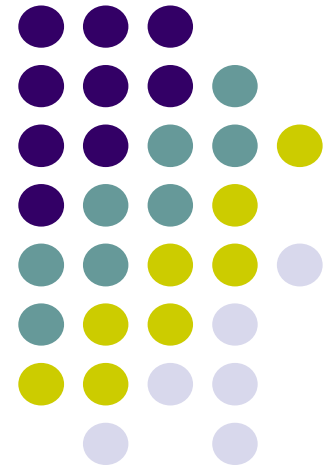


Object-Oriented Modeling and Software

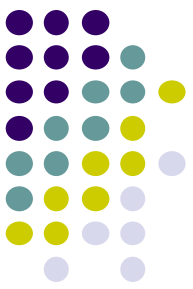
Introduction to modeling
Visual modeling
OO modeling
Software complexity
Software architecture



Model



- *Model*: textual or graphical description of a system or process (existing or such to be created);
- In ENCYCLOPEDIA BRITANICA (<https://www.britannica.com/science/scientific-modeling>):
 - *Scientific models* are used to explain and predict the behavior of real objects or systems and are used in a variety of scientific disciplines, ranging from physics and chemistry to ecology and the Earth sciences.
 - *Scientific models* at best are approximations of the objects and systems that they represent—they are not exact replicas.



The Role of a Model

- Models in human life and engineering activities
- History of modeling – drawings describing static character of an object/a system, and its behavior (dynamics character)
- Role of modeling at:
 - Planning of activities
 - Project evaluation – as time and money
 - Work load distribution and allocation of resources

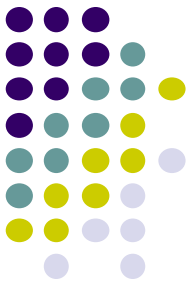


Early humans brainstorming on a proto-whiteboard (by Tom Gullion)



Model and Modeling

- **Model:** (mathematical) presentation of structure and processes of a given system (used for analysis and planning)
- **Modeling:** process of describing of the system by means of its model (physical, conceptual, mathematical or based on imitation) and simulation of system activities by means of applying the model on a data set. Models represent real phenomenon that are difficult to observe directly.



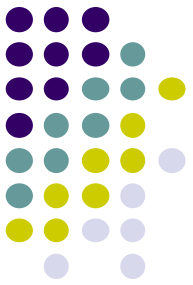
Modeling

- **Phases in model development**
- **Iterations in model development**
- **Requirements to the end product – about functionality, external design, performance and reliability**
- **Models are not final – usually, they change during the project time in order to reflect new approaches and experiences**



Types of Models

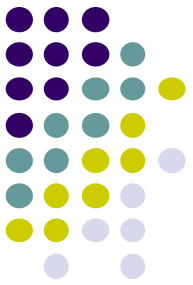
- **Static models – describe system structure, e.g. E/R data model. Data types – analogues, discrete, or hybrid data.**
- **Dynamic models – describe system behavior. They depending on the chosen apparatus and can be:**
 - **Analytical – describe by differential equations**
 - **Imitation models – usually, they are built by a visual language such as Petri nets, OMT, OBLOG, UML, etc. We cannot describe all the system details visually. Thus, we have to look for *a balance between visual and textual description.***



Appliance of Models

Practical models should be:

- **describing the system in a correct way**
- **consistent – different views should not describe things being in conflict each other**
- **easy to be explained to and understood by other people – *as simple as possible but not simplified***
- **easy for updates and maintenance**
- **in a form suitable for transfer to other people**



Visual Modeling

Visual modeling:

- a way of thinking about problems using practical and graphical models derived by ideas of the real world
- graphical modeling presenting treated system in two- and three dimensional way.

Visual modeling language – manipulates visual information, i.e. presents systems through graphical (icon-based) and textual expressions according given dimensional grammar¹. Visual languages for specification and for programming.

¹ Microsoft Visual Basic and Visual C++ are not visual languages



Visual Models

Graphical abstractions describe the essence of a complex problem or of a structure by filtering immaterial details. In such a way, they do the problem easier for understanding and presentation. We can compare them with the architectural approach – drawn models.

When building a complex system, developers should do:

- **Extract different views about the system**
- **Build models using precise symbols (notations)**
- **Verify/validate that the models satisfy system requirements**
- **Add details in order to transform models into implementation, step by step**



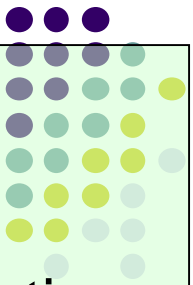
Modeling Method

A method is a generic guideline for realization of modeling; contains specific knowledge for various cases such as patterns and conventions.

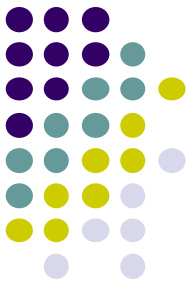
A ***method*** is a mixed bag of guidelines and rules, including the following components (*Dr. James Roumbaugh, 1995*):

- ***modeling concepts***
- ***views and notations***
- ***development process***
- **hints and rules-of-thumb**

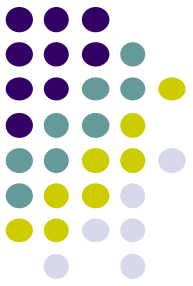
More precisely, a **method** has to include:



- A set of fundamental ***modeling concepts*** for capturing semantic knowledge about a problem and its solution. The modeling concepts are independent of how they are visualized. They are the inputs for semantic tools, such as code generators, semantic checkers, and traceability tools.
- A set of ***views and notations*** for presenting the underlying modeling information to human beings which allow them to examine and modify it. Normally the views are graphic, but multimedia interfaces are feasible with current technology. Graphic views use geometric arrangement and graphic markers to highlight portions of the semantic information. Usually each view shows only a part of the entire semantic model.



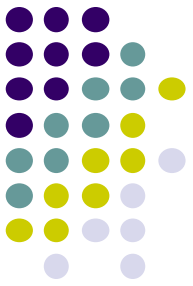
- A step-by-step iterative ***development process*** for constructing models and implementations of them. The process may be described at various levels of detail, from the overall project management down to the specific steps to build low-level models. The process describes which models to construct and how to construct them. It may also specify measures of goodness for evaluating proposed designs.
- A collection of **hints and rules-of-thumb** for performing development. These are not organized into steps. They may be applied wherever they make sense. The concept of ***patterns*** is an attempt to describe case-based experience in a uniform way. Patterns represent specific design solutions to recurring problems. These may apply at various levels of detail, from large-scale architecture down to low-level data structures and algorithms.



Why do we model?

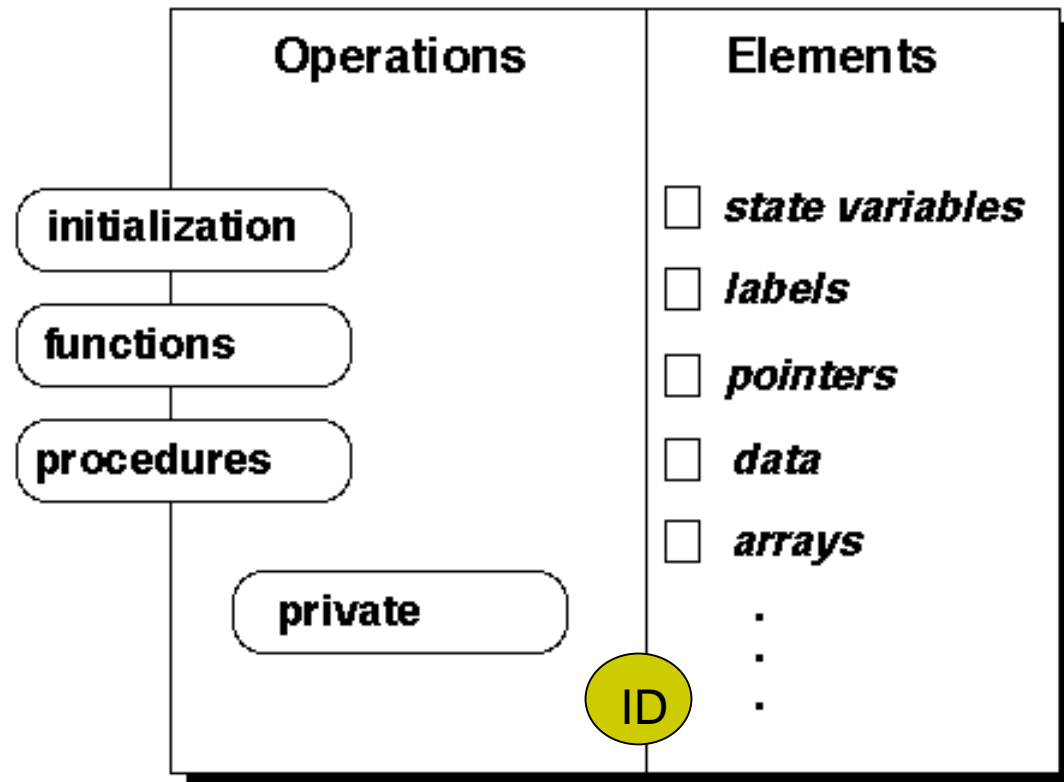
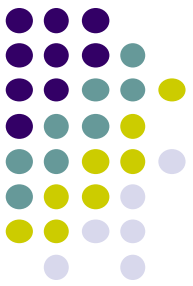
- **Models help in organizing, visualizing, understanding and creation of complex information systems.**
 - **Two challenges for software developers:**
 - **Business environment is highly concurrent and dynamic**
 - **Increasing system complexity**
- Models help meeting them.**

Object-Oriented Programming and Modelling

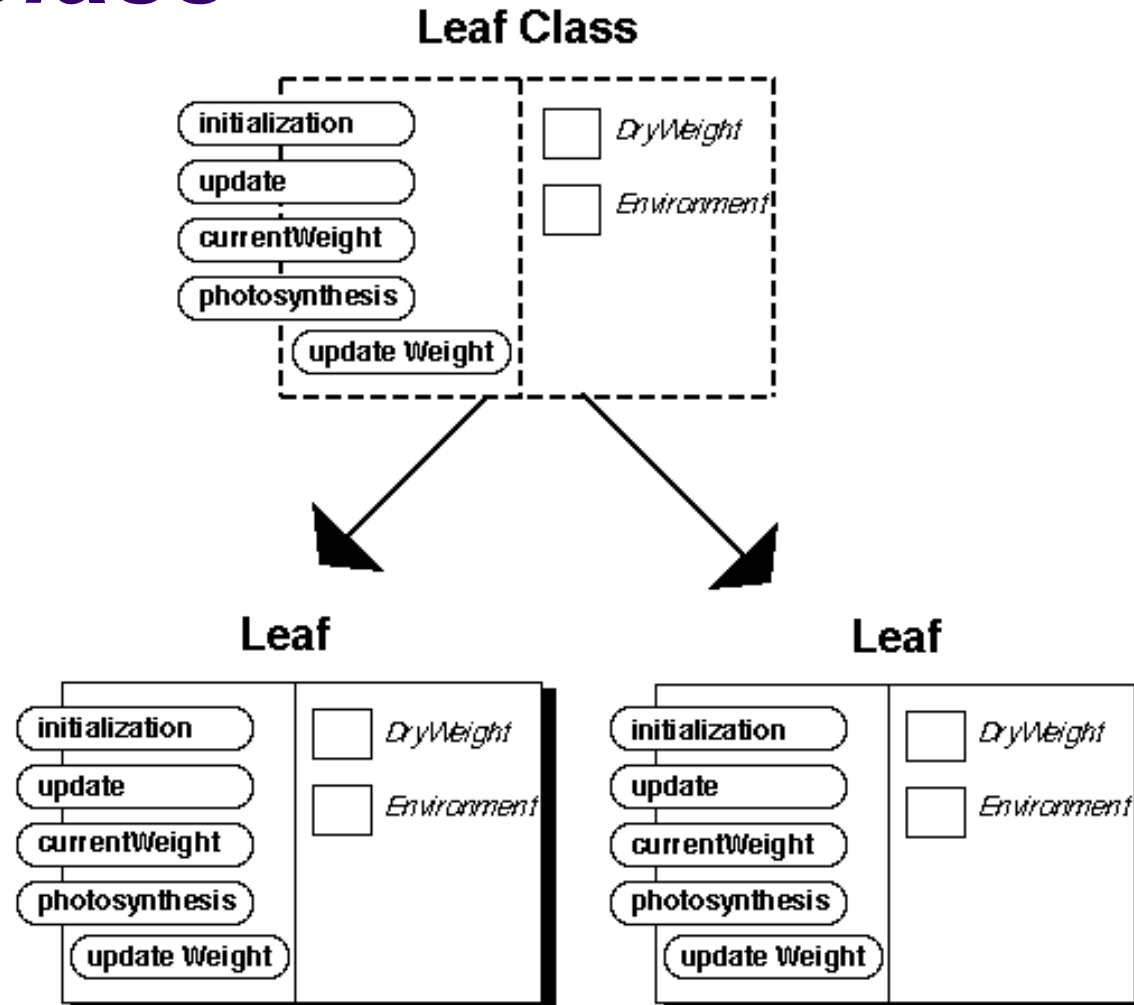


- **Language Level Definition**
 - Class
 - Object
 - Inheritance
- **Conceptual Level Definition**
 - Abstraction
 - Delegation
 - Encapsulation
 - Information Hiding
 - Hierarchy

Language Level Definition of OOP : *Object*

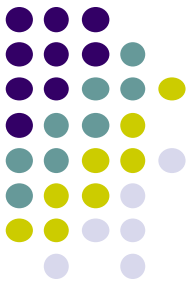


Language Level Definition of OOP : *Class*



The God Class Problem

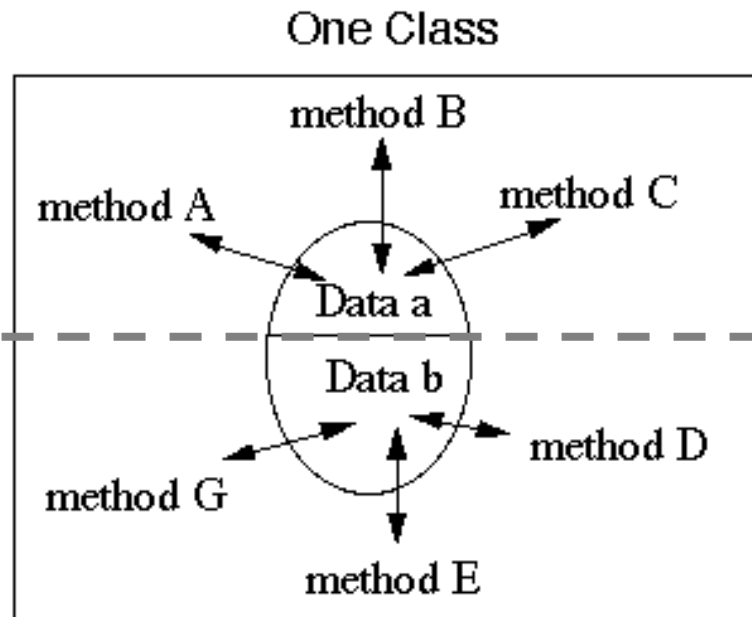
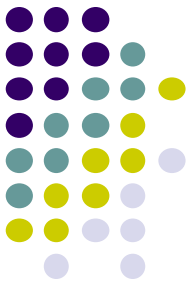
[Roger Whitney] (1/2)



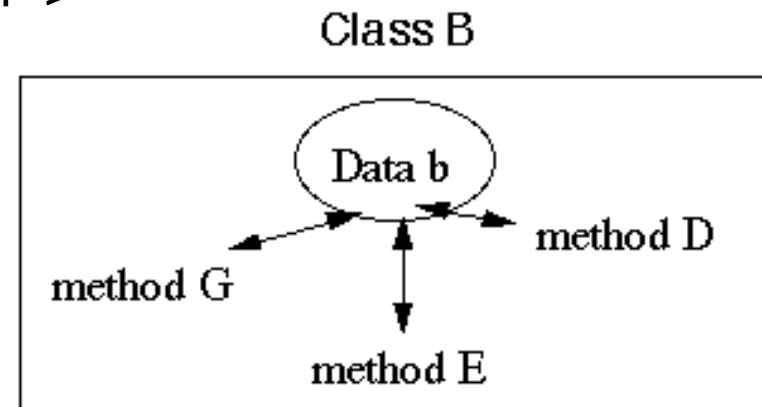
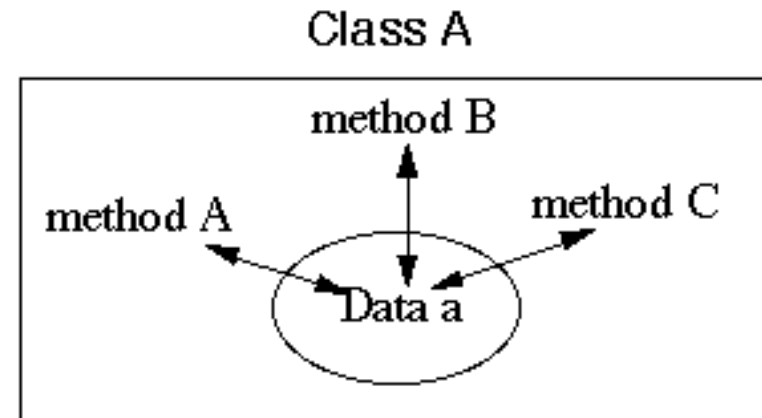
- Distribute system intelligence horizontally as uniformly as possible, that is, the top-level classes in a design should share the work uniformly.
- Do not create god classes/objects in your system. Be very suspicious of a class whose name contains **Driver**, **Manager**, **System**, or **Subsystem**
- Beware of classes that have too much non-communicating behavior, that is, methods that operate on a subset of the data members of a class. God classes often exhibit much non-communicating behavior.

The God Class Problem

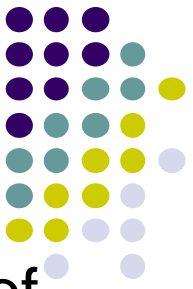
[Roger Whitney] (2/2)



solution ->



Conceptual Level Definition of OOP: *Abstraction*



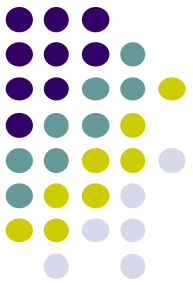
- "Extracting the essential details about an item or group of items, while ignoring the unessential details."
Edward Berard
- "The process of identifying common patterns that have systematic variations; an abstraction represents the common pattern and provides a means for specifying which variation to use."
Richard Gabriel

Example

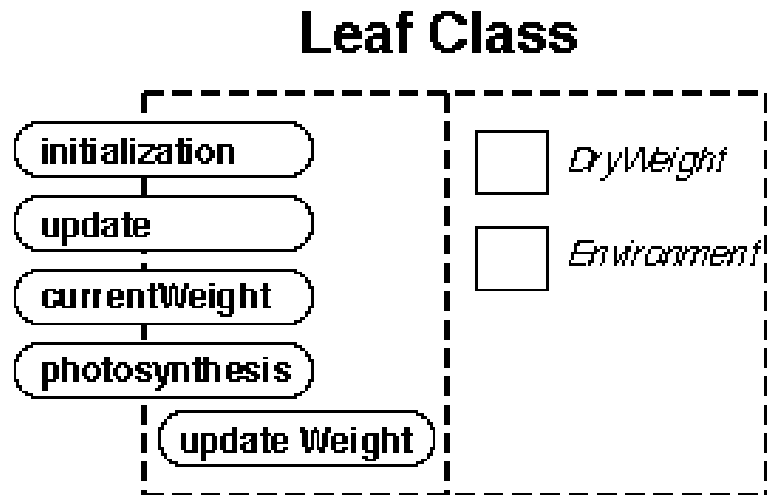
- Pattern: Priority queue
- Essential Details:
 - length
 - items in queue
 - operations to add/remove/find item

- Variation: link list vs. array implementation; stack, queue

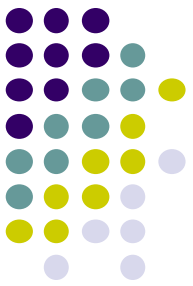
Conceptual Level Definition of OOP: *Encapsulation, Information Hiding*



- ***Encapsulation*** - Enclosing all parts of an abstraction within a *class container*
- ***Information Hiding*** - Hiding parts of the abstraction within an *object*
- **Example**

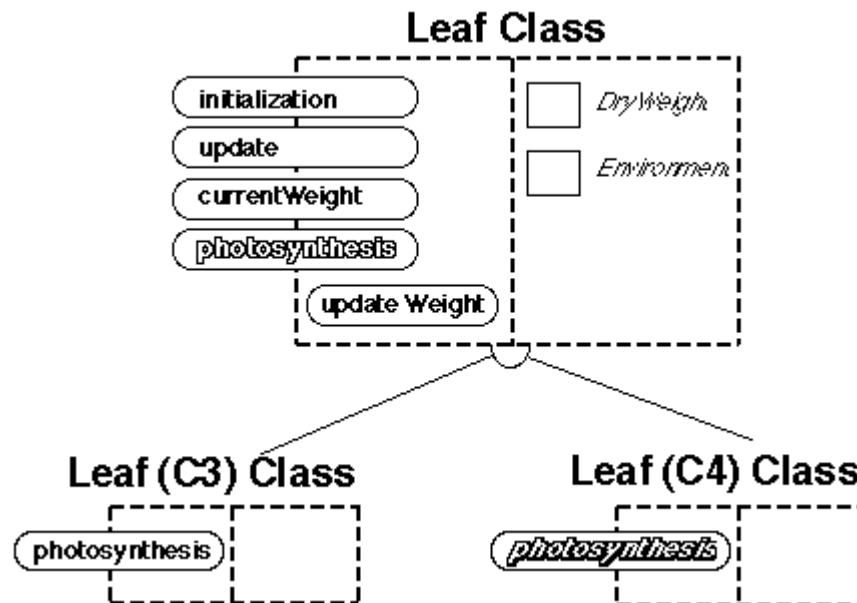


Conceptual Level Definition of OOP: *Hierarchy*

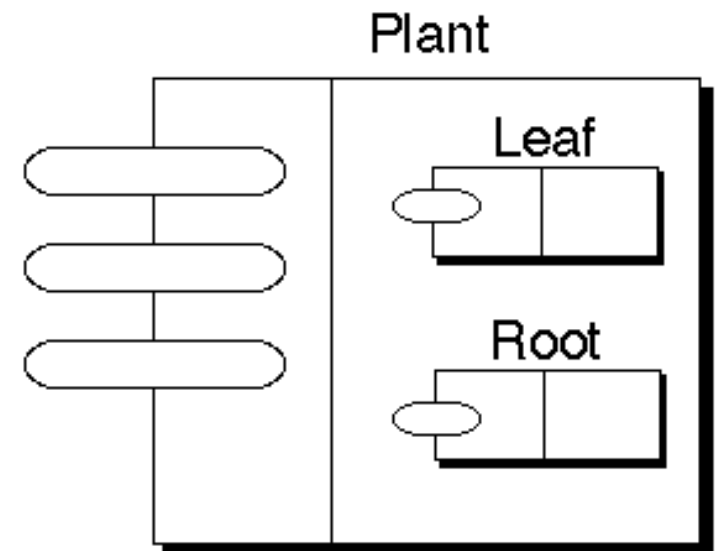


- *Abstractions arranged in order of rank or level*

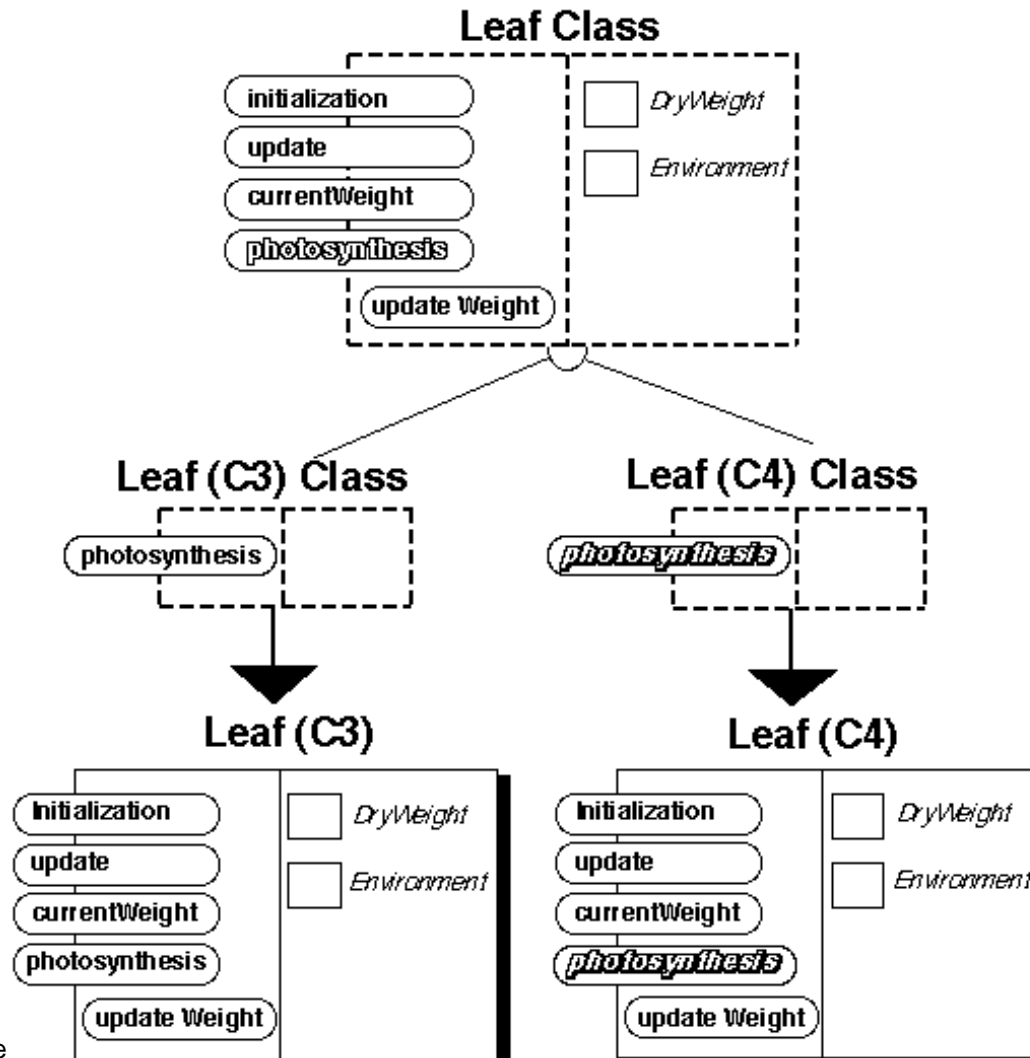
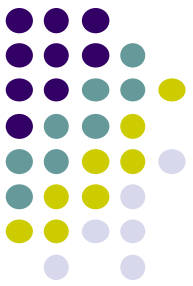
Class Hierarchy



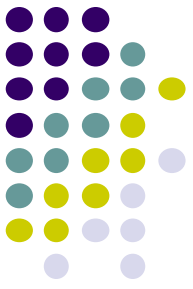
Object Hierarchy



Language Level Definition of OOP : *Inheritance*



Sub-typing vs. implementation inheritance



We can distinguish two broad classes of inheritance:

- **Sub-typing**

- A logical classification of B as a subtype of A. Also called *interface inheritance*. The is-a argument is straightforward. A is usually an abstract class (or an interface in Java). All (or almost all) the operations of A are applicable to B. We expect to use Bs in mixed collections of As etc. We may often inherit some implementation at the same time.

- **Implementation inheritance**

- Using the implementation of A for convenience in B. The is-a argument may take some ingenuity. A is often concrete. Only some the operations of the operations of A are applicable to B. We probably won't use Bs in place of As.

- **Sub-typing is almost always OK, provided the classification is valid. Implementation inheritance is controversial.**

Method Overriding vs. Overloading



```
class Dog{
    public void bark(){
        System.out.println("woof ");
    }
}

class Hound extends Dog{
    public void sniff(){
        System.out.println("sniff ");
    }

    public void bark(){
        System.out.println("bowl");
    }
}
```

Same Method Name,
Same parameter

Overriding

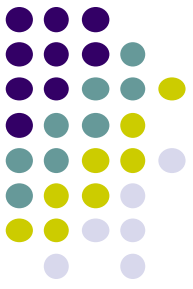
```
class Dog{
    public void bark(){
        System.out.println("woof ");
    }

    //overloading method
    public void bark(int num){
        for(int i=0; i<num; i++){
            System.out.println("woof ");
        }
    }
}
```

Same Method Name,
Different Parameter

Overloading

Downcasting




```
class Animal {  
    void makeNoise() {System.out.println("generic noise"); }  
}  
class Dog extends Animal {  
    void makeNoise() {System.out.println("bark"); }  
    void playDead() { System.out.println(" roll over"); }  
}  
class CastTest2 {  
    public static void main(String [] args) {  
        Animal [] a = {new Animal(), new Dog(), new Animal() };  
        for(Animal animal : a) {  
            animal.makeNoise();  
            if(animal instanceof Dog) {  
                Dog d = (Dog) animal; //need casting to the ref. var.!  
                d.playDead(); // try to do a Dog behavior  
            }  
        }  
    }  
}
```

OK, because
animal instanceof Dog

We convert down the
inheritance tree to a
more specific class

Don't just rely on the compiler!



```
class Animal { }  
class Dog extends Animal { }  
class DogTest {  
    public static void main(String [] args) {  
        Animal animal = new Animal();  
        Dog d = (Dog) animal; // compiles but fails later  
                                // with run-time java.lang.ClassCastException  
         String s = (String) animal; // cannot compile!  
                                // animal can't EVER be a String  
    }  
}
```

Downcasting suggests that later a more specific method may be called

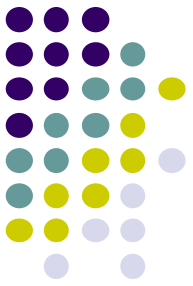
Upcasting



- Unlike **downcasting**, upcasting works implicitly
- With **upcast**, we implicitly limit the number of possible methods, whereas with downcast we may later call a more specific method:

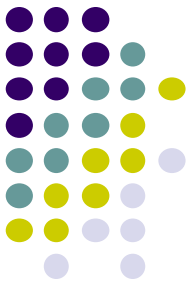
```
class Animal { }  
class Dog extends Animal { }  
    class DogTest {  
        public static void main(String [] args) {  
            Dog d = new Dog();  
            Animal a1 = d; // upcast ok with no explicit cast  
            Animal a2 = (Animal) d; // upcast ok with an explicit cast  
        }  
    }
```

Conceptual Level Definition of OOP: *Delegation*



Three ways of reusing classes:

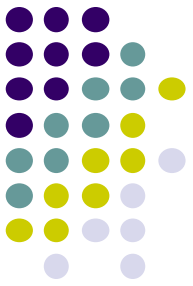
- Generalize A, usually by *parameterization*, so that it does the what you want as well as what it originally did.
- *Create a new class B which **delegates** operations to A, usually by having an object of class A as an instance variable and calling methods on that object. B is then a client of A.*
- Create a new class B which *inherits* from A. B is then a *subclass* of A.
- Open discussion – *pro's and con's?*



The OO Design and Coupling

- **Decomposable system** - One or more of the components of a system have no interactions or other interrelationships with any of the other components at the same level of abstraction within the system
- **Design Goal** - The interaction or other interrelationship between any two components at the same level of abstraction within the system be as weak as possible

Measure of the modular interdependence



- Unnecessary object coupling:
 - needlessly decreases the **reusability** of the coupled objects
 - increases the chances of **system corruption** when changes are made to one or more of the coupled objects

Types of Modular Coupling

In order of desirability

Cure:
Decompose
the operation
into multiple
primitive
operations

- Data Coupling (weakest most desirable) - output from one module is the input to another
- Control Coupling - passing control flags between modules so that one module controls the sequencing of the processing steps in another module.
- Global Data Coupling - two or more modules share the same global data structures
- Internal Data Coupling (strongest least desirable) - one module directly modifies local data of another module (like C++ Friends)
- Content Coupling (unrated) - some or all of the contents of one module are included in the contents of another (like C/C++ header files)

Cohesion

- "Cohesion is the degree to which the tasks performed by a single module are functionally related." *IEEE, 1983*
- "A software component is said to exhibit a high degree of cohesion if the elements in that unit exhibit a high degree of functional relatedness. This means that each element in the program unit should be essential for that unit to achieve its purpose." *Sommerville, 1989*
- Types of Module Cohesion
 - Coincidental (worst)
 - Logical
 - Temporal
 - Procedural
 - Communication
 - Sequential
 - Functional (best)



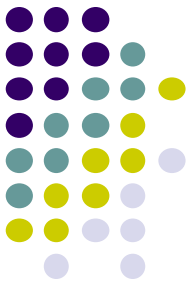


Coincidental Module Cohesion

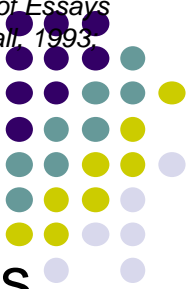
- Little or no constructive relationship among the elements of the module
- Common Object Occurrence:
 - Object does not represent any single object-oriented concept
 - Collection of commonly used source code as a class inherited via multiple inheritance
- Example:

```
class Rous {  
    public static int findPattern( String text, String pattern) {  
        // blah  
    }  
    public static int average( Vector numbers ) {  
        // blah  
    }  
    public static OutputStream openFile( String fileName ) {  
        // blah  
    }  
}
```

Coincidental Module Cohesion



Умало едно време ...
Орел, Рак и Щука.
Нищо не направил,
а цял живот се греел
на лъчите на славата.



Logical Module Cohesion

- Module performs a set of related functions, one of which is selected via function parameter when calling the module
- Similar to control coupling
- **Cure:** isolate each function into separate operations

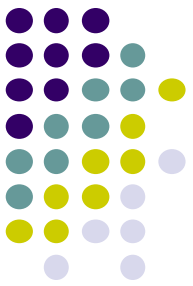
```
public void sample( int flag ) {  
    switch ( flag ) {  
        case ON:  
            // bunch of on stuff  
            break;  
        case OFF:  
            // bunch of off stuff  
            break;  
        case CLOSE:  
            // bunch of close stuff  
            break;
```

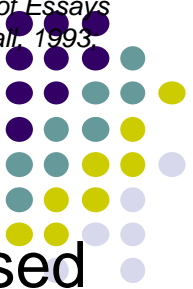
```
}
```

Logical Module Cohesion

The module performs a set of related functions;
when calling the module we choose one by passing a parameter ...

How long?





Temporal Module Cohesion

- Elements are grouped into a module as they are all processed within the same limited time period
- Common example:
 - "Initialization" modules that provide default values for objects
 - "End of Job" modules that clean up

```
procedure initializeData() {  
    font = "times"; windowSize = "200,400";  
    foo.name = "Not Set"; foo.size = 12;  
    foo.location = "/usr/local/lib/java";  
}
```

- **Cure:** Each object should have a constructor and destructor

```
class foo {  
    public foo() {  
        foo.name = "Not Set";  
        foo.size = 12;  
        foo.location = "/usr/local/lib/java";  
    }  
}
```

*Call these constructors/
destructors from a non-object
oriented routine that performs a
single, cohesive task*

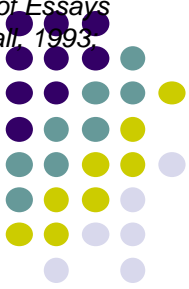
Temporal Module Cohesion



Individual Destructors



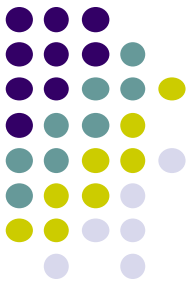
"End of Job" modules for many objects



Procedural Module Cohesion

- Associates processing elements on the basis of their procedural or algorithmic relationships
- Procedural modules are application specific
- In context the module seems reasonable
- Removed from the context these modules seem strange and very hard to understand
- *BUT:*
 - Cannot understand module without understanding the program and the conditions existing when module is called
 - Makes module hard to modify and understand
- **Cure:** redesign the system
- **Class Builder** verse **Program writer**
- If a module is necessary, remove it from objects

Procedural Module Cohesion



- The module cannot be understood without understanding the program and the existing conditions when calling this module
- Makes the module difficult to change and understand

Solution: Make a total redesign of the system!

Communication

Module Cohesion

Source: 1) *Object Coupling and Object Cohesion*, chapter 7 of *Essays on Object-Oriented Software Engineering*, Vol 1, Berard, Prentice-Hall, 1993.
2) SDSU & Roger Whitney;



- Operations of a module all operate upon the same input data set
and/or
produce the same output data

Cure:

- Isolate each element into a separate modules
- Rarely occurs in object-oriented systems due to polymorphism

Communication Module Cohesion



- Module operations operate on the same input data



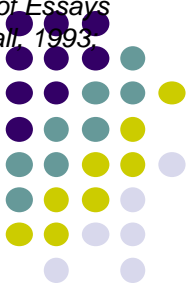
Solution:

- Isolate each element in separate modules

Sequential (Pipeline) Module Cohesion

Source:

1) Object Coupling and Object Cohesion, chapter 7 of Essays
on Object-Oriented Software Engineering, Vol 1, Berard, Prentice-Hall, 1993;
2) SDSU & Roger Whitney;

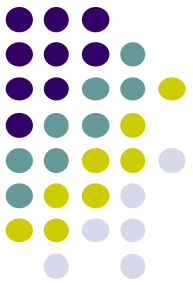


- Sequential association the type in which the output data from one processing element serve as input data for the next processing element
- A module that performs multiple sequential functions where the sequential relationship among all of the functions is implied by the problems or application statement and where there is a data relationship among all of the functions

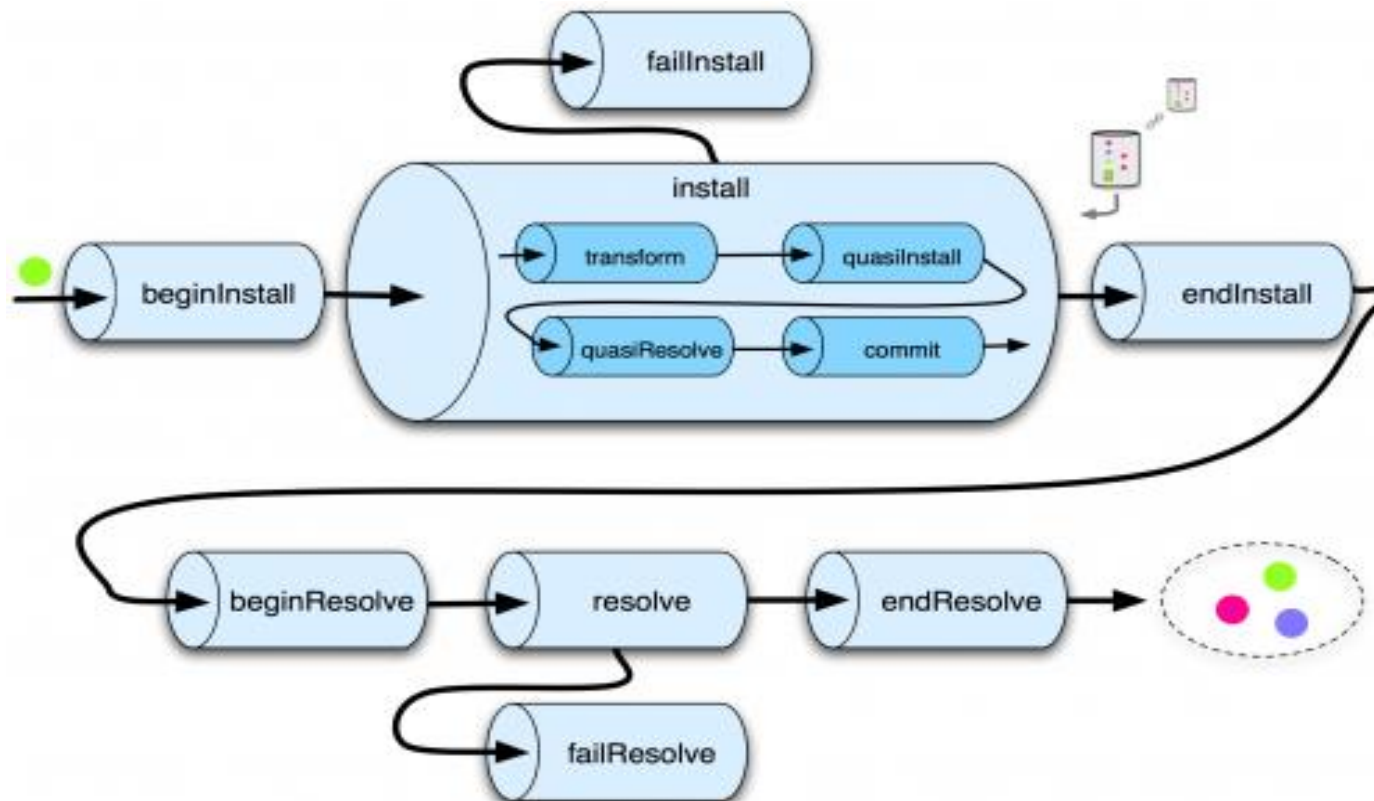
Cure:

- Decompose into smaller modules

Sequential (Pipeline) Module Cohesion



Solution:
Decompose into smaller modules





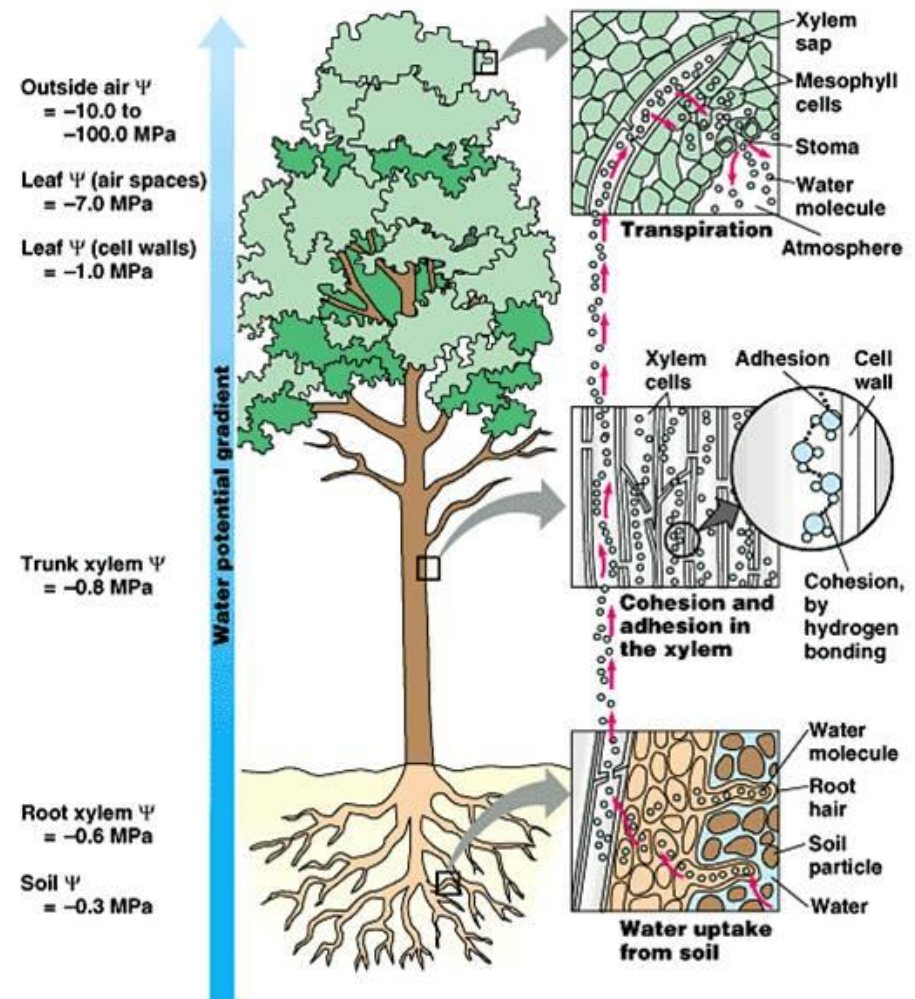
Functional Module Cohesion

- If the operations of a module can be collectively described as a single specific function in a coherent way, the module has functional cohesion
- If not, the module has lower type of cohesion
- In an object-oriented system:
 - Each operation in public interface of an object should be functional cohesive
 - Each object should represent a single cohesive concept

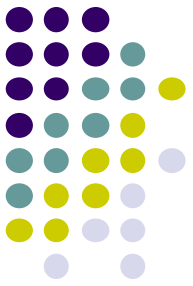
Functional Module Cohesion



- If module operations can be described as one specific function in a coherent way ...
- In an OO system:
 - Every operation in the public interface of an object must be functionally cohesive, cohesive
 - Each object must represent a cohesive concept



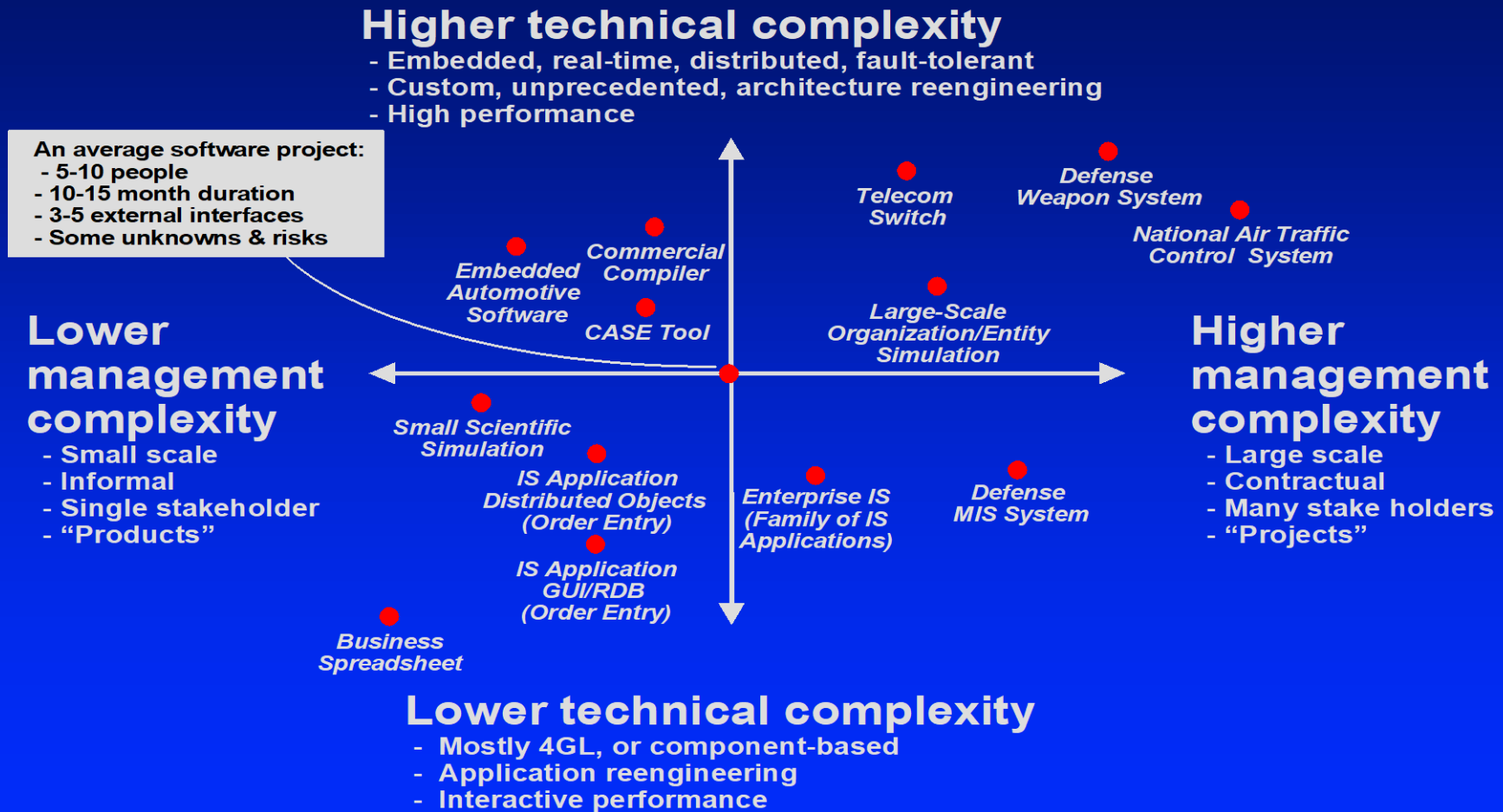
Conceptual diagrams help software development

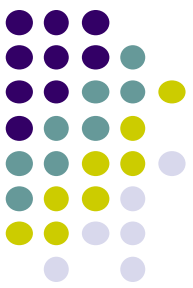


By OO modeling, we can represent visually using industry-standard diagrams many issues of software development:

- Overall architecture of the system
- Business requirements
- System boundaries and dependencies
- System complexity
- Flow of information through a system
- Data model organization and structure

Dimensions of software complexity





Software Architecture

Complexity determines the software architecture – a set of important decisions about software system organization.

- Choice of structural elements and interfaces for system composition
- Behavior – determined by elements' collaboration
- Composition of the chosen structural elements and their behavior into a larger system
- Architectural style

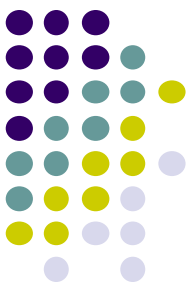


Software Architecture – issues

Moreover, the software architecture determines:

1. Usability
2. Functionality
3. Performance
4. Flexibility
5. Reusability
6. Portability
7. ...

Economical and technical restrictions and their balance – “*good, fast, cheap – choose only two!*”



Architectural Style

Defines a family of systems by means of pattern for structural organization. In other words, it defines:

- **Component dictionary and types of connecting elements**
- **Set of restrictions and how we can combine them**
- **One or more semantic models specifying how to determine common system properties based on the properties of its building blocks.**

Representing System Architecture

