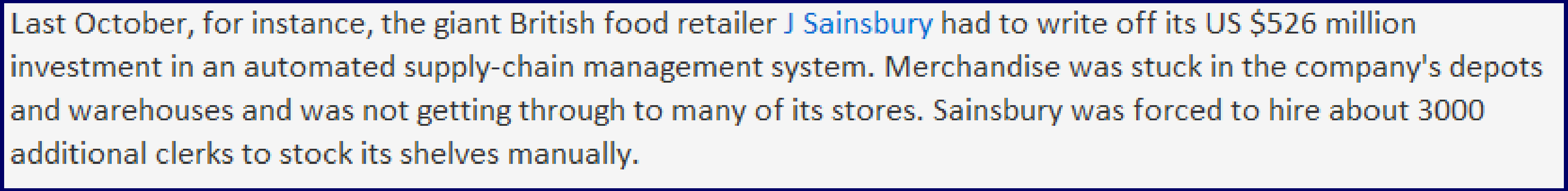
Waste Management says it spent more than $100 million on a computer system that was supposed to help it save money, but instead turned out to be a "complete failure."

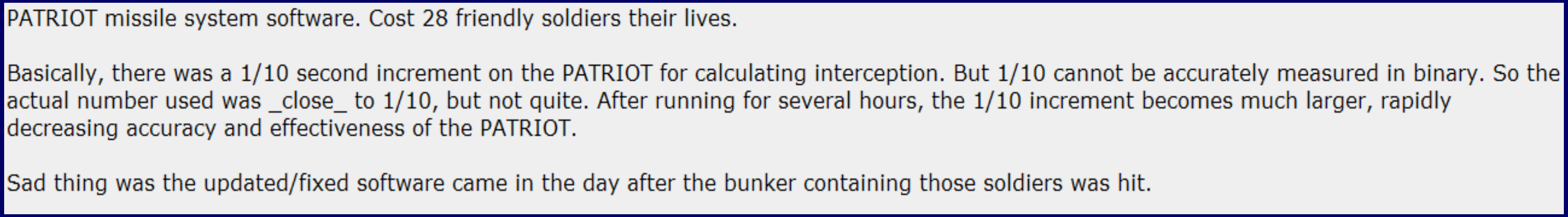
Waste Management spokeswoman Lynn Brown said Wednesday that her company is suing SAP, the German-based company that sold it the system, seeking all its expenses plus punitive damages.

"Unknown to Waste Management, this 'United States' version of the Waste and Recycling Software was undeveloped, untested, and defective," the suit says.

Software Engineering: the crisis

- More examples


Last October, for instance, the giant British food retailer J Sainsbury had to write off its US $526 million investment in an automated supply-chain management system. Merchandise was stuck in the company's depots and warehouses and was not getting through to many of its stores. Sainsbury was forced to hire about 3000 additional clerks to stock its shelves manually.

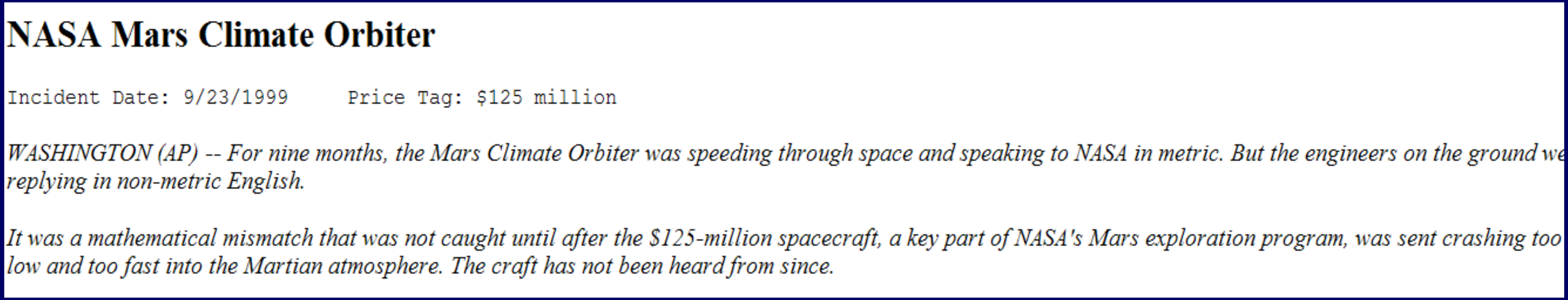

PATRIOT missile system software. Cost 28 friendly soldiers their lives.

Basically, there was a 1/10 second increment on the PATRIOT for calculating interception. But 1/10 cannot be accurately measured in binary. So the actual number used was _close_ to 1/10, but not quite. After running for several hours, the 1/10 increment becomes much larger, rapidly decreasing accuracy and effectiveness of the PATRIOT.

Sad thing was the updated/fixed software came in the day after the bunker containing those soldiers was hit.

Software Engineering: the crisis

- More examples!

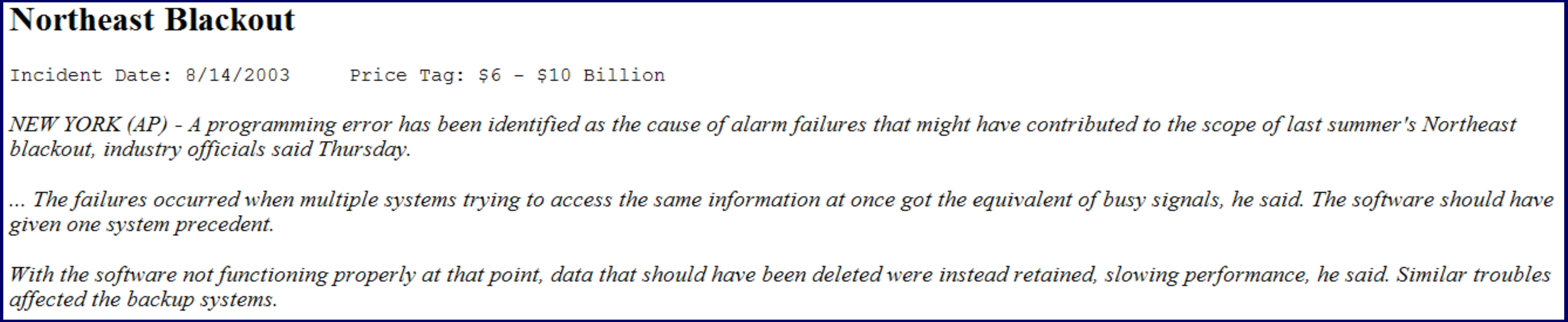


NASA Mars Climate Orbiter

`Incident Date: 9/23/1999      Price Tag: $125 million`

WASHINGTON (AP) -- For nine months, the Mars Climate Orbiter was speeding through space and speaking to NASA in metric. But the engineers on the ground we replying in non-metric English.

It was a mathematical mismatch that was not caught until after the $125-million spacecraft, a key part of NASA's Mars exploration program, was sent crashing too low and too fast into the Martian atmosphere. The craft has not been heard from since.



Northeast Blackout

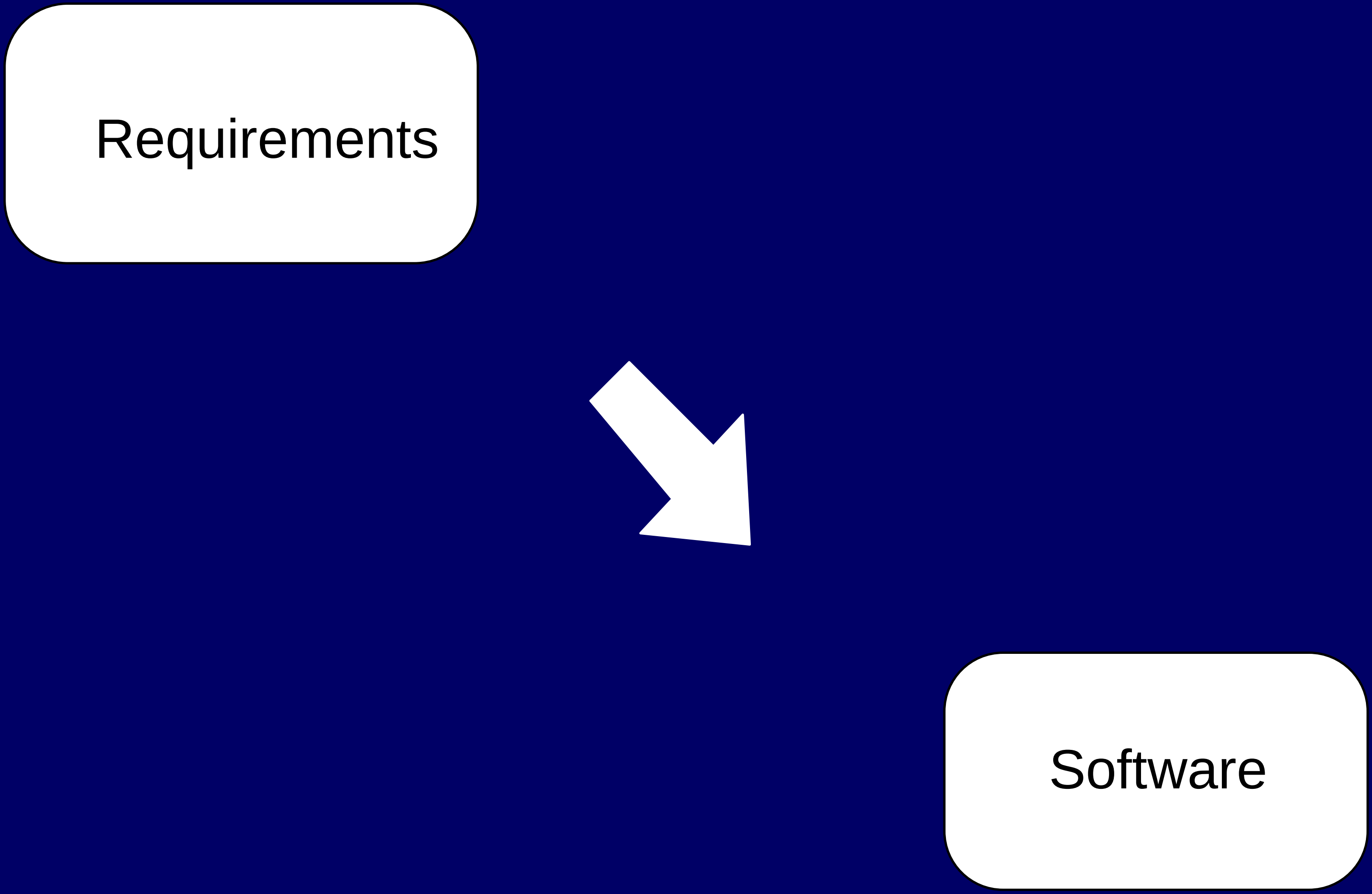
`Incident Date: 8/14/2003      Price Tag: $6 - $10 Billion`

NEW YORK (AP) - A programming error has been identified as the cause of alarm failures that might have contributed to the scope of last summer's Northeast blackout, industry officials said Thursday.

... The failures occurred when multiple systems trying to access the same information at once got the equivalent of busy signals, he said. The software should have given one system precedent.

With the software not functioning properly at that point, data that should have been deleted were instead retained, slowing performance, he said. Similar troubles affected the backup systems.

What's happening ?!



Factors affecting the quality of software

- Complexity:
  - The system is so complex that no single programmer can understand it anymore
  - The introduction of one bug fix causes another bug
- Change:
  - The “Entropy” of a software system increases with each change: Each implemented change erodes the structure of the system which makes the next change even more expensive (“Second Law of Software Dynamics”).
  - As time goes on, the cost to implement a change will be too high, and the system will then be unable to support its intended task. This is true of all systems, independent of their application domain or technological base.

Dealing with Complexity

1. Abstraction
2. Decomposition
3. Hierarchy

Dealing with Complexity: abstraction

- Inherent human limitation to deal with complexity
  - The 7 +- 2 phenomena
- Chunking: Group collection of objects
- Ignore unessential details: => Models

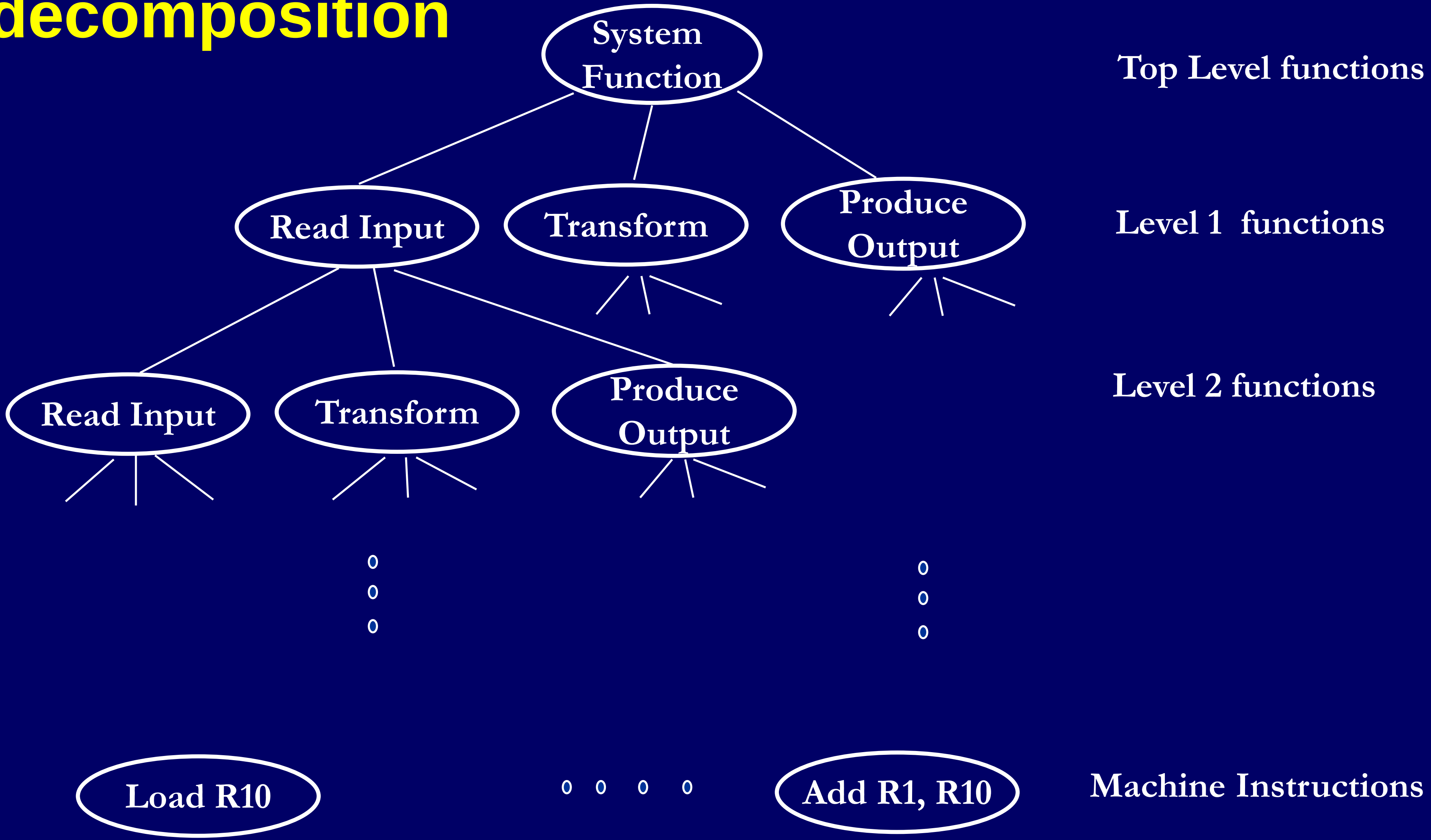
Dealing with Complexity: abstraction models

- System Model:
  - Object Model: What is the structure of the system? What are the objects and how are they related?
  - Functional model: What are the functions of the system? How is data flowing through the system?
  - Dynamic model: How does the system react to external events? How is the event flow in the system ?
- Task Model:
  - PERT Chart: What are the dependencies between the tasks?
  - Schedule: How can this be done within the time limit?
  - Org Chart: What are the roles in the project or organization?
- Issues Model:
  - What are the open and closed issues? What constraints were posed by the client? What resolutions were made?

Dealing with Complexity: decomposition

- A technique used to master complexity (“divide and conquer”)
- Functional decomposition
  - The system is decomposed into modules
  - Each module is a major processing step (function) in the application domain
  - Modules can be decomposed into smaller modules
- Object-oriented decomposition
  - The system is decomposed into classes (“objects”)
  - Each class is a major abstraction in the application domain
  - Classes can be decomposed into smaller classes

Dealing with Complexity: functional decomposition



Dealing with Complexity: decomposition

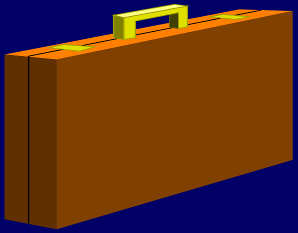
- A technique used to master complexity (“divide and conquer”)

- Object-oriented decomposition
  - The system is decomposed into classes (“objects”)
  - Each class is a major abstraction in the application domain
  - Classes can be decomposed into smaller classes

Dealing with Complexity: decomposition

- A technique used to master complexity ("divide and conquer")

- Object-oriented decomposition
  - The system is decomposed into classes ("objects")
  - Each class is a major abstraction in the application domain
  - Classes can be decomposed into smaller classes



Dealing with Complexity: decomposition

- A technique used to master complexity (“divide and conquer”)

- Object-oriented decomposition
  - The system is decomposed into classes (“objects”)
  - Each class is a major abstraction in the application domain
  - Classes can be decomposed into smaller classes

```
public class BriefCase {

int Capacity;
int Weight;

public void Open() {…}
public void Close() {…}
public void Carry() {…}
```

Dealing with Complexity: decomposition

- A technique used to master complexity (“divide and conquer”)

- Object-oriented decomposition
  - The system is decomposed into classes (“objects”)
  - Each class is a major abstraction in the application domain
  - Classes can be decomposed into smaller classes

Dealing with Complexity: decomposition

- A technique used to master complexity (“divide and conquer”)

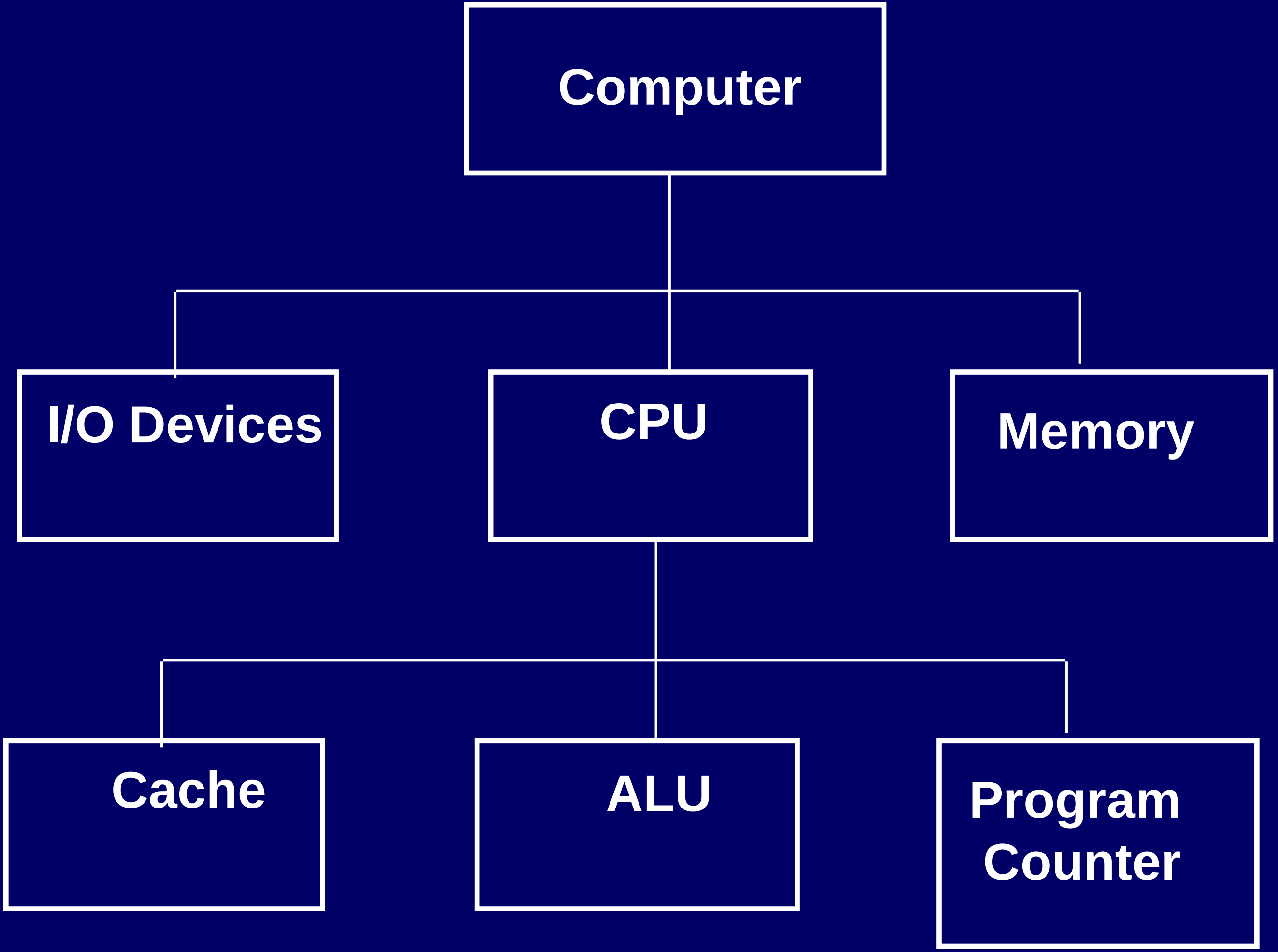
- Object-oriented decomposition
  - The system is decomposed into classes (“objects”)
  - Each class is a major abstraction in the application domain
  - Classes can be decomposed into smaller classes

```
public class BriefCase {

int Capacity;
int Weight;

public void Open() {…}
public void Close() {…}
public void Carry() {…}
public void SitOn() {…}
```

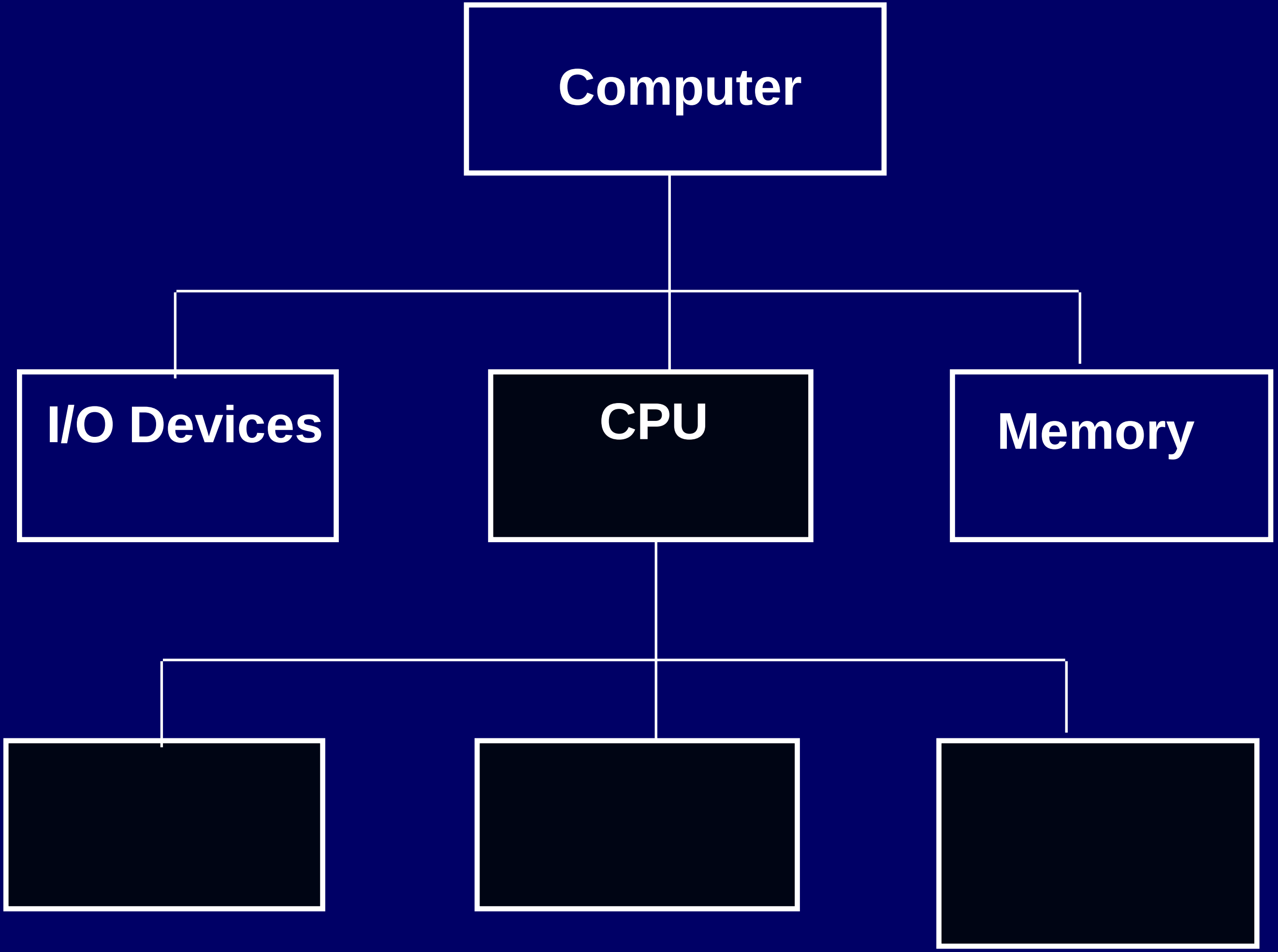
Dealing with Complexity: hierarchy

- We got abstractions and decomposition
  - This leads us to chunks (classes, objects) which we view with object model
- Another way to deal with complexity is to provide simple relationships between the chunks
- One of the most important relationships is hierarchy
- 2 important hierarchies
  - "Part of" hierarchy
  - "Is-kind-of" hierarchy

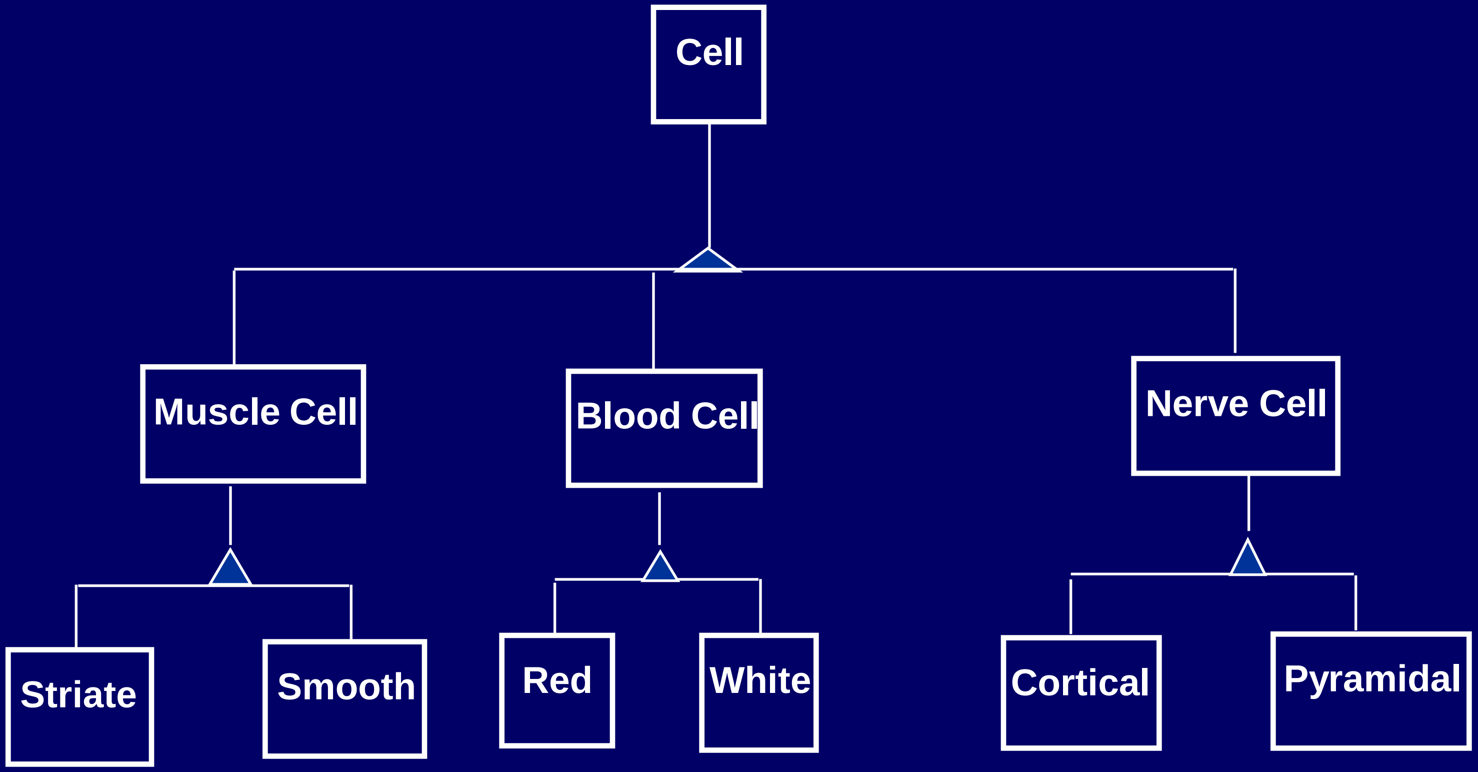
Dealing with Complexity: part of hierarchy



Dealing with Complexity: part of hierarchy



Dealing with Complexity: is-kind-of hierarchy

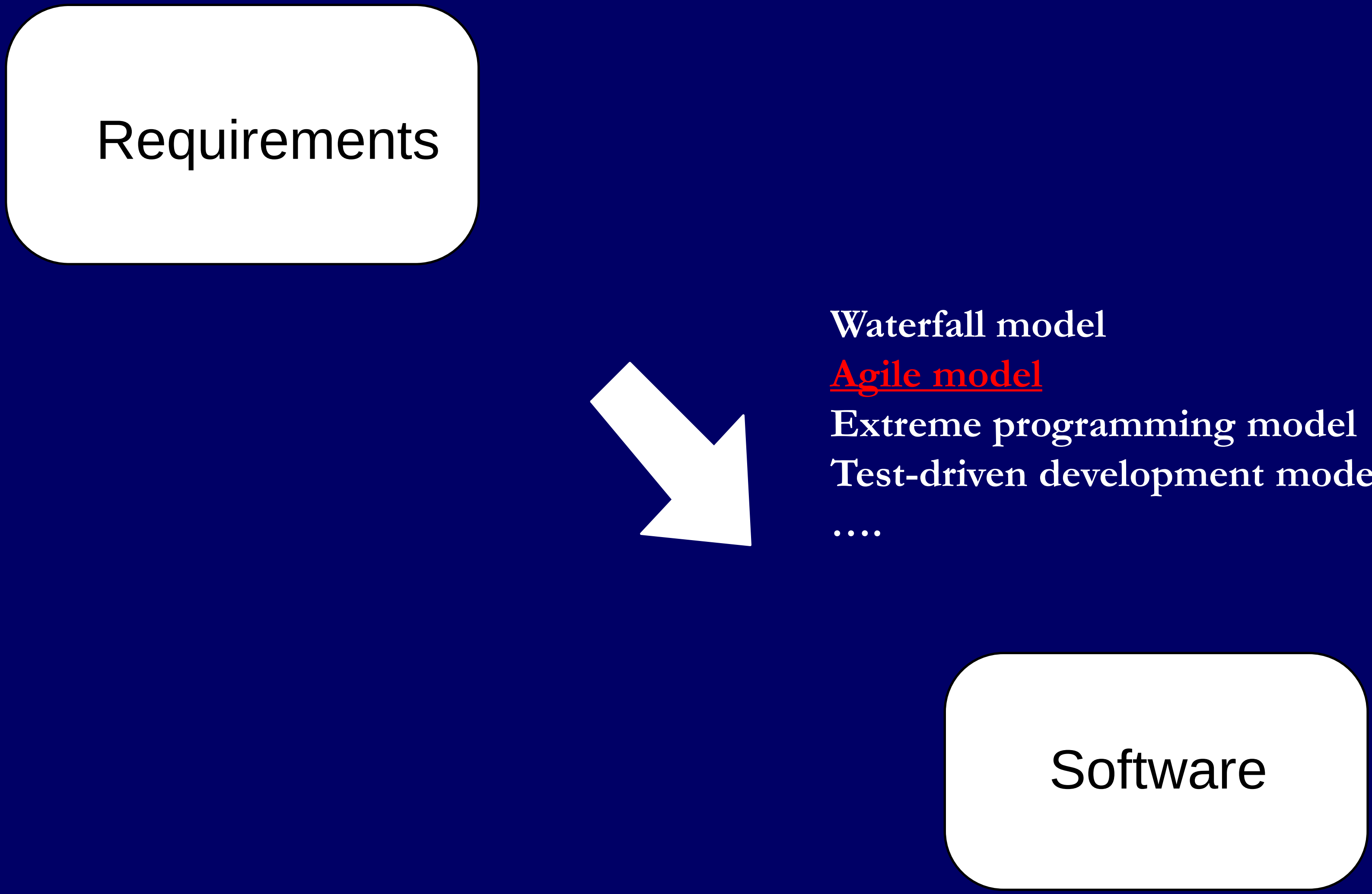


So where are we right now?

- Three ways to deal with complexity:
  - Abstraction
  - Decomposition
  - Hierarchy

- How can we do it right?

Software Development Models



Waterfall model
Agile model
Extreme programming model
Test-driven development model
….

UML

Why model software?

- Software is getting increasingly more complex
  - Windows XP > 40 MN lines of code
  - A single programmer cannot manage this amount of code in its entirety.
- Code is not easily understandable by developers who did not write it
- We need simpler representations for complex systems
  - Modeling is a mean for dealing with complexity

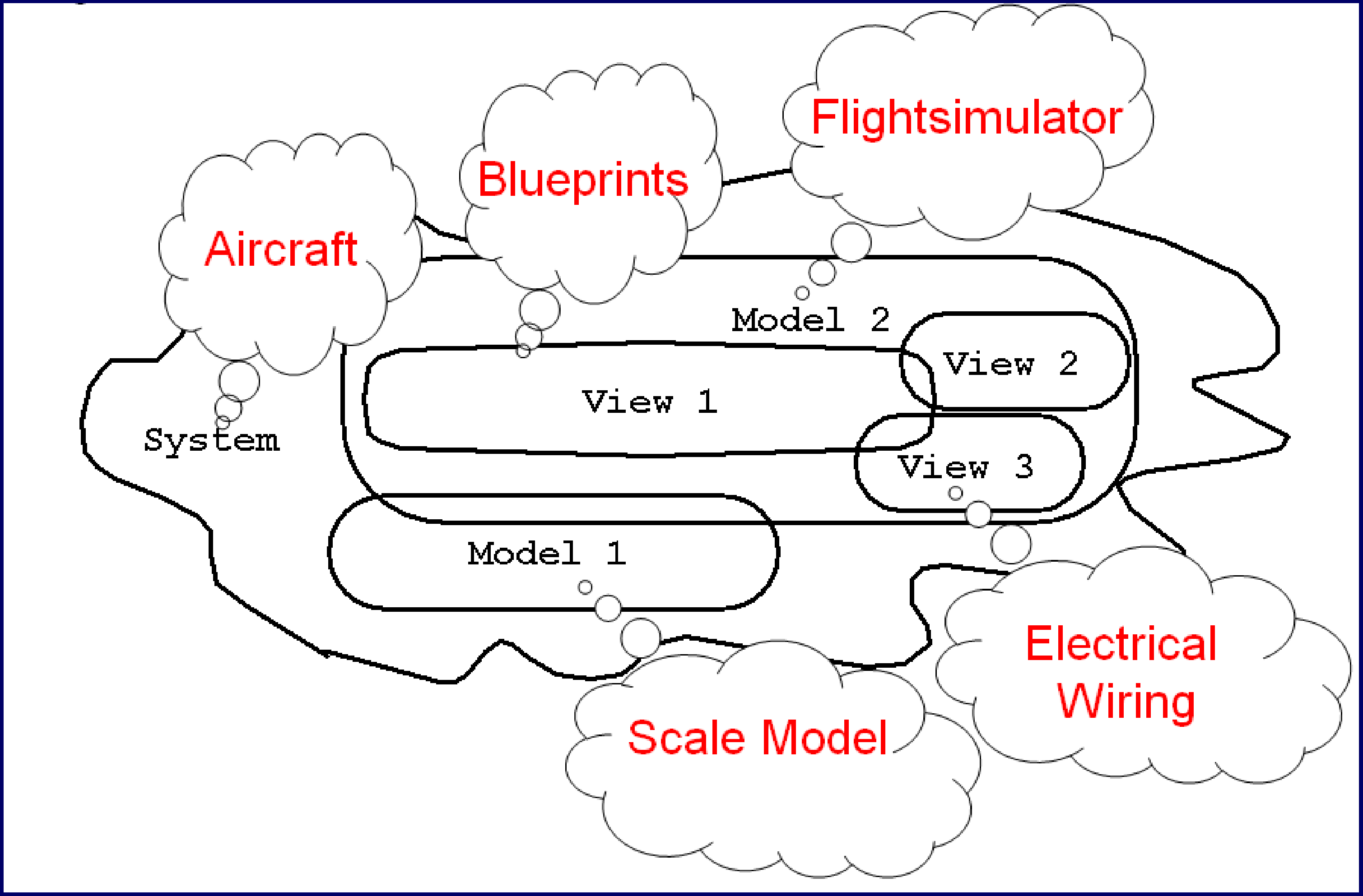
Systems, Models and Views

- A ***model*** is an abstraction describing a subset of a system
- A ***view*** depicts selected aspects of a model
- A ***notation*** is a set of graphical or textual rules for depicting views
- Views and models of a single system may overlap each other

Examples:

- System: Aircraft
- Models: Flight simulator, scale model
- Views: All blueprints, electrical wiring, fuel system

Systems, Models and Views



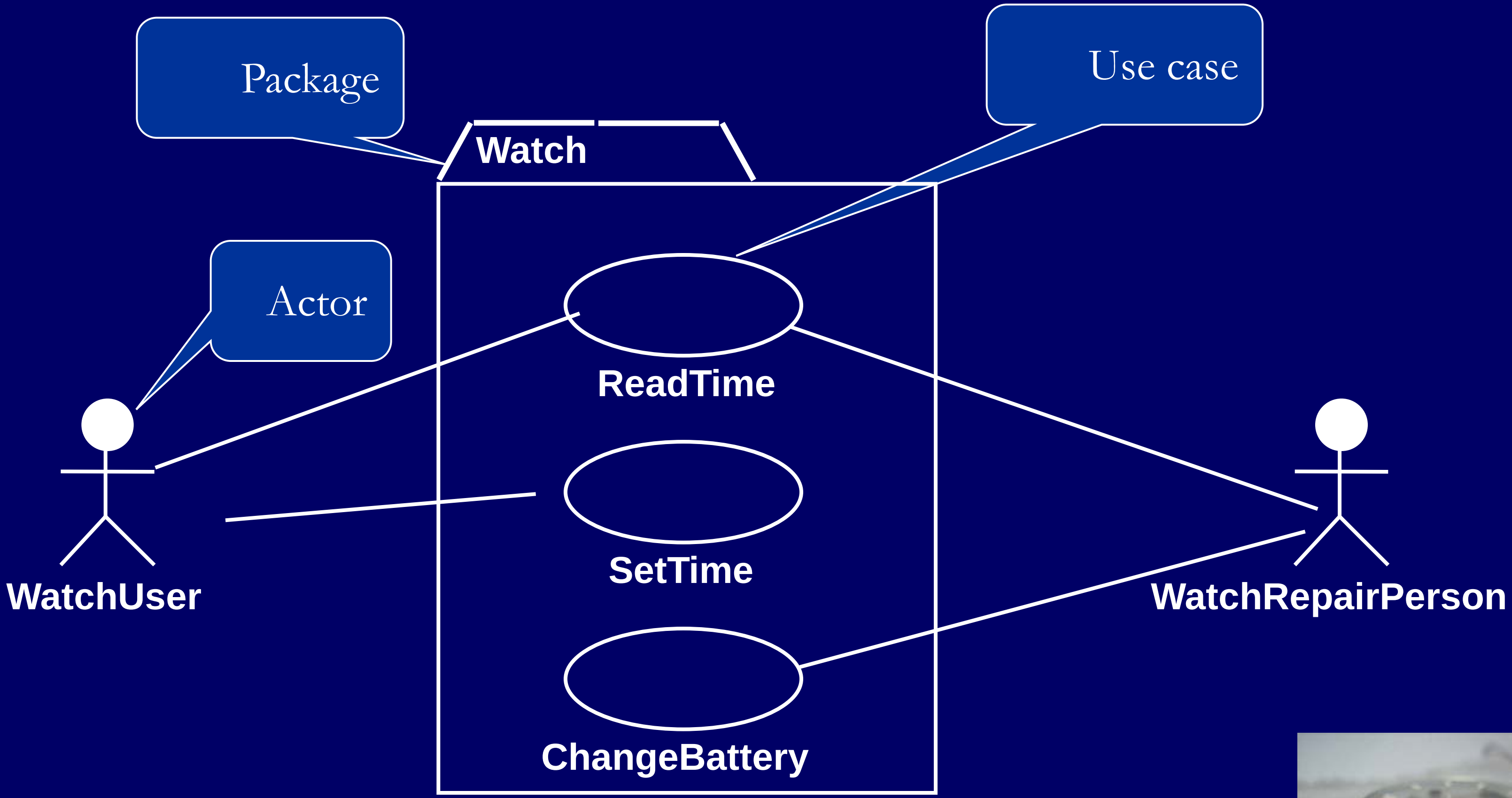
UML: First Pass

- You can model 80% of most problems by using about 20 % UML
- We teach you those 20%

UML First Pass

- Use case Diagrams
  - Describe the functional behavior of the system as seen by the user.
- Class diagrams
  - Describe the static structure of the system: Objects, Attributes, Associations
- Sequence diagrams
  - Describe the dynamic behavior between actors and the system and between objects of the system
- State chart diagrams
  - Describe the dynamic behavior of an individual object (essentially a finite state automaton)
- Activity Diagrams
  - Model the dynamic behavior of a system, in particular the workflow (essentially a flowchart)

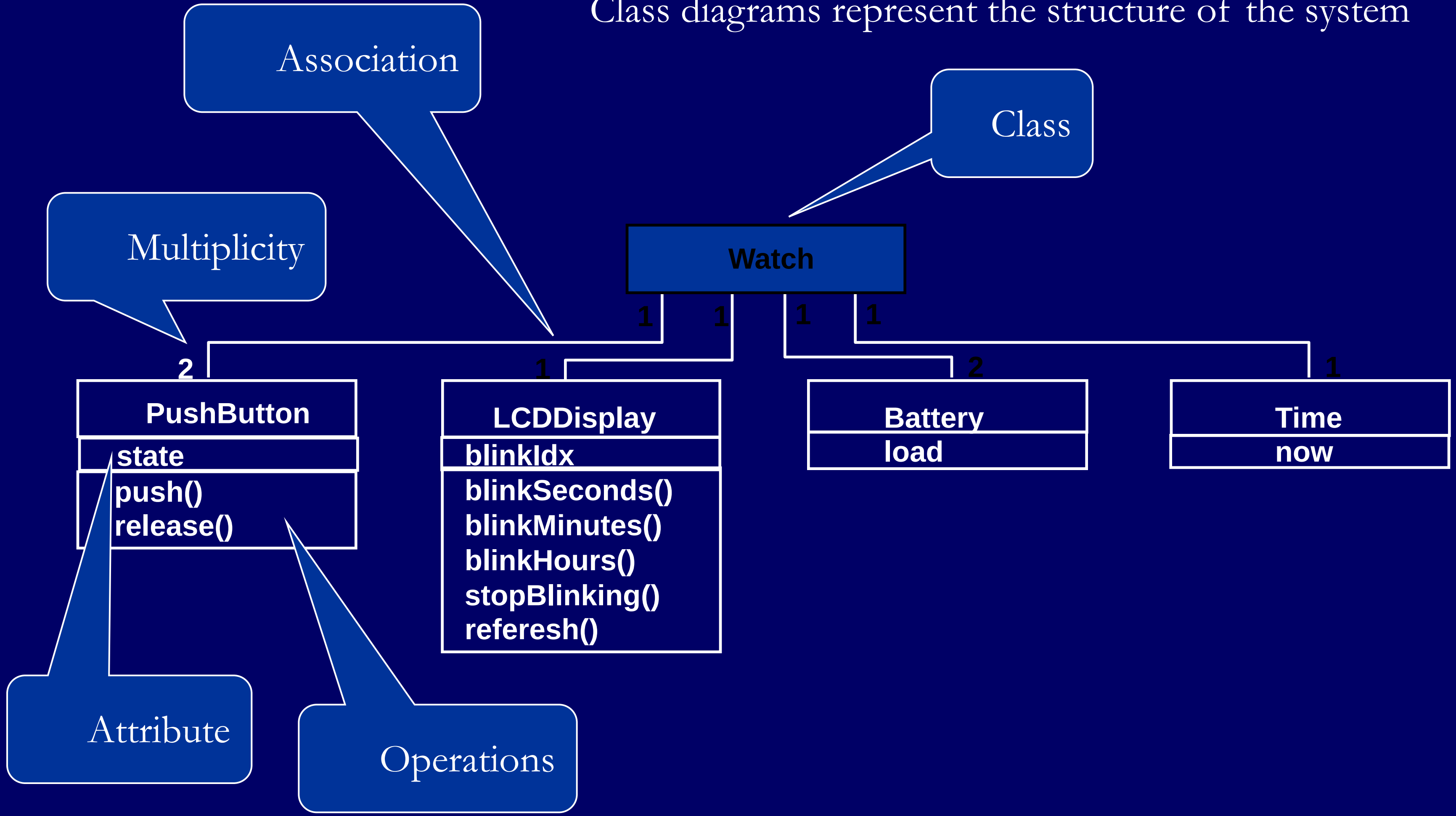
UML first pass: Use case diagrams



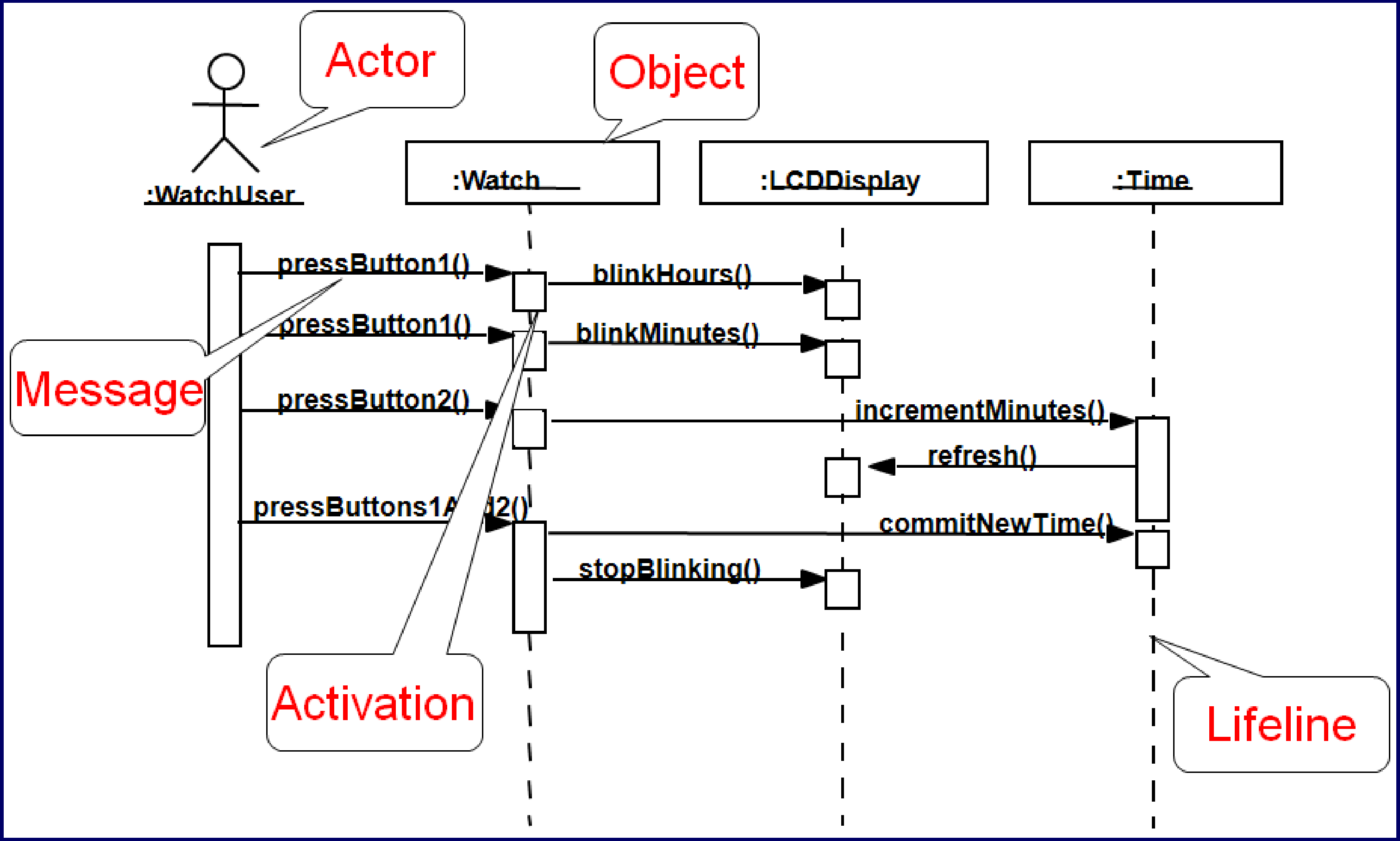
Use case diagrams represent the functionality of the system from user's point of view

UML first pass: Class diagrams

Class diagrams represent the structure of the system

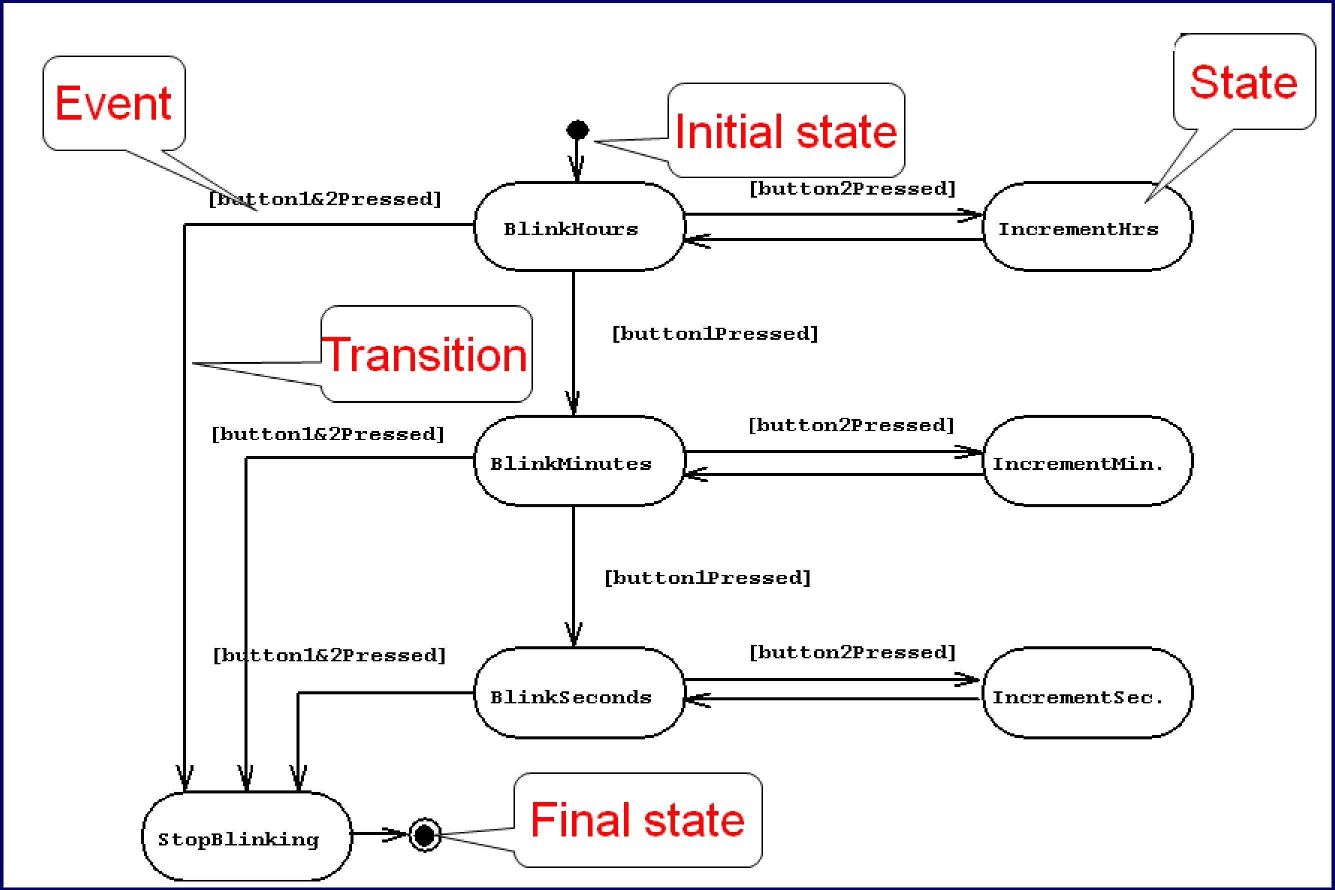


UML first pass: Sequence diagram



Sequence diagrams represent the behavior as interactions

UML first pass: State-chart diagrams for objects with interesting dynamic behavior



Represent behavior as states and transitions

Other UML Notations

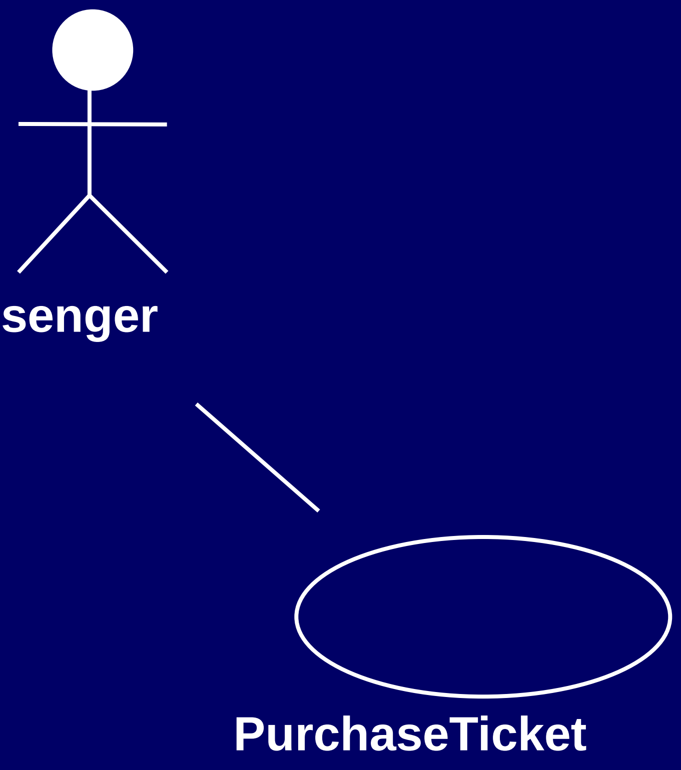
UML provide other notations that we will be introduced in subsequent lectures, as needed.

- Implementation diagrams
  - Component diagrams
  - Deployment diagrams
  - Introduced in lecture on System Design
- Object constraint language
  - Introduced in lecture on Object Design

UML Core Conventions

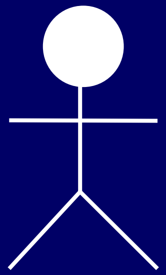
- Rectangles are classes or instances
- Ovals are functions or use cases
- Instances are denoted with an underlined names
  - <u>myWatch:SimpleWatch</u>
  - <u>Joe:Firefighter</u>
- Types are denoted with non underlined names
  - SimpleWatch
  - Firefighter
- Diagrams are graphs
  - Nodes are entities
  - Arcs are relationships between entities

Use Case Diagrams



- Used during requirements elicitation to represent external behavior
- ***Actors*** represent roles, that is, a type of user of the system
- ***Use cases*** represent a sequence of interaction for a type of functionality
- The use case model is the set of all use cases. It is a complete description of the functionality of the system and its environment

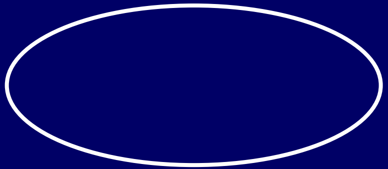
Actors



- An actor models an external entity which communicates with the system:
  - User
  - External system
  - Physical environment
- An actor has a unique name and an optional description.
- Examples:
  - Passenger: A person in the train
  - GPS satellite: Provides the system with GPS coordinates

Use Case

A use case represents a class of functionality provided by the system as an event flow.



A use case consists of:

- Unique name
- Participating actors
- Entry conditions
- Flow of events
- Exit conditions
- Special requirements

Use Case Diagram: Example

Name: Purchase ticket

Participating actor: Passenger

Entry condition:

- Passenger standing in front of ticket distributor.
- Passenger has sufficient money to purchase ticket.

Exit condition:

- Passenger has ticket.

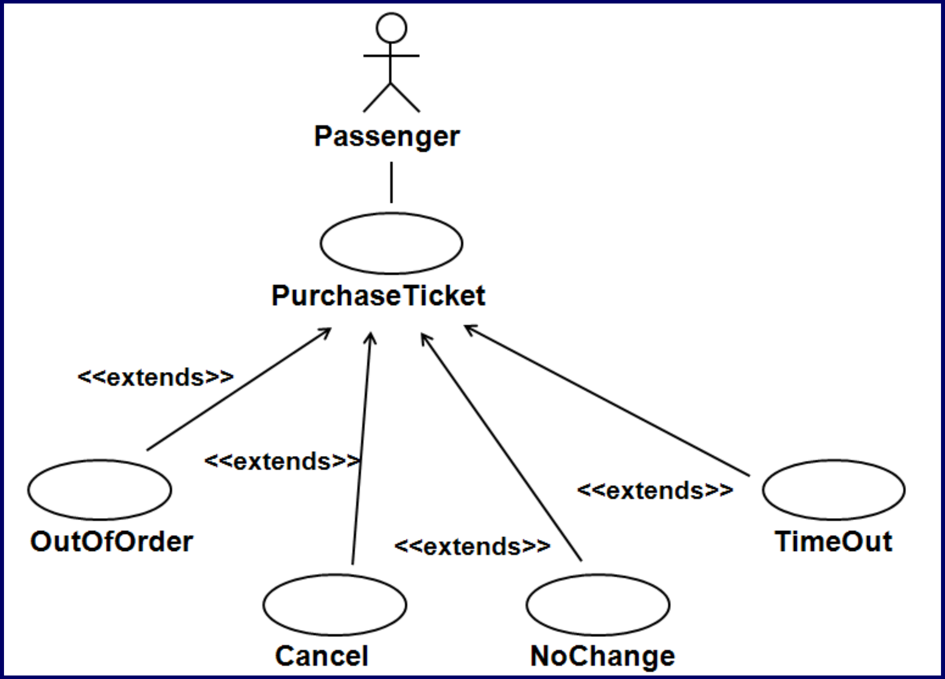
Event flow:

1. Passenger selects the number of zones to be traveled.
2. Distributor displays the amount due.
3. Passenger inserts money, of at least the amount due.
4. Distributor returns change.
5. Distributor issues ticket.

Anything missing?

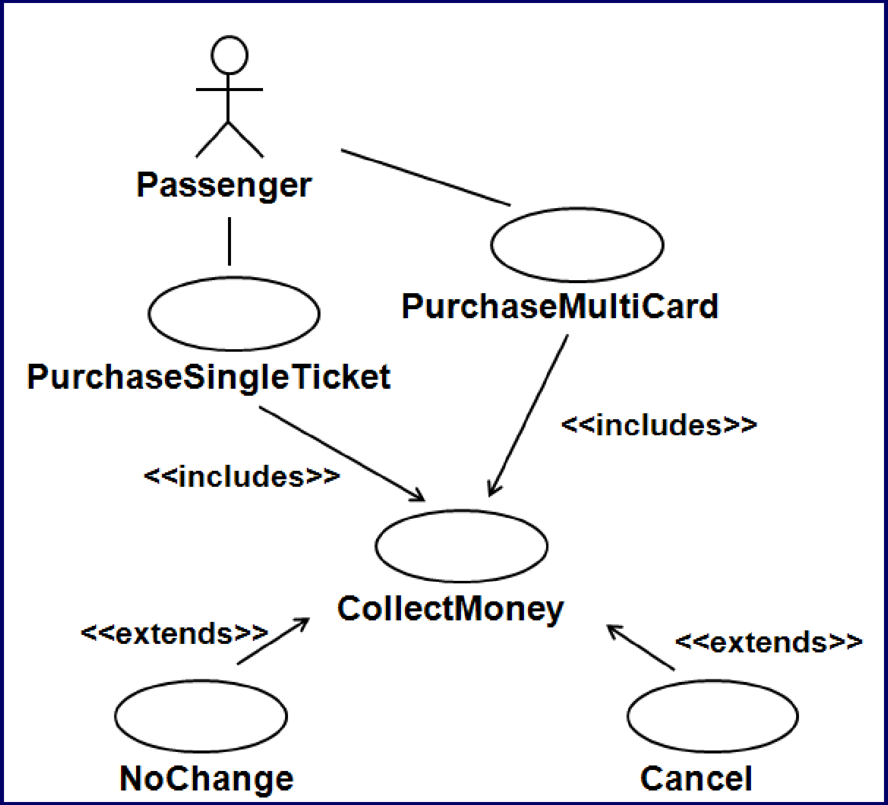
Exceptional cases!

The <<extends>> Relationship



- <<extends>> relationships represent exceptional or seldom invoked cases.
- The exceptional event flows are factored out of the main event flow for clarity.
- Use cases representing exceptional flows can extend more than one use case.
- The direction of a <<extends>> relationship is to the extended use case

The <<includes>> Relationship

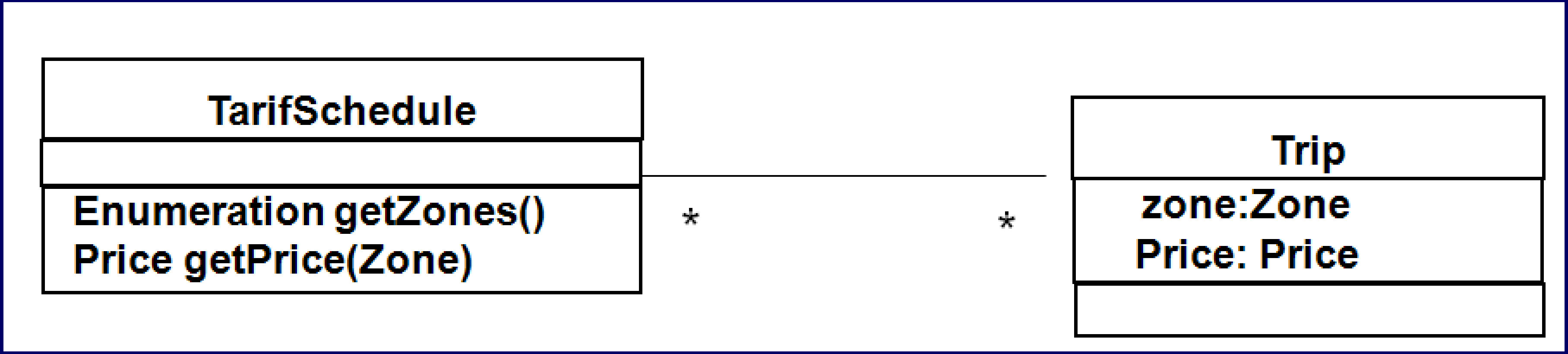


- <<includes>> relationship represents behavior that is factored out of the use case.
- <<includes>> behavior is factored out for reuse, not because it is an exception.
- The direction of a <<includes>> relationship is to the using use case (unlike <<extends>> relationships).

Use Case Diagrams: Summary

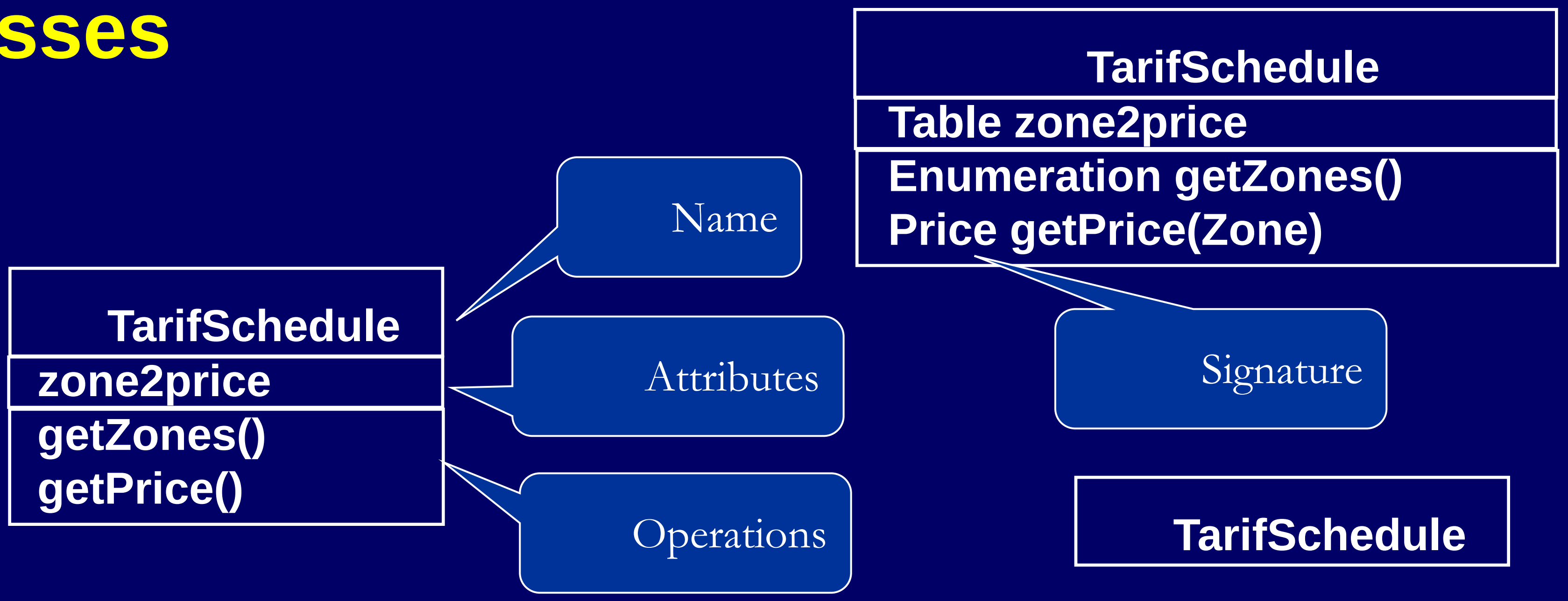
- Use case diagrams represent external behavior
- Use case diagrams are useful as an index into the use cases
- Use case descriptions provide meat of model, not the use case diagrams.
- All use cases need to be described for the model to be useful.

Class Diagrams



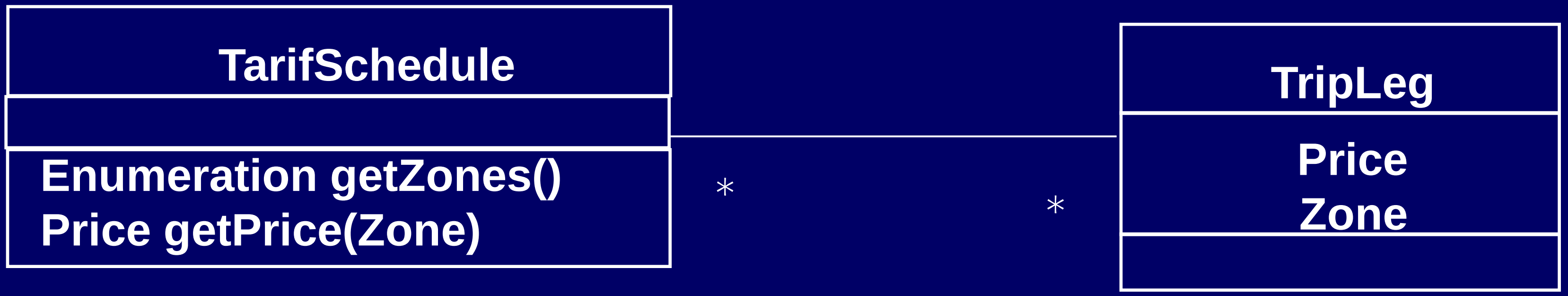
- Class diagrams represent the structure of the system.
- Used
  - during requirements analysis to model problem domain concepts
  - during system design to model subsystems and interfaces
  - during object design to model classes.

Classes



- A ***class*** represent a concept
- A class encapsulates state ***(attributes)*** and behavior ***(operations).***
- Each attribute has a ***type***.
- Each operation has a ***signature***.
- The class name is the only mandatory information.

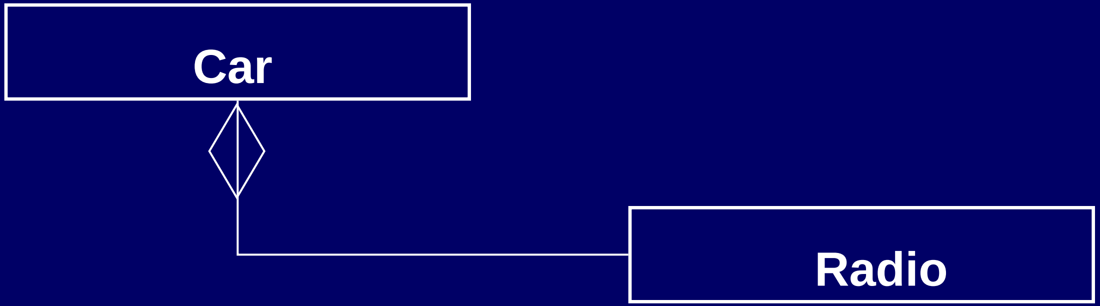
Associations



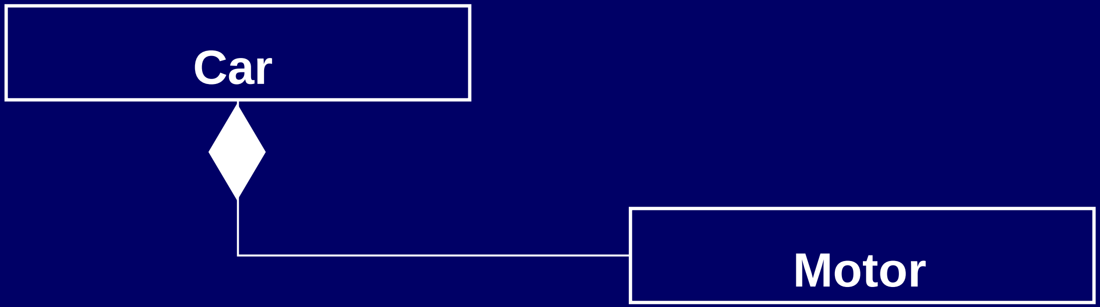
- Associations denote relationships between classes.
- The multiplicity of an association end denotes how many objects the source object can legitimately reference.

UML: Aggregation

- An ***aggregation*** is a special case of association denoting a "consists of" hierarchy.
- The ***aggregate*** is the parent class, the ***components*** are the children class.



- A solid diamond denotes ***composition***, a strong form of aggregation where components cannot exist without the aggregate. (Bill of Material)



How to code a has-relation ?

- X has Y (composition)

```
public class X
{
    Y _yObject;

    public X( )
    {
      _yObject = new Y( );             // creating Y
    }

    public void someMethod( )
   {
     _yObject.doSomething( );
   }

   public void destroy( )
  {
       _yObject.destroy( );
       _yObject= null; // deleting Y from memory
  }

}
```

How to code a uses-relation ?

- X uses Y (aggregation)

```
public class X
{
  Y _yObject;
  public X( )
  {
  }

  public void setY( Y yObject )
  {
     _yObject = yObject;
  }

 public void someMethod( )
 {
     _yObject.doSomething( );
 }

 public void destroy( )
 {
     _yObject = null;// don't delete
 }
}
```

```
public class Y
{
  public Y( )
  {

  }

  public void doSomething( )
  {

  }
}
```

```
public class SomeManager
{
  Y _yObject;
  public SomeManager( )
  {

  }

  public void init( )
  {
     _yObject = new Y( );

       X xObject;
       xObject = new X( );
       xobject.setY( _yObject );
  }

  public void destroy( )
  {
       _yObject.destroy( );
  }
}
```

How to code a uses-relation ?

- X uses Y (aggregation)

```
public class X
{
   SomeManager _someManager;

   public X( SomeManager someManager )
   {
      _someManager = someManager;
   }

   public void someMethod( )
   {
     yObject = someManager.getY( );
     // use yObject...
   }

}
```

```
public class SomeManager
{
   Y _yObject;

   public SomeManager( )
   {
       _yObject = new Y( );
   }

   public void getY( )
   {
       return _yObject;
   }

   public void destroy( )
   {
       _yObject.destroy( );
   }

}
```

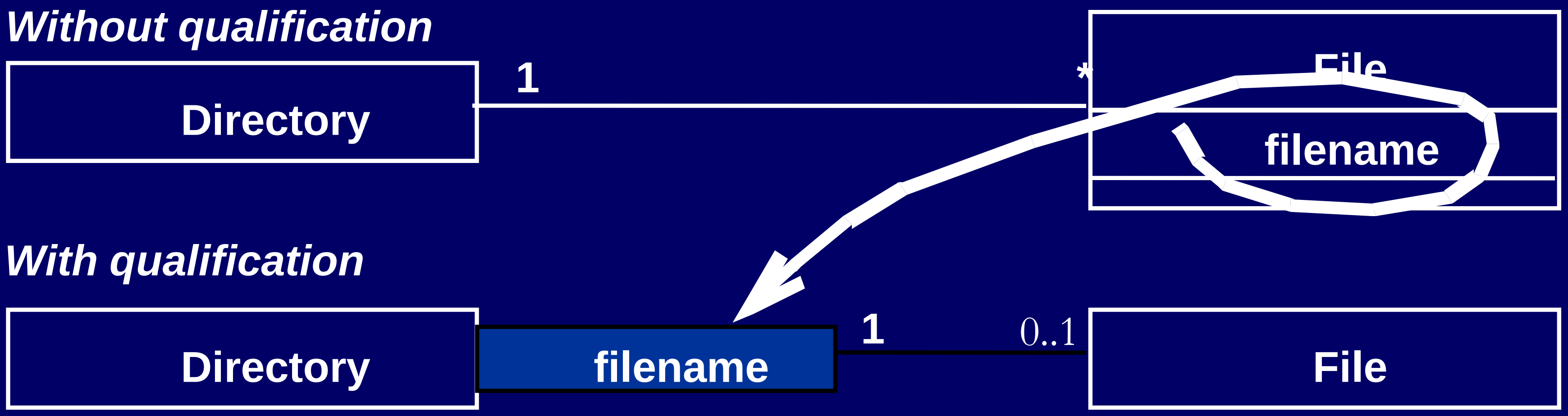
How to code a uses-relation ?

- X uses Y (aggregation)

```
public class X
{


  public X( )
  {

  }

  public void someMethod( )
  {
    yObject = Y.getInstance( );
    // use yObject...
  }

}
```

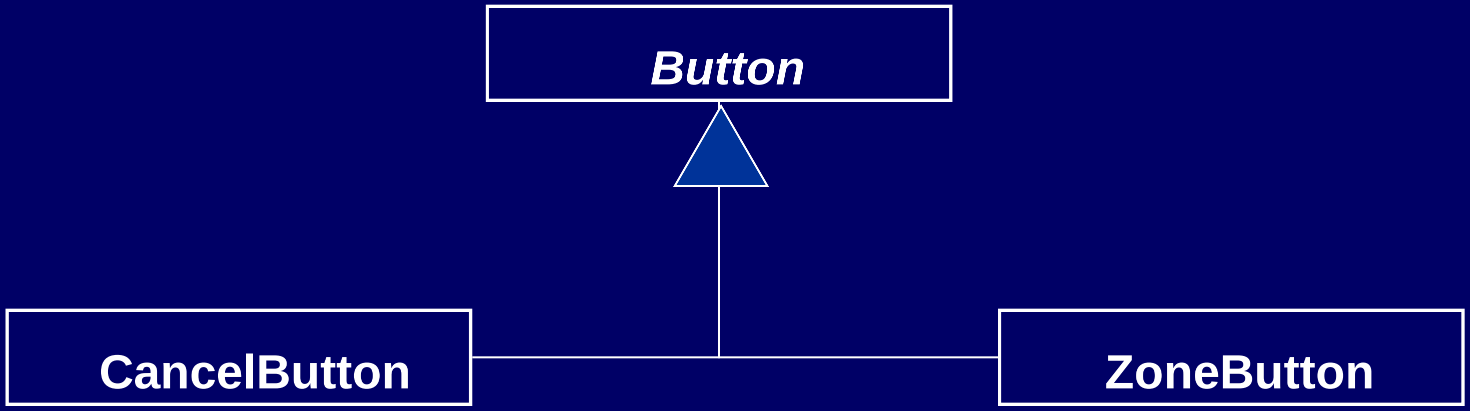
```
public class Y
{
   protected static Y thisY;

   public static Y getInstance( )
   {
     if( thisY == null )
      thisY = new Y( );
     return thisY;
   }
}
```

Qualifiers



- Qualifiers can be used to reduce the multiplicity of an association.

Inheritance



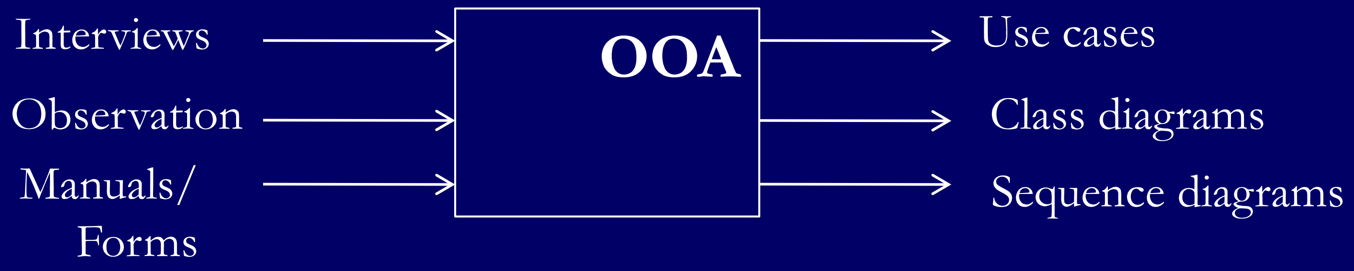
- The **children classes** inherit the attributes and operations of the **parent class**.
- Inheritance simplifies the model by eliminating redundancy.

Object Oriented Analysis & Design

Object Oriented Analysis & Design

- OOA:
  - Input: written requirements statement, a formal vision document, interviews with stakeholders or other interested parties
  - Target: produce a conceptual model of the information that exists in the area being analyzed
  - Output: set of use cases, one or more UML class diagrams, and a number of interaction diagrams. It may also include some kind of user interface mock-up

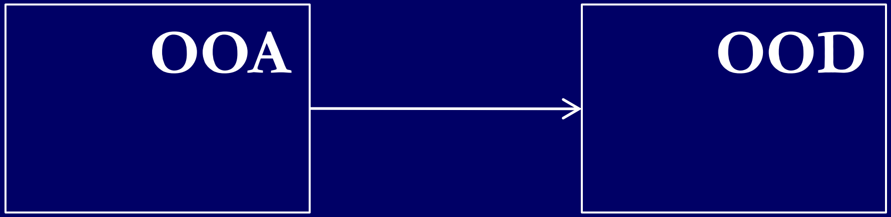
Object Oriented Analysis: creating a class diagram



- Identify classes
- Identify relations between classes

Object Oriented Analysis & Design

- OOD:
  - Input: conceptual model produced in OOA
  - Target: a model of the solution domain, *how* the system is to be built, given the constraints identified in the OOA
  - Output: a model of the solution domain, *how* the system is to be built.., specification of implementation classes and interfaces



How to identify a class ?

- nouns → class
- verbs → methods
- adjectives → attribute/member-variables

Noun/Verb/Adjective example

*The **circuit controller** shall support **digital and analog circuits**. The circuit controller shall contain 32 **DSPs**. When the circuit controller receives a request to setup a circuit, it shall allocate a DSP to the circuit.*

- We discover the following classes from the requirement description
  - CircuitController
  - DigitalCircuit
  - AnalogCircuit
  - DSP
  - ?
- What functions can you discover?
- What is implicit here?
  -

How to identify relations ?

- Owns/has it?
- Uses it?
- Is type of ?

Analysis Example