# UML Notation

Jim Fawcett

CSE681 – Software Modeling & Analysis

Fall 2013

# Table of Contents

# Notation Sources & Notes

- These notes describe the Universal Modeling language (UML) as described in:

  - UML Distilled, Martin Fowler, Addison Wesley, 1997
  - Developing Software with UML, Bernd Oestereich, Addison Wesley, 1999

- We also discuss some additional diagrams, not part of UML. The diagrams discussed in this chapter are presented in order from highest level of abstraction, dealing with descriptions of top-level software activities to the lowest level of design detail, concerning states and control.
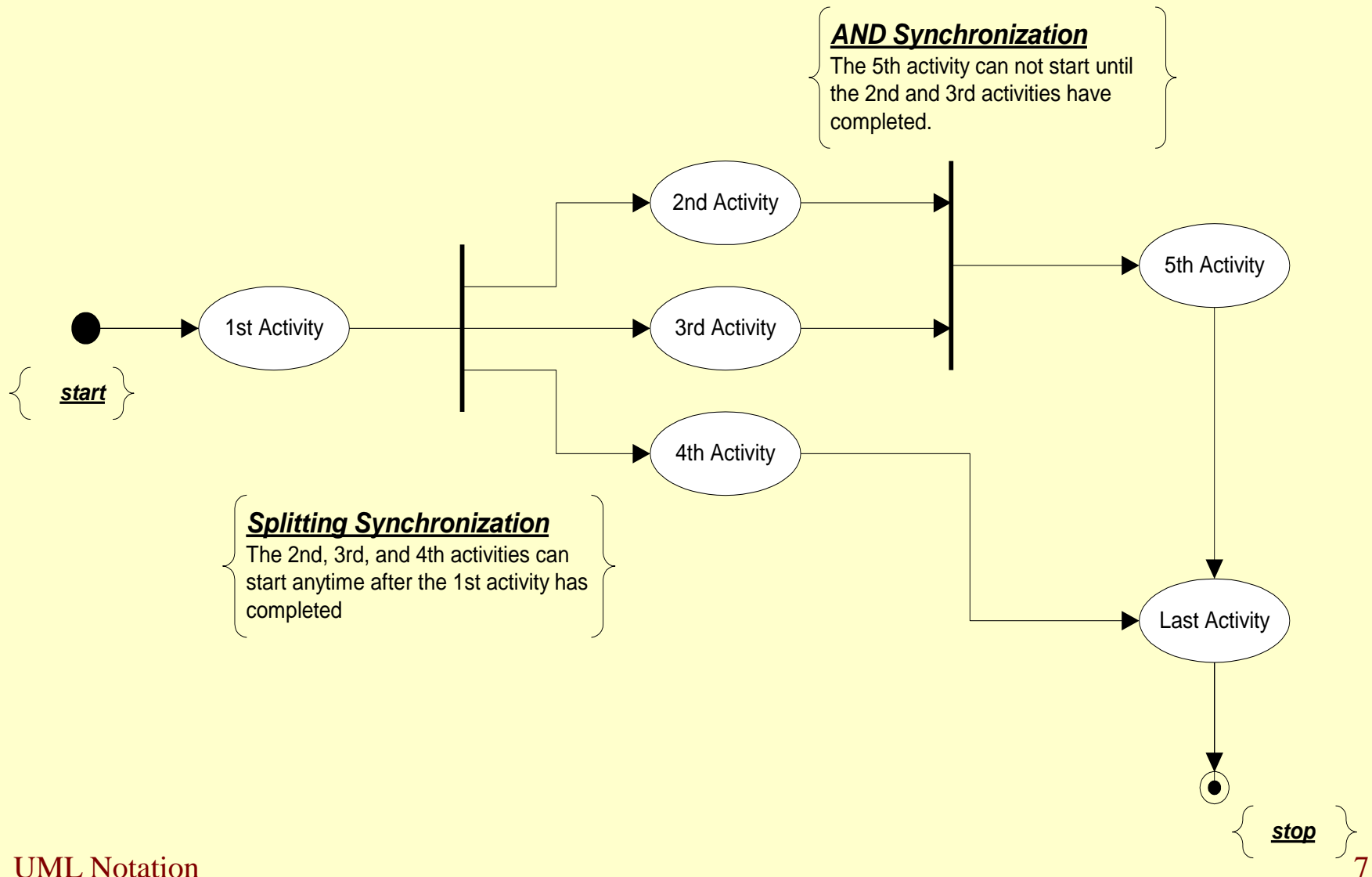
# Architecture and Requirements

- ***Activity Diagram***
  Used for high level descriptions of program behavior, often associated with software architecture.
  - shows the activities a program carries out
  - which activities may be conducted in parallel
  - which activities must be synchronized for correct operation

- ***Package diagram***
  Package structure of program or software system.
  - Enumerates all software packages
  - Used to show calling dependencies between packages

- ***Module diagram***
  (variant of structure chart – see below)
  One of the main diagrams used to describe software architecture.
  - shows calling dependencies between modules

- ***Context diagram***
  Also a high level description, used in documentation of architecture.
  - used to show how a program interacts with its environment

- ***Data Flow diagram***
  Used in requirements documents.
  - represents processing requirements and the information flows necessary to sustain them

# Design Documentation

- **_Class diagram_ (OMT diagram)**
  - shows classes that are used in a program along with their relationships
  - sometimes also shows their physical packaging into modules
- **_Event Trace diagram_**
  - illustrates the timing of important messages (member function invocations) between objects in the program
- **_Structure Chart_**
  - shows calling relationships between every function in a module and the calls into and out of the module
- **_State Diagram_**
  - shows how program navigates through its states
- **_Data structure diagram_**
  - illustrates the layout and relationships between important pieces of data in the program

UML Notation

# Activity Diagram (Petri Net)

- An Activity diagram shows:
  - activities a program carries out
  - which activities may be conducted in parallel
  - which activities must be synchronized for correct operation
- Each activity is shown by a labeled bubble.
- Start and stop activities are shown by darkened circles.
- Two or more activities which can be conducted in any order or in parallel are shown starting after a synchronizing bar.
- If two or more activities must all be completed before another activity begins, the synchronized activities are shown flowing into a synchronizing bar.
- Activities shown in series must be completed in the order shown.

# Activity Diagram

**AND Synchronization**
The 5th activity can not start until the 2nd and 3rd activities have completed.

start

1st Activity

2nd Activity

3rd Activity

4th Activity

5th Activity

Last Activity

**Splitting Synchronization**
The 2nd, 3rd, and 4th activities can start anytime after the 1st activity has completed
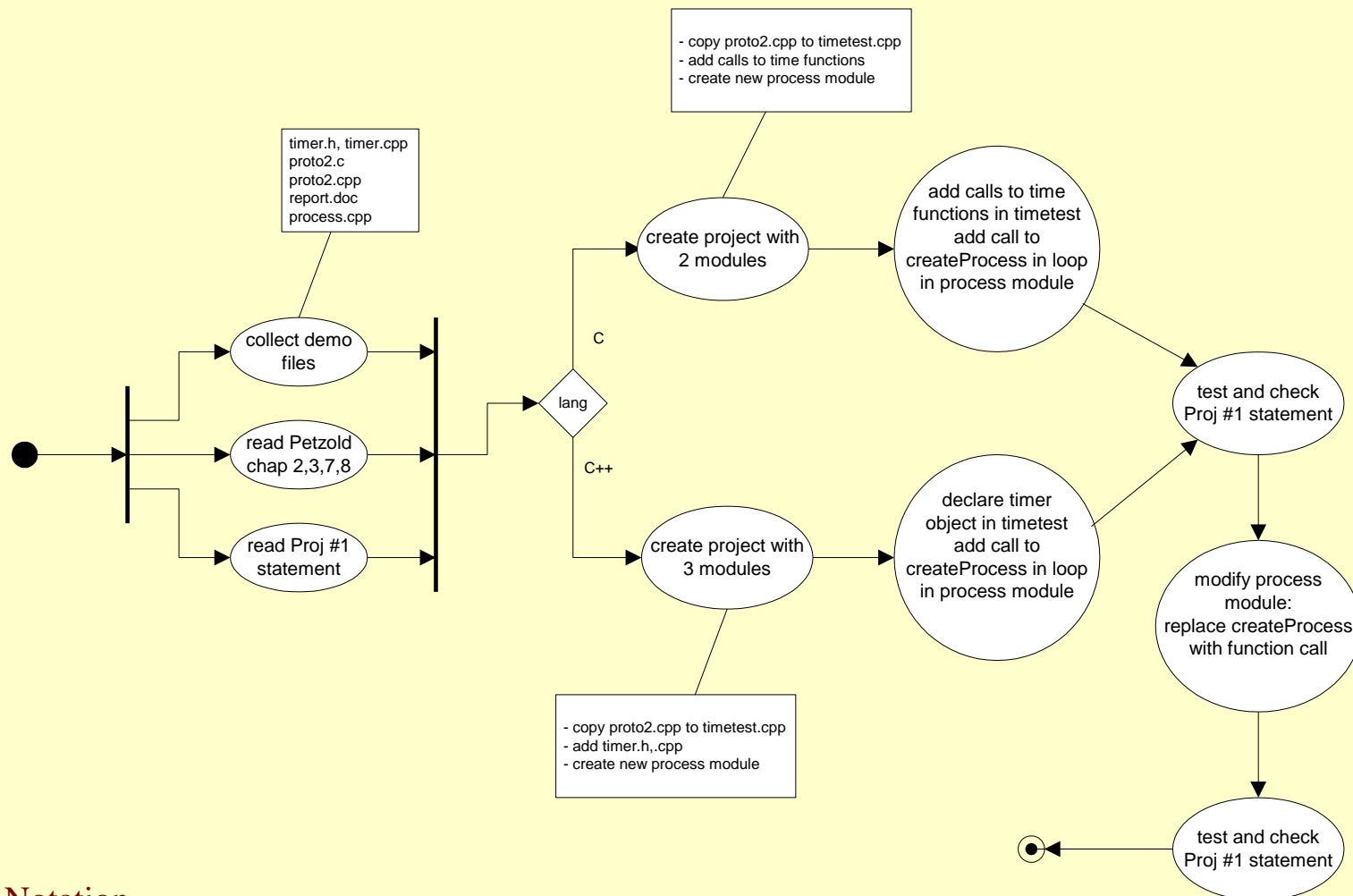
stop

UML Notation

7

# Activity Diagram

- The Activity Diagram is used to model high level activities in programs and systems. It is particularly useful for representing business systems and other human activities.

- The activity diagram is especially useful for representing systems that use synchronization. Often the synchronization points, shown by thick bars, are places where materials or information is enqueued, waiting for a subsequent activity to begin.

# Extended Petri Nets

- Activity diagrams extend the notation used for Petri nets by explicitly showing decision operations with a diamond symbol and labeled paths flowing out of the decision operation.

- Activity diagrams which incorporate decision processing are used in much that same way that flow charts were used (one of the earliest forms of graphical program documentation).

- They are more powerful than flow charts, however, as they make explicit the opportunity for parallel processing and the need for synchronization.

# Activity Diagram Example

- This Diagram represents work remaining to do on a Project.

# Context Diagram

- A context diagram shows how the processing you will build interacts with its environment.
  - Each rectangle represents some source of information used by your program or some sink of information provided by your program.  Your program does not provide these sources and sinks.
  - The central oval represents <u>all</u> the processing you are obligated to develop.
  - Each line represents information required for your processing to succeed (inputs) or information your processing will generate (outputs).

- The information flows shown on the context diagram must match <u>exactly</u> the inputs and outputs on your top level Data Flow Diagram (DFD), described next.

UML Notation

# Context Diagram <u>Contents</u>
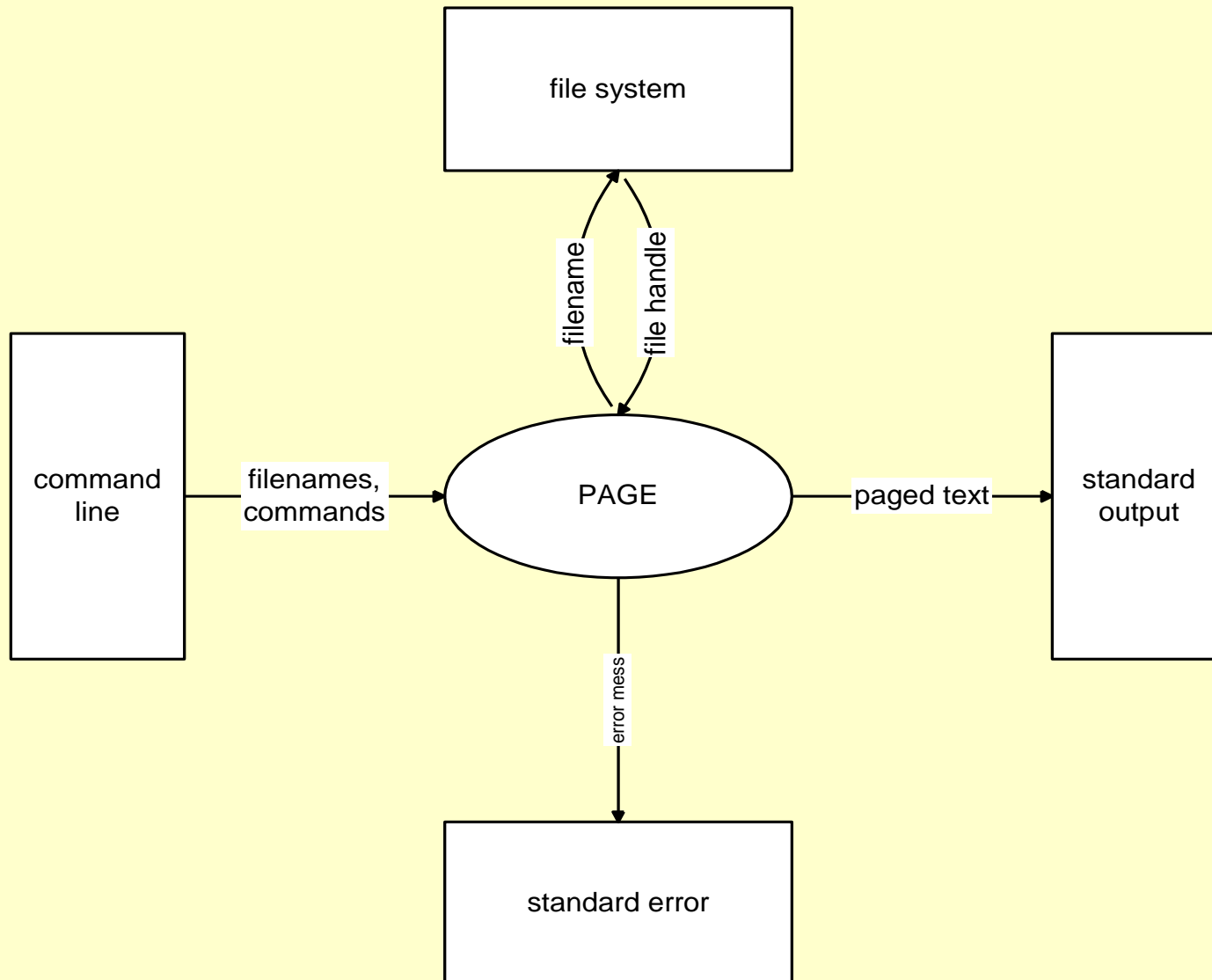
# Data Flow Diagram

- A data flow diagram represents processing requirements of a program and the information flows necessary to sustain them.
  - All processing represented by the context diagram is decomposed into a set of a few (perhaps three or four) process bubbles which are labeled and numbered.
  - The information necessary to sustain each process and generated by each process are shown as input and output data flows.
  - Inputs from the environment and outputs to the environment are show <u>exactly</u> as they appear in the context diagram.
  - When the inputs and outputs exactly match the context diagram we say that the data flow diagram is balanced.
  - If each of the processes represents approximately the same amount of requirements detail we say that the diagram is properly leveled.
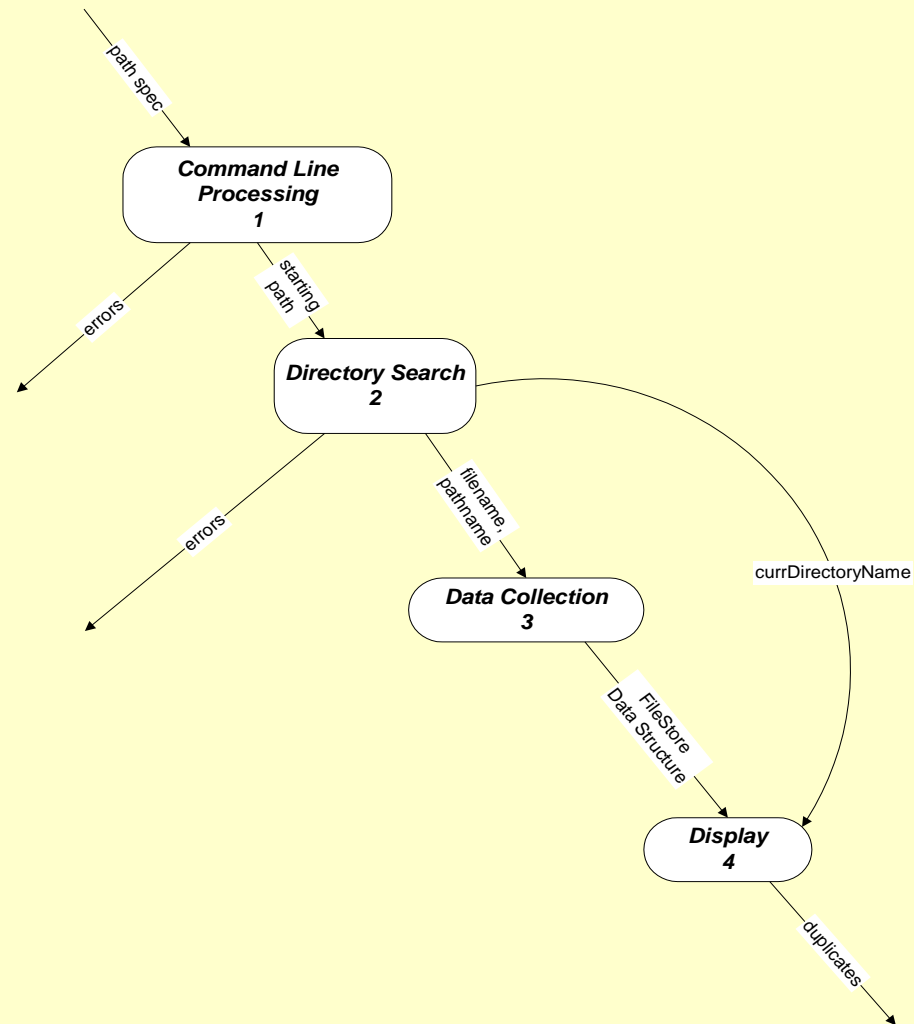
# Data Flow Diagram Contents
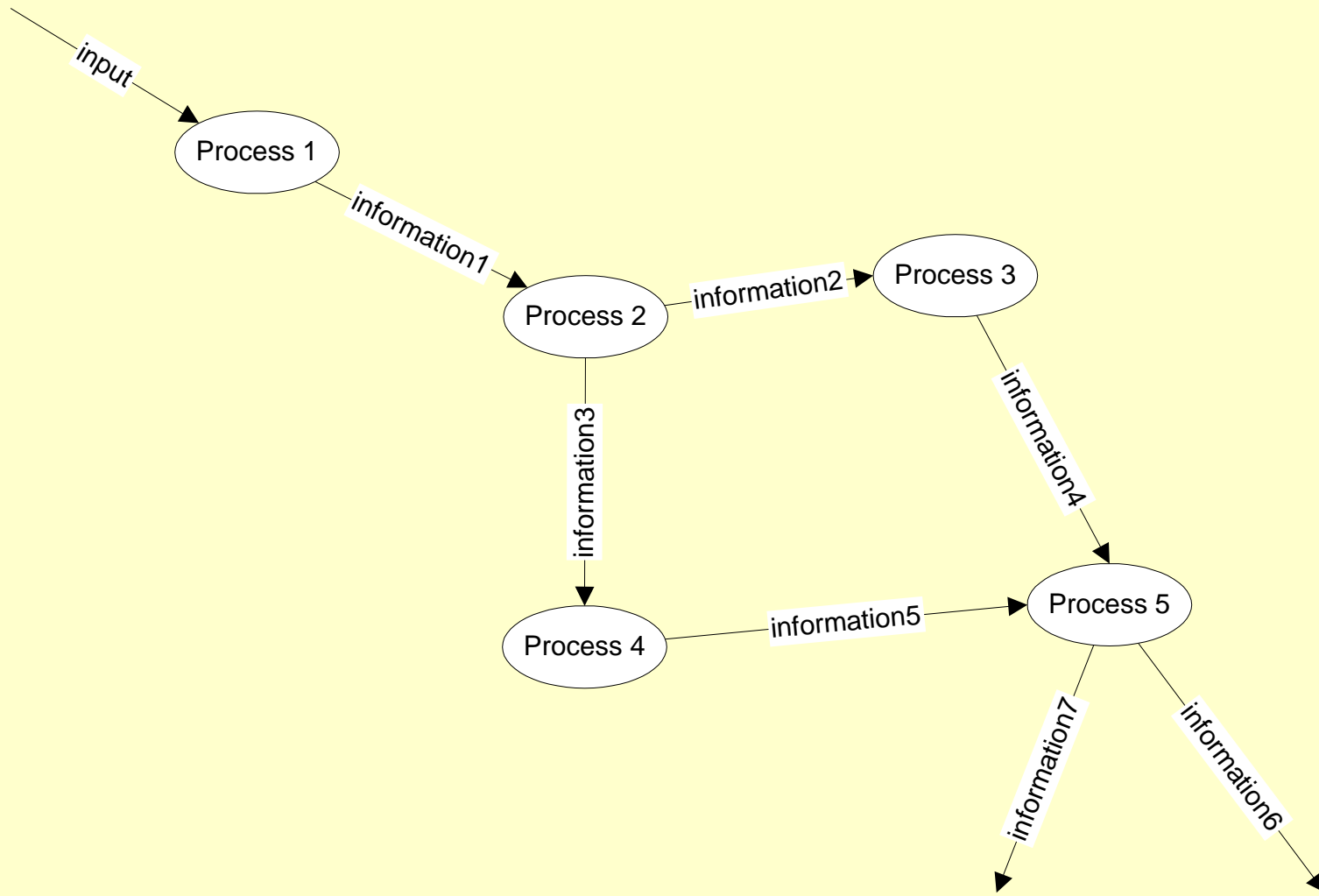
# Data Flow Diagram [Contents](#)

- Data Flow Diagrams (DFDs) are used during the analysis of requirements for complex systems.  Each bubble represents a specific process which has been allocated tasks and requirements, so that all of the program's obligations are partitioned among the processes shown on the top level DFD.

- Each data flow represents information necessary to sustain a process or generated by a process.

- Note that Data Flow Diagrams are not officially part of the UML.

UML Notation

# An Example Data Flow Diagram
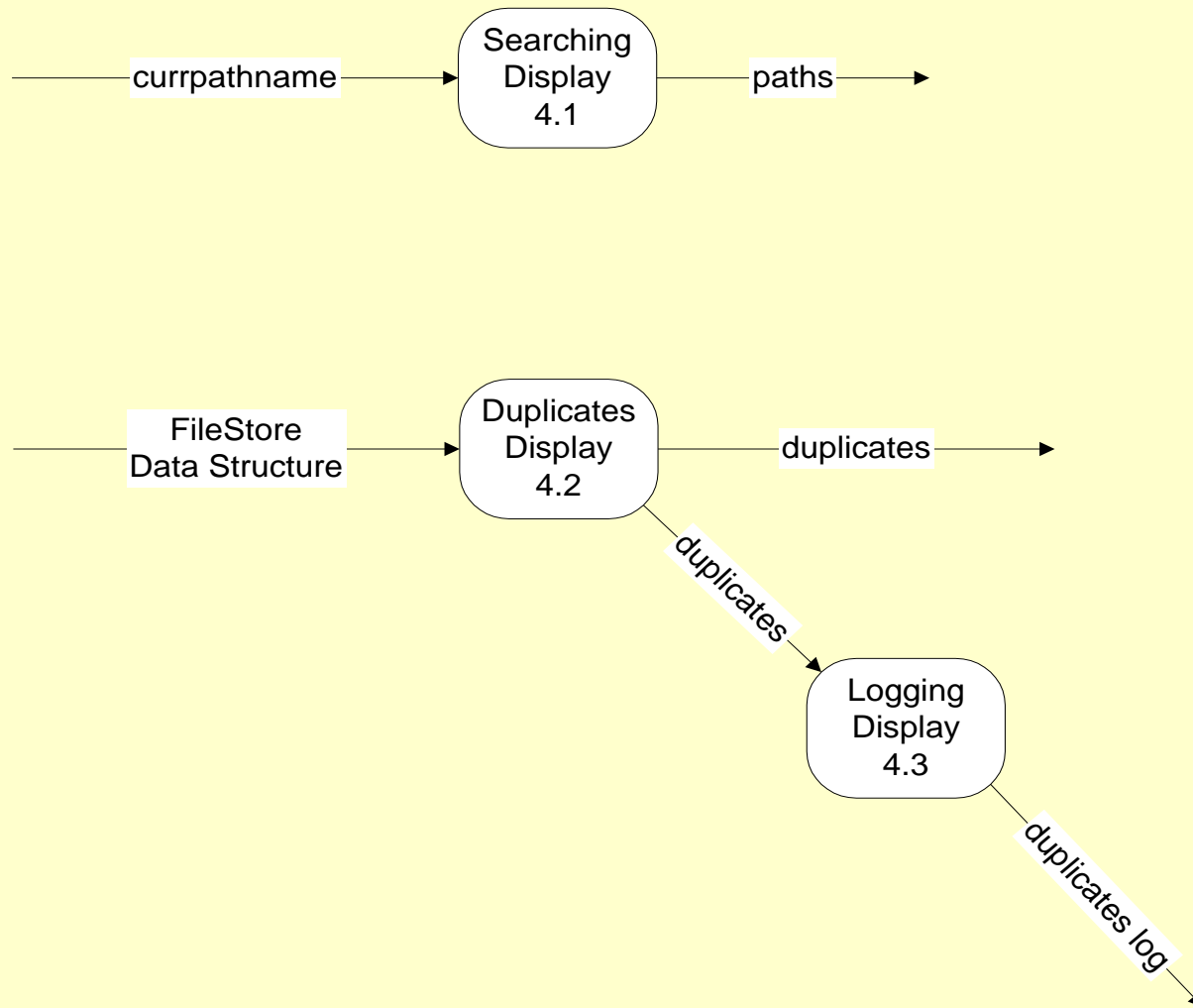
This diagram represents processing in the DUPLICATES program.

# Lower Level Data Flow Diagrams

- We usually divide the processes in a data flow diagram into logical operations which may not all need the same amount of detail to describe their processing requirements.  When this is the case, we decompose the more complex processes into lower level data flow diagrams.

    – If a process is decomposed into lower level sub-processes this is shown on a lower level data flow diagram.

    – Each process in the lower level data flow diagram must be numbered showing its parent's number and a unique number for each of its own processes, e.g., 3.4.

    – The lower level diagram <u>must</u> balance with its parent.  That is, each of its input flows and output flows must match those of its parent.

    – If necessary a lower level data flow diagram may be further decomposed into still lower level diagrams.  This is not uncommon for complex programs.

# Duplicates Program
# Lower Level Data Flow Diagram

currpathname ⟶ **Searching Display 4.1** ⟶ paths ⟶

FileStore Data Structure ⟶ **Duplicates Display 4.2** ⟶ duplicates ⟶

**Duplicates Display 4.2** ⟶ *duplicates* ⟶ **Logging Display 4.3** ⟶ *duplicates log* ⟶

UML Notation

# Class Diagrams

- A class diagram shows the classes that are used in a program along with all relevant relationships between classes.

  - A class diagram sometimes also shows the physical packaging of classes into modules.

  - There are two especially important relationships between classes:

    - Aggregation shows an ownership or "part-of" relationship. This relationship is denoted by a line with a diamond attached to the owning class and terminating on the owned class. The UML requires the aggregation diamond to be filled with black if the owning class creates and destroys the owned object.

    - Inheritance shows a specialization or "is-a" relationship between classes. This relationship is denoted by a line with a triangle pointing toward the base class. The line terminates on one or more derived classes which specialize the behaviors of the base class. However, each derived class is required to handle all of the messages the base class responses to and are therefore also considered to be (specialized) base class objects.
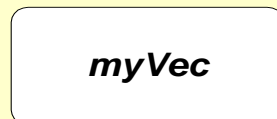
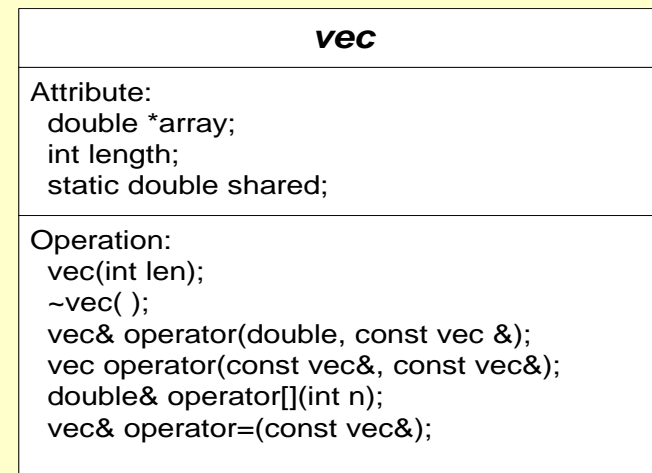# Classes and Objects

Class : a set of objects of one specific type

class symbol

```
vec
```

object of class

```
myVec
```

class symbol with details

| vec |
|---|
| Attribute:<br> double *array;<br> int length;<br> static double shared; |
| Operation:<br> vec(int len);<br> ~vec( );<br> vec& operator(double, const vec &);<br> vec operator(const vec&, const vec&);<br> double& operator[](int n);<br> vec& operator=(const vec&); |

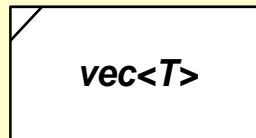# Objects

Object: an element of some class

Each class represents a specific collection of data attributes of one or more types (its state) and a collection of functions (behaviors) which modify or disclose the state of an object of the class.

Each class has, by default, a unique state, independent of any other object of the class. However, a class may declare that one or more data members must be shared by all objects of the class.
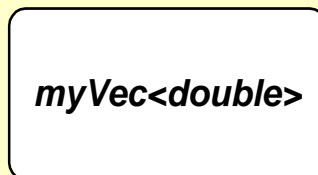
# Generic Classes <span>Contents</span>

- It is frequently convenient to define a class in terms of a generic parameter of unspecified type. We call these generic classes and represent them with the symbols:
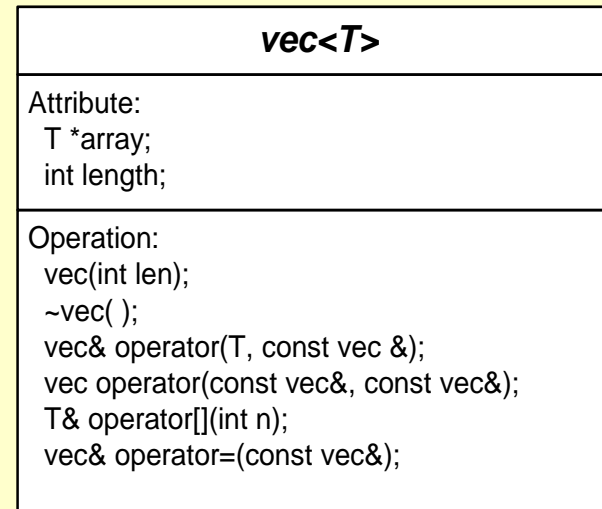
class symbol

```
| vec<T> |
```

class symbol with details

| vec<T> |
|---|
| Attribute:<br> T *array;<br> int length; |
| Operation:<br> vec(int len);<br> ~vec( );<br> vec& operator(T, const vec &);<br> vec operator(const vec&, const vec&);<br> T& operator[](int n);<br> vec& operator=(const vec&); |

object of class

```
myVec<double>
```

# Template Class Declaration

```
template <class T> class vec {

  public:
    vec(int size=0);                    // constructor
    vec(const vec<T>& v);               // copy constructor
    ~vec(void);                         // destructor
    vec<T>& operator=(const vec<T>&);   // assignment
        T& operator[](int n);          // indexing
         T operator[](int n) const;    // indexing
     vec<T> operator*(T &t);            // scalar multiplication
     friend vec<T> operator*(T &t, const vec<T>& v);
                                        // scalar multiplication
     vec<T> operator+(const vec<T>&);   // vector addition
     vec<T> operator-(const vec<T>&);   // vector subtraction
     vec<T> operator*(const vec<T>&);   // vector multiplication
         T operator,(const vec<T>&);    // inner product
      int size();                       // show size
      void write(ostream&, int, int);  // formatted write to output
    friend ostream& operator<<(ostream&, const vec<T>&);
                                        // output stream inserter
      void read(istream&);             // formatted read from in
    friend istream& operator>>(istream&, vec<T>&);
                                        // input stream extractor
  private:

        char *_vName;         // pointer to name allocated on heap
         int _arSize;         // vector dimension
          T *_array;          // pointer to array allocated on heap
};
```
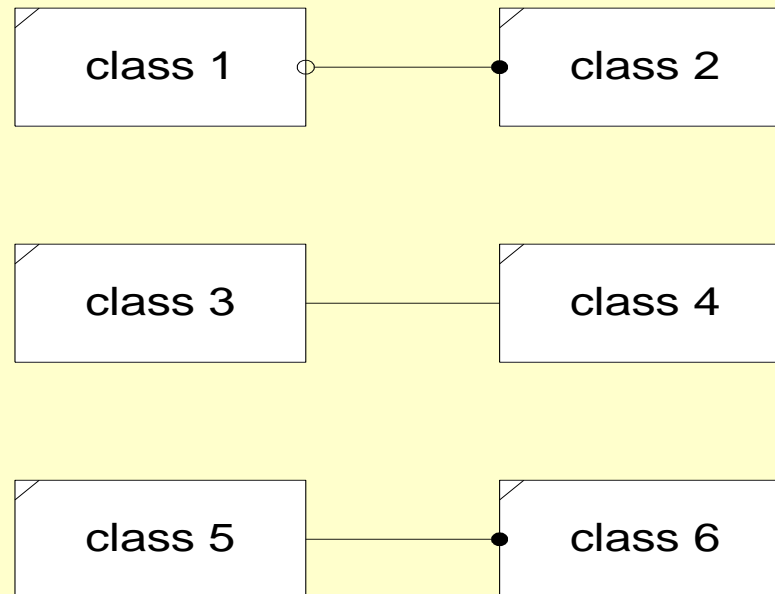
UML Notation                                                    23

# Associations

An association is a relationship linking two or more classes or objects.

# Relationships

- The hollow ball indicates a multiplicity of zero or one for class 1

- The solid ball indicates a multiplicity of zero or more (many) for class 2.

- Absence of a ball indicates a multiplicity of one.

- There is one-to-one relationship between classes 3 and 4 and one to many relationship between classes 5 and 6.

# Link Attributes

- A model may have attributes which clearly belong to the association relationship rather than to one of the classes in the association. In this case the association is given those attributes, and is denoted as shown below.
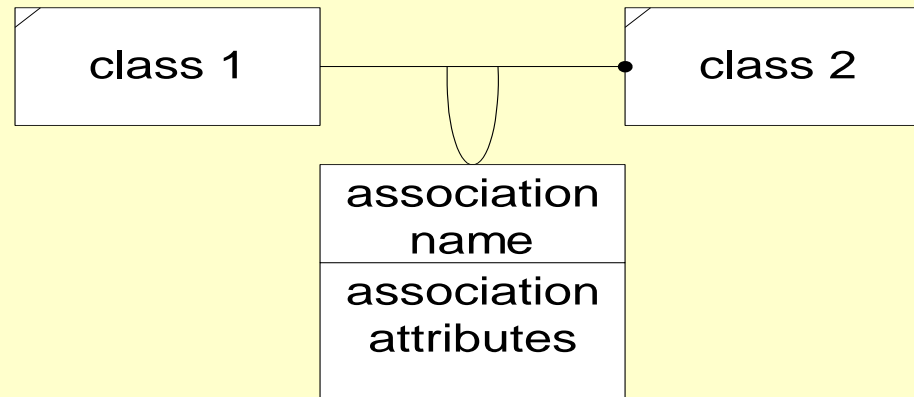


Here, class 1 and class 2 have a one to many relationship in which the relationship has attributes denoted by an association attributes list.

# Association Examples

The diagram below captures the relationship between a student and her department.
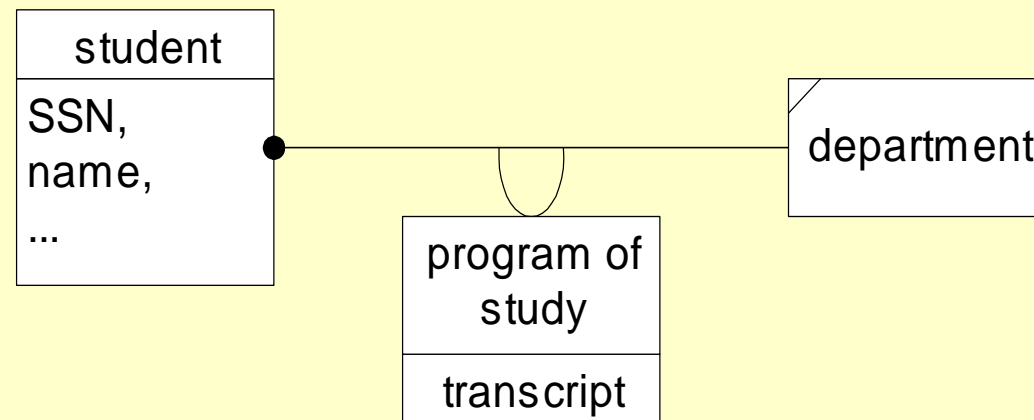


Here, the program of study is meaningless without the relationship between the student and her department.
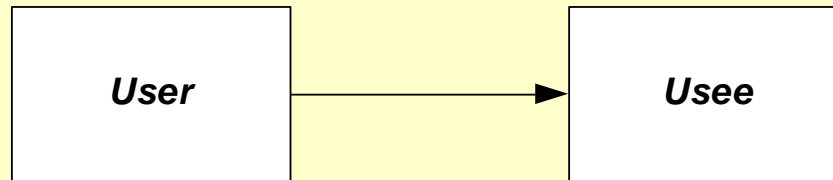
# Association Example

- This second diagram illustrates the relationship between a team leader and his team members.



team member
2-5

project person

skills, assignment, role

team leader
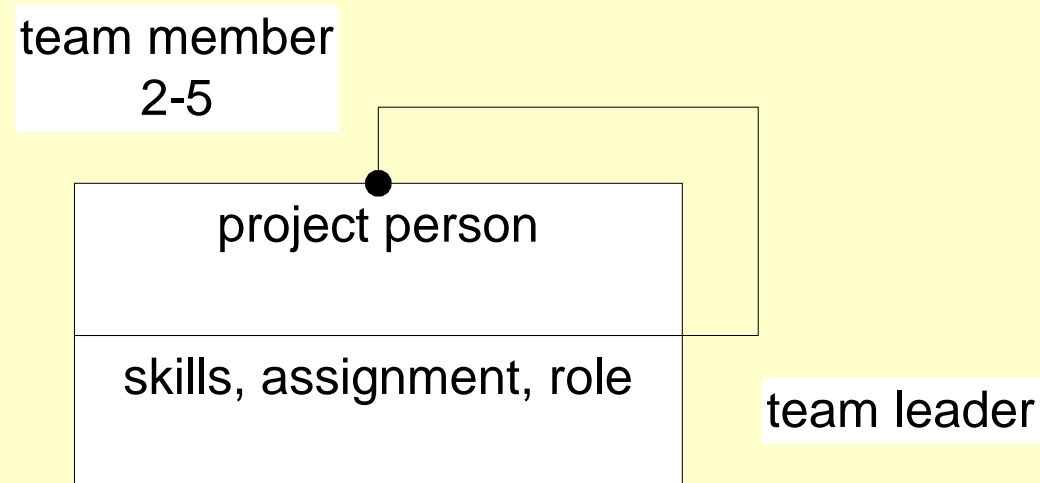
# Using Relationship

- Using is an association that models one class using the behavior of another to carry out its own activities.

```
┌──────────────┐                    ┌──────────────┐
│              │                    │              │
│     User     │────────────────▶   │     Usee     │
│              │                    │              │
└──────────────┘                    └──────────────┘
```

- The using relationship is implemented when a member function of the User class is passed an object of the Usee class or when it creates a local instance of the Usee class.

- In a typical design there are many using relationships – too many to show conveniently.  In this case we show only those that are critical to the design.
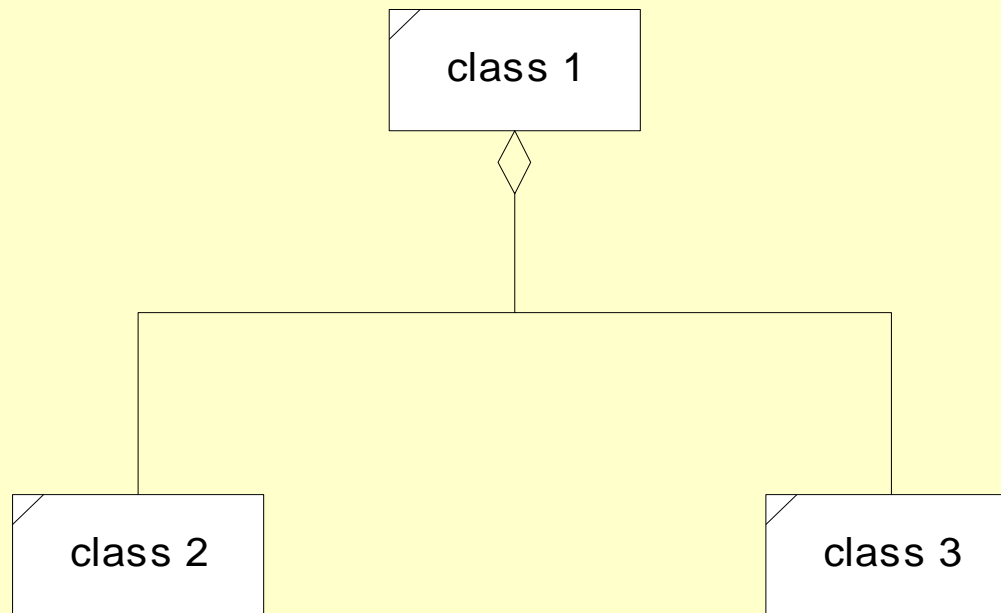
```cpp
class User {

  public:
    User(const std::string &name);
    void show(Usee &usee);

  private:
    std::string _name;
};
```
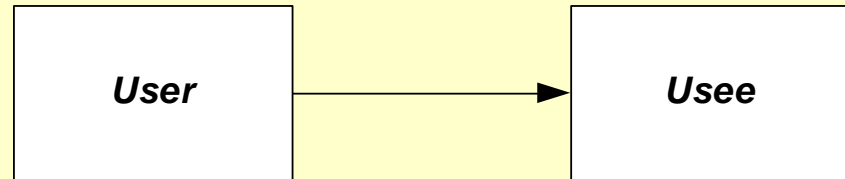


```cpp
Void User::show(Usee &usee1) {

  Usee usee2("Jake");
  std::cout << "\n  I am " << _name << ", and use two objects.";
  usee1.showUsee();
  usee2.showUsee();
}
```

UML Notation

30

# **Aggregation**

- Aggregations are special associations which model a "part-of" or "contained" semantic relationship.



In this diagram class 1 contains classes 2 and 3. Classes 2 and 3 are part-of class 1.

# Aggregation and Composition

Aggregations are "part-of" or containment relationships. Here, class 2 is part of class 1. A stronger form of aggregation is the composition relationship. This is an aggregation in which the part-of represents an exclusive ownership. An owned object is created when the owner is created and is destroyed when its owner is destroyed.
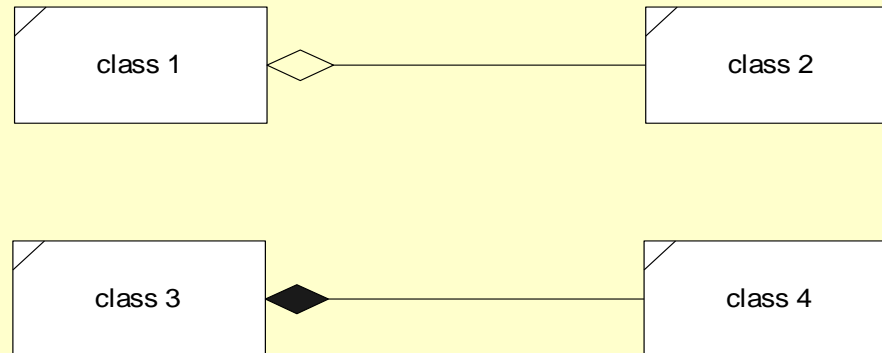


Composition is denoted by a dark fill in the diamond end of the aggregation symbol. When C++ programmers use the term aggregation they mean this stronger compositional form since aggregation is usually implemented by making the owned class a data member of the owning class. In C++ this creates the stronger compositional relationship.
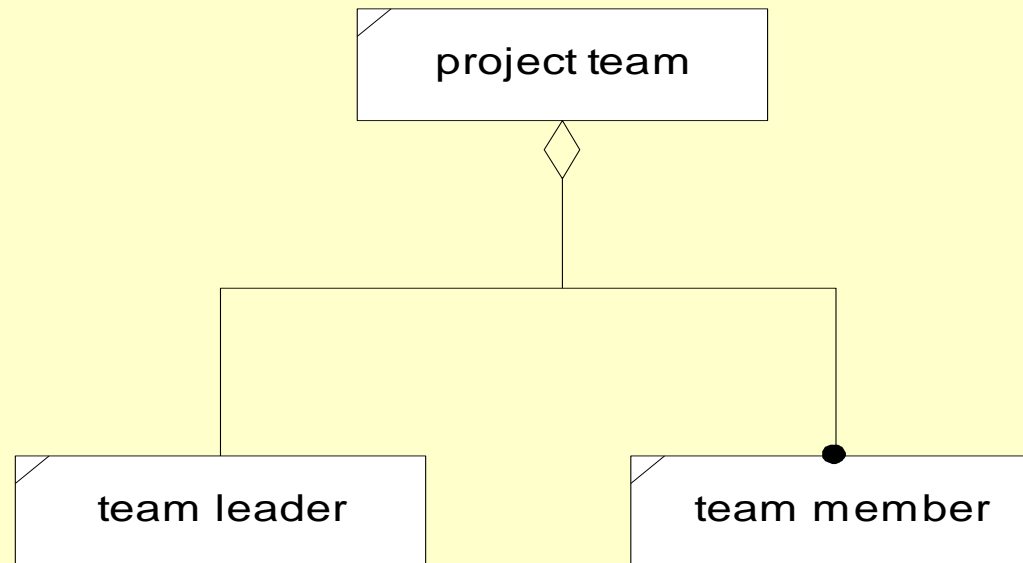
# Composition Example

```
class part {

  public:
    part(const std::string &name);
    void showPart();

  private:
    std::string _name;
};

class whole {

  public:
    whole(const std::string &name);
    void show();

  private:
    std::string _name;
    part a;
    part b;
};
```
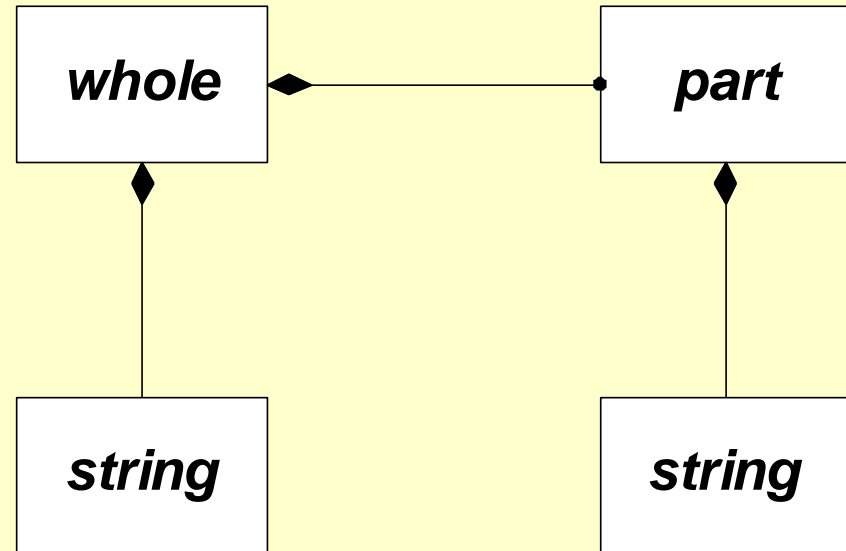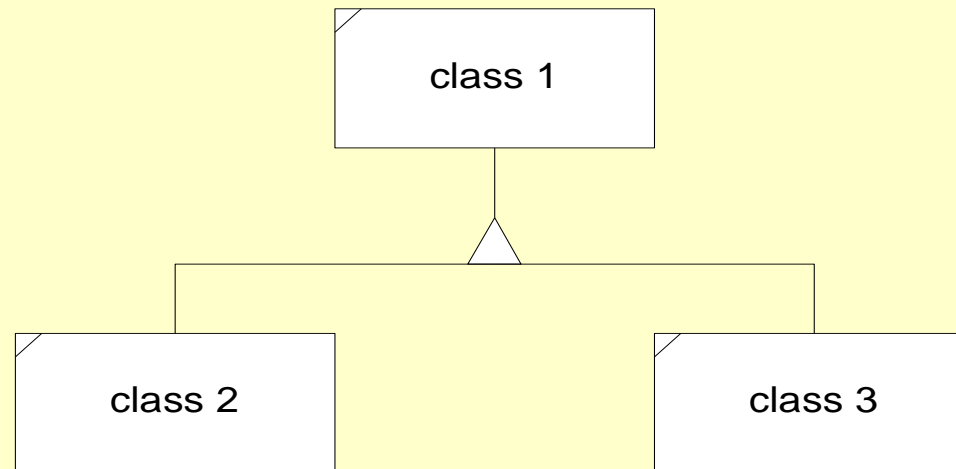
# Aggregation Example

The diagram below illustrates the aggregation relationship inherent in a team.

```
                    ┌──────────────────┐
                    │  project team    │
                    └──────────────────┘
                             ◇
              ┌──────────────┴──────────────┐
    ┌──────────────────┐          ┌──────────────────┐
    │  team leader     │          │● team member     │
    └──────────────────┘          └──────────────────┘
```

The behaviors and attributes of the team are the sum of all the behaviors and attributes of the team leader and all the team members. In this sense aggregation represents an "and" semantic relationship.
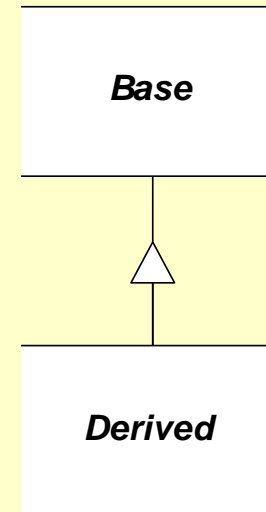
# Inheritance

Inheritance models an "is-a" semantic relationship.  Here the classes 2 and 3 inherit from class 1.  We say that  classes 2 and 3 are derived from base class 1.

```
                        ┌──────────────┐
                        │  class 1     │
                        └──────┬───────┘
                               │
                              △
                    ┌──────────┴──────────┐
         ┌──────────┴───┐          ┌───────┴──────┐
         │   class 2    │          │   class 3    │
         └──────────────┘          └──────────────┘
```

That means that class 2 "is-a" class 1 and the same must be true for class 2. The "is-a" relationship is always a specialization.  That is, both classes 2 and 3 must have all attributes and behaviors of class 1, but may also extend the attributes and extend and modify the behaviors of class 1.
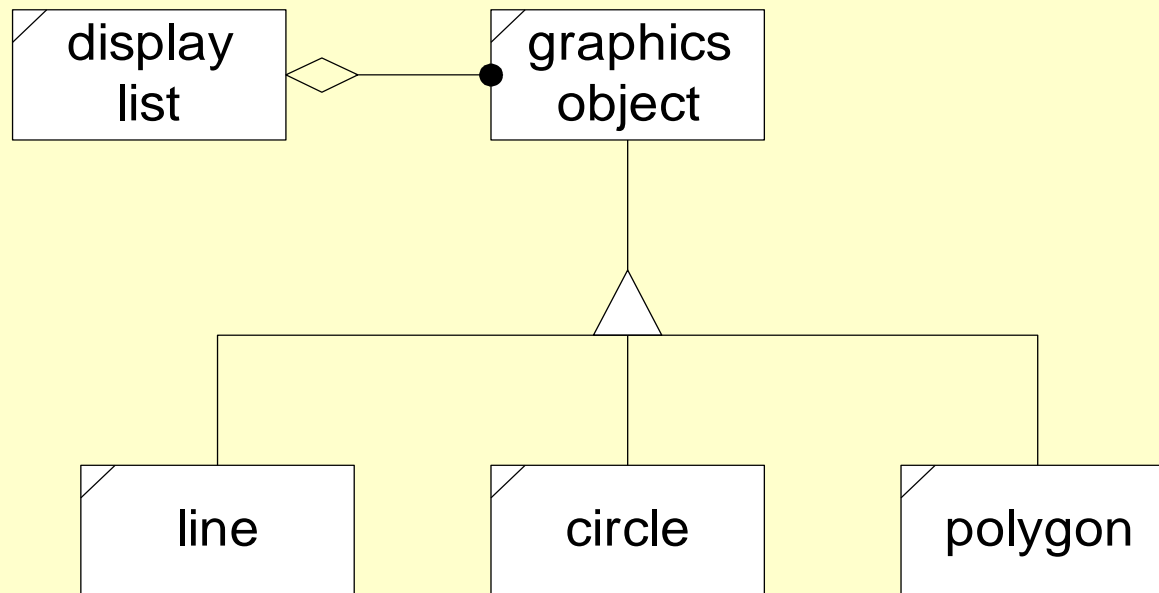
# Inheritance Example

```cpp
class Base {

  public:
    Base(const std::string &name);
    virtual ~Base() { }
    virtual void show();

  private:
    std::string _name;
};

class Derived : public Base {

  public:
    Derived(const std::string &name);
    virtual void show();

  private:
    std::string _name;
};
```

# Inheritance Example

The inheritance diagram below represents an architecture for a graphics editor.  The display list refers to graphics objects, which because of the "is-a" relationship, can be any of the derived objects.

# Inheritance Example

The base class graphicsObject provides a protocol for clients like the display list to use, e.g., draw(), erase(), move(), ...  Clients do not need to know any of the details that distinguish one of the derived class objects from another.
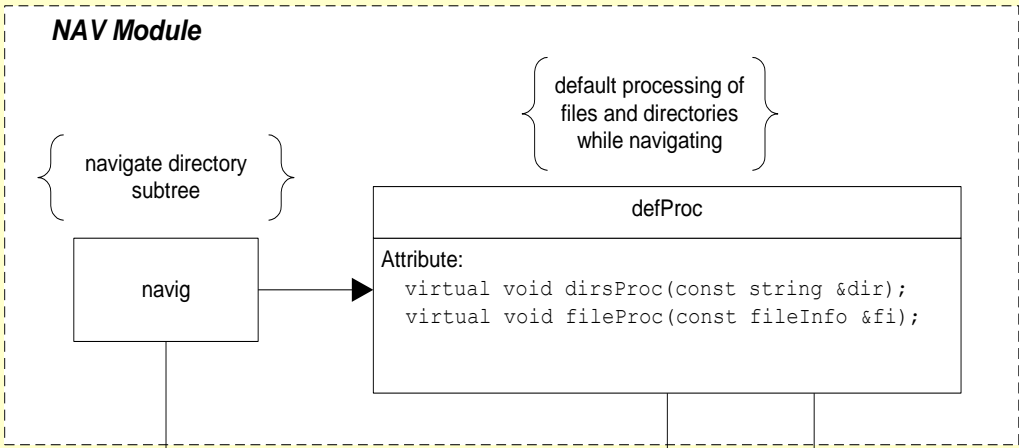
In C++, the protocol functions are qualified as virtual.  This means that a derived class may override any base class definition to provide class specific semantics for this function.  Furthermore, this means that the list manager client can be ignorant of specific types of objects addressed, simply calling the base protocol on any one of them.

# Importance of Polymorphism

- When a base class provides a protocol by defining one or more virtual functions that are overridden by derived classes, clients can use the base protocol to interact with any of the derived classes and need not know the details that distinguish one derived class from another.  This is called ***polymorphism***.

- Polymorphism lets us minimize coupling between clients and the objects they use.

- Polymorphism also allows us to extend a library to satisfy the needs of an application, provided that the library designer has defined a base protocol and allowed us to derive from that base.  The next example illustrates this.  A directory navigation object uses a base processing class that applications can derive from to insert their own processing into the computa-tional stream.
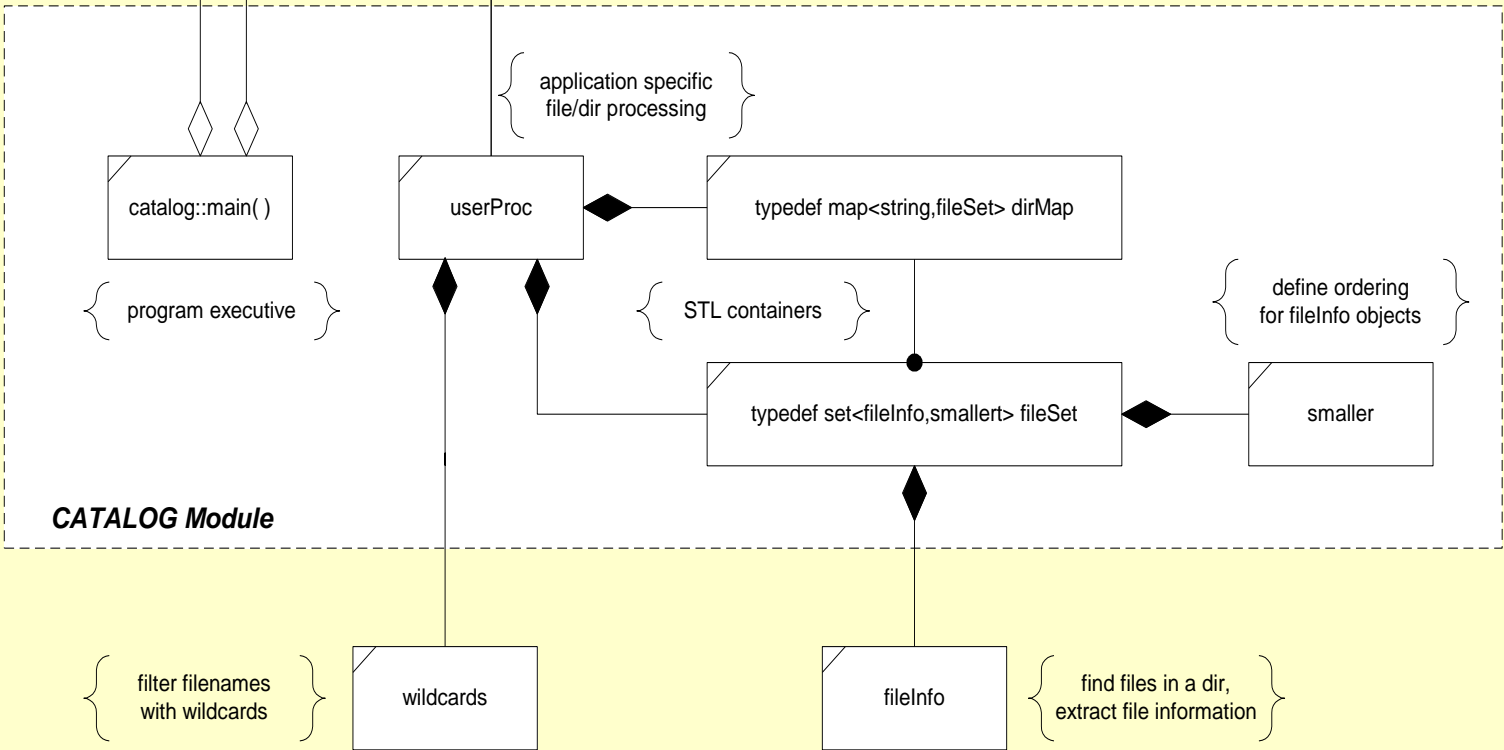
# Class Diagram Example
Logical and Physical Structures of the CATALOG program

## NAV Module

{ navigate directory subtree }

{ default processing of files and directories while navigating }

**navig**

**defProc**

Attribute:
```
virtual void dirsProc(const string &dir);
virtual void fileProc(const fileInfo &fi);
```

Note that catalog::main( ) and navig actually refer to a userProc object through defProc pointers

## CATALOG Module

{ store a set of directories and their associated files }

{ application specific file/dir processing }

**catalog::main( )**

{ program executive }

**userProc**

**typedef map<string,fileSet> dirMap**

{ STL containers }

{ define ordering for fileInfo objects }

**typedef set<fileInfo,smallert> fileSet**

**smaller**

{ filter filenames with wildcards }

**wildcards**

**fileInfo**

{ find files in a dir, extract file information }

# Typical Output from CATALOG

# Multiple Inheritance

A derived class may have more than one base class.  In this case we say that the design structure uses multiple inheritance.
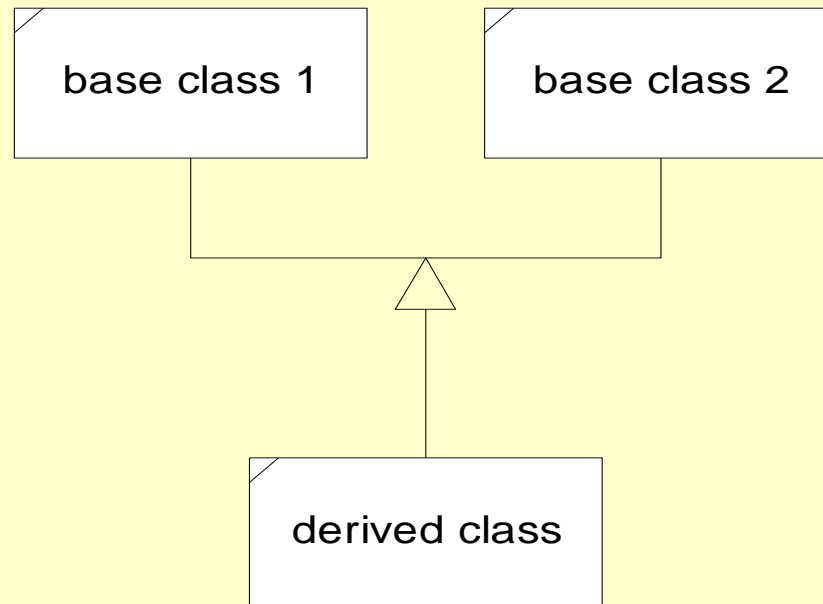
# Multiple Inheritance

The derived "is-a" base 1 and "is-a" base 2. Multiple inheritance is appropriate when the two base classes are orthogonal, e.g., have no common attributes or behaviors, and the derived class is logically the union of the two base classes.

The next page shows an example of multiple inheritance taken from the C++ Standard Library iostream module. The classes iostream, ifstream, and ofstream all use multiple inheritance to provide their behaviors.

# IOSTREAM Hierarchy

## Contents
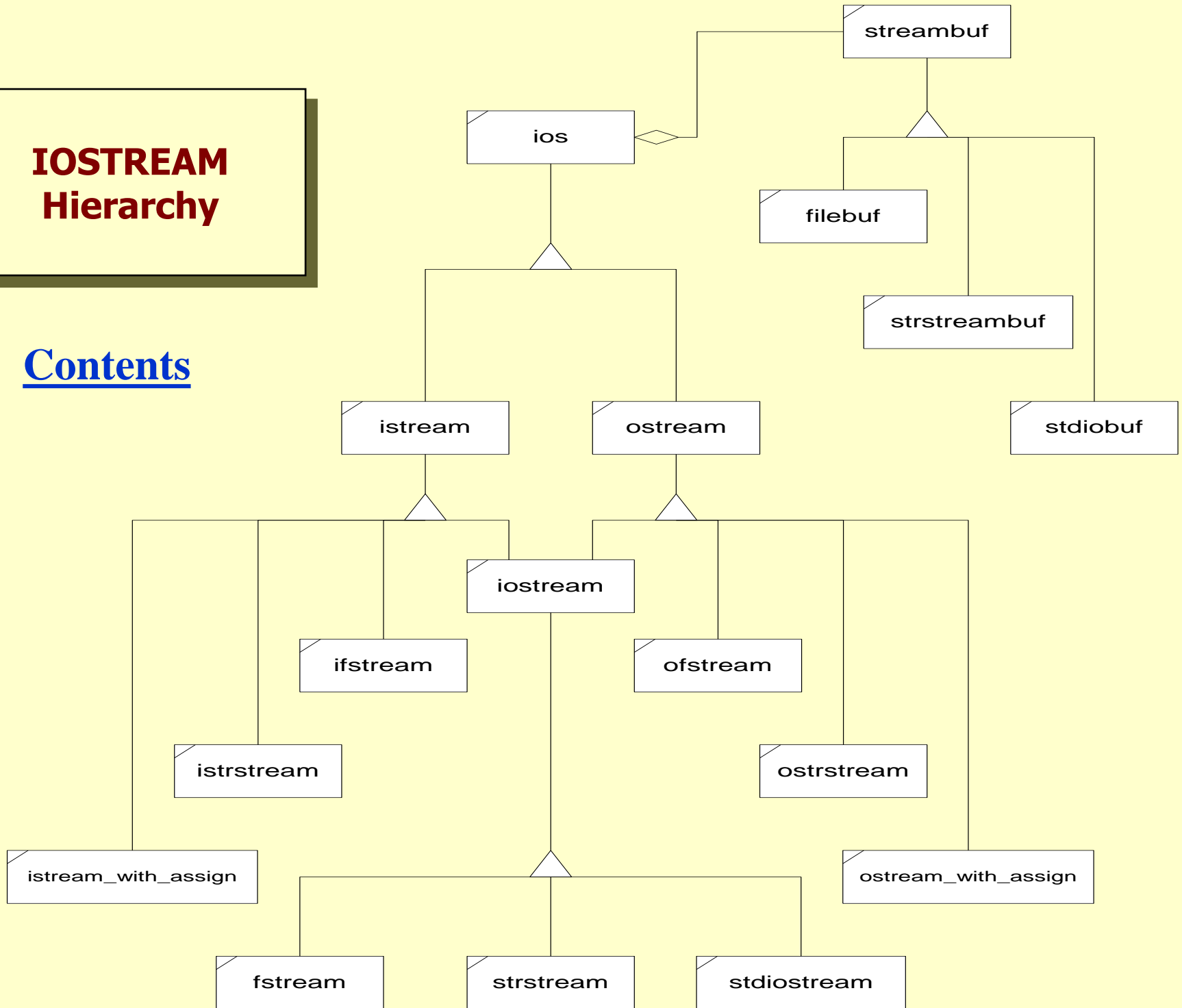
# Summary of Class Relationships

| Relationship | Diagram | Code | Explanation |
|---|---|---|---|
| **Inheritance**<br><br>**D "is-a" B** | B ◁— D | public class D : B { ... } | Derived class D is a specialization of the Base class B.  D inherits all the members of B except constructors |
| **Composition**<br><br>**Ownership, P is "part-of" C** | C ◆— P | public class C {<br>   ...<br>   private double p = 3.142;<br>} | Composite class C owns, or contains, a part class P.  P is created and destroyed with C. The interface of P is visible only to C, not its clients. Example: P is a value type. |
| **Aggregation**<br><br>**Ownership, P is "part-of" A** | A ◇— P | public class A {<br>   ...<br>   private P p = new P();<br>} | The Aggregator class A owns a part class P.  P is created by a member function of A, and so its lifetime is strictly less than that of A.  Example: P is a reference type. |
| **Using**<br><br>**Referral:<br>U uses R through a reference** | U —→ R | public class U {<br>   ...<br>   public void register(R r)<br>      // use r<br>   }<br>} | A class U uses instance of class R, to which it holds a reference.  R is created by some other entity and a reference to it is passed to some member function of class U. |

# Event Trace Diagram

- An Event Trace diagram illustrates the timing of important messages (member function invocations) between objects in a program.
    - Each object is shown by a vertical bar
    - Message traffic is shown by labeled horizontal lines flowing toward the object on which a method was invoked.
    - Time progresses downward in the diagram, but note that the diagram does not attempt to show iteration loops or calling options.  If one of two calls may be made depending on some condition they are either both shown or neither is shown.
    - Iterations are sometimes hinted by preceding a method name with a * symbol indicating that that method will be invoked multiple times in succession.
- These diagrams usually show the major events, but don't try to capture all little details - there may be hundreds of messages flowing, but perhaps only a few are important enough to show.

# Event Trace Diagram



UML Notation

47
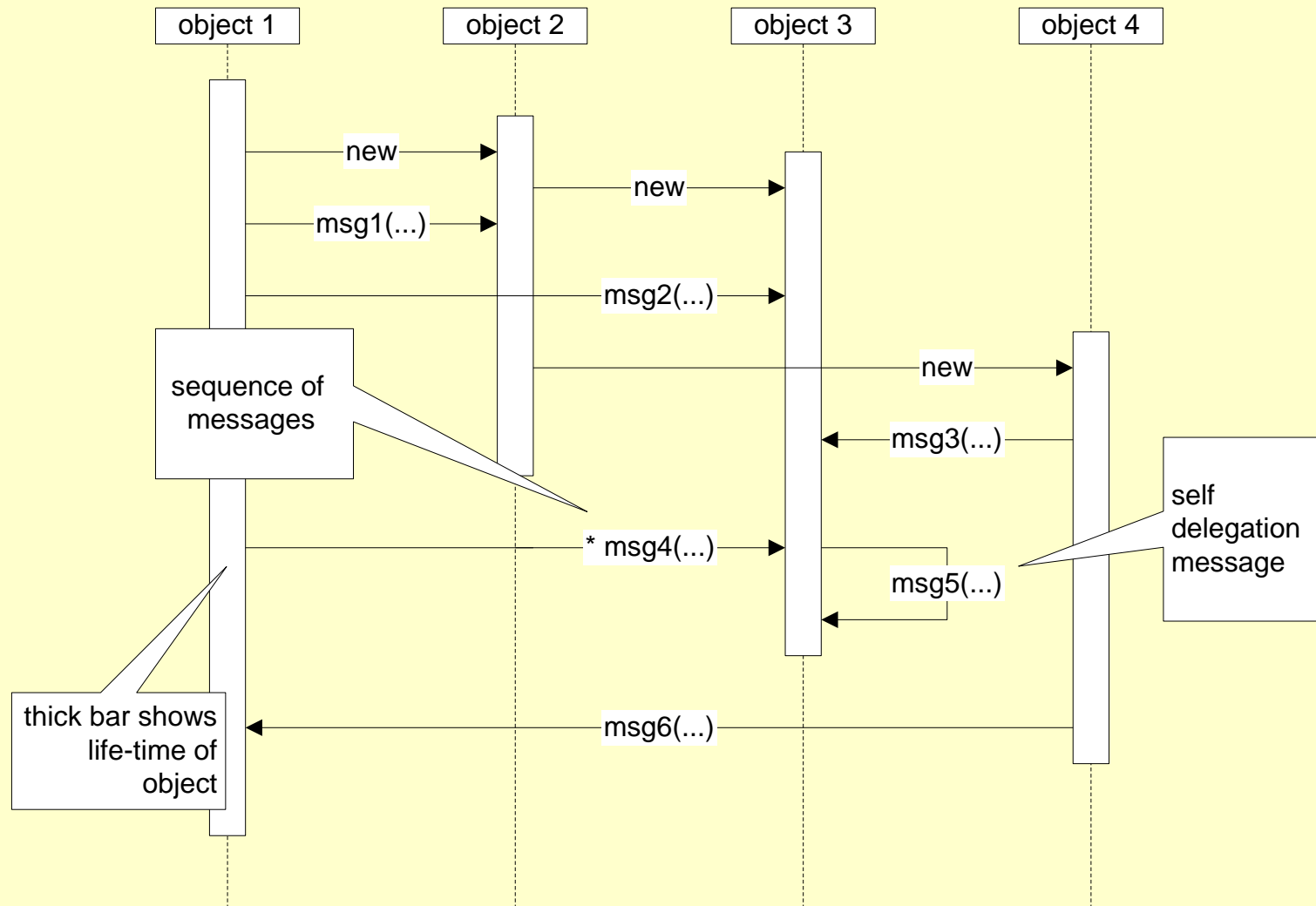
# Event Trace Diagram Example

This example is from the Duplicates program.



UML Notation
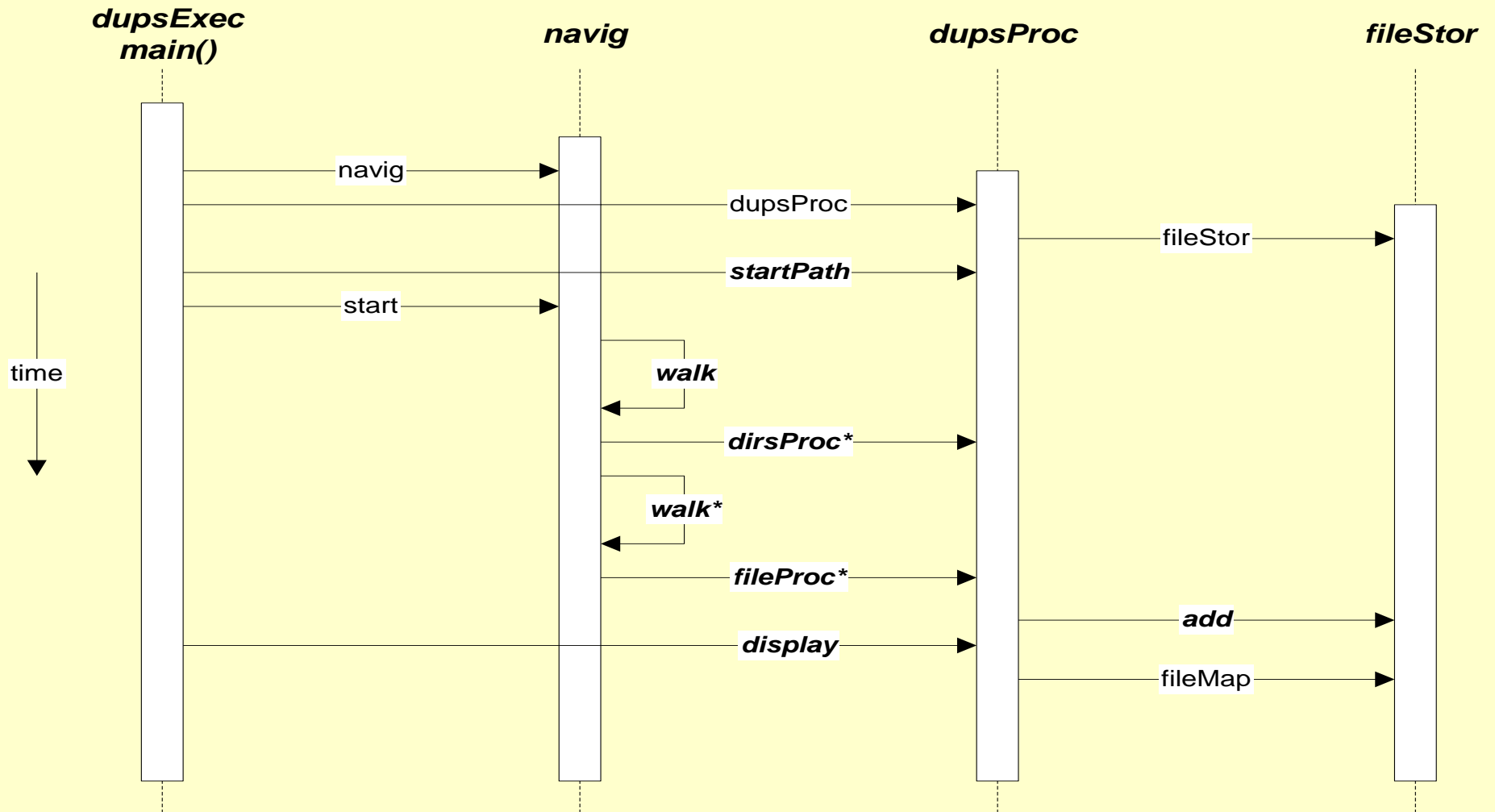
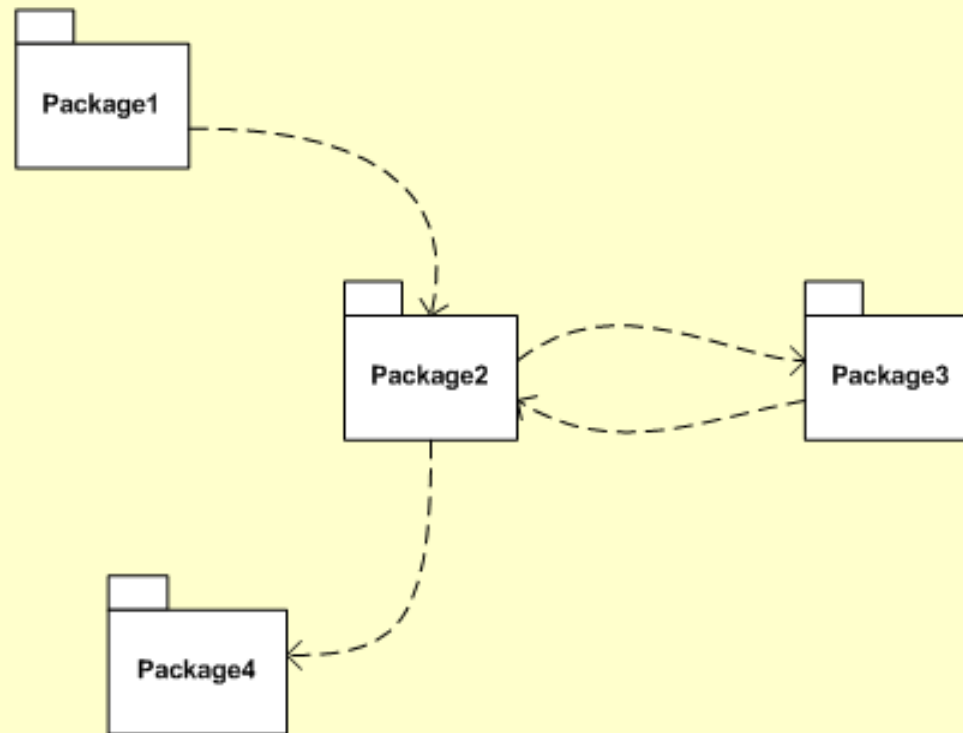# Module Diagram <span style="color:blue">**Contents**</span>

- Module diagrams show function calling dependencies between modules in a program.
  - Each module is represented by a labeled rectangle. Calling modules are shown above the modules they call.
  - A program should be decomposed into a single executive module which directs the activities of the program and one or more server modules that provide processing necessary to implement the program's requirements.
  - If we use a relatively large number of cohesive small server modules it is quite likely that we will be able to reuse some of the lower level modules in other programs we develop.
  - An executive module usually is composed of a single file containing manual page, maintenance page, and implementation.
  - A server module is composed of two files
    - header file with manual and maintenance pages
    - implementation file with function bodies and test stub

# Hypothetical Module Diagram



UML Notation

50
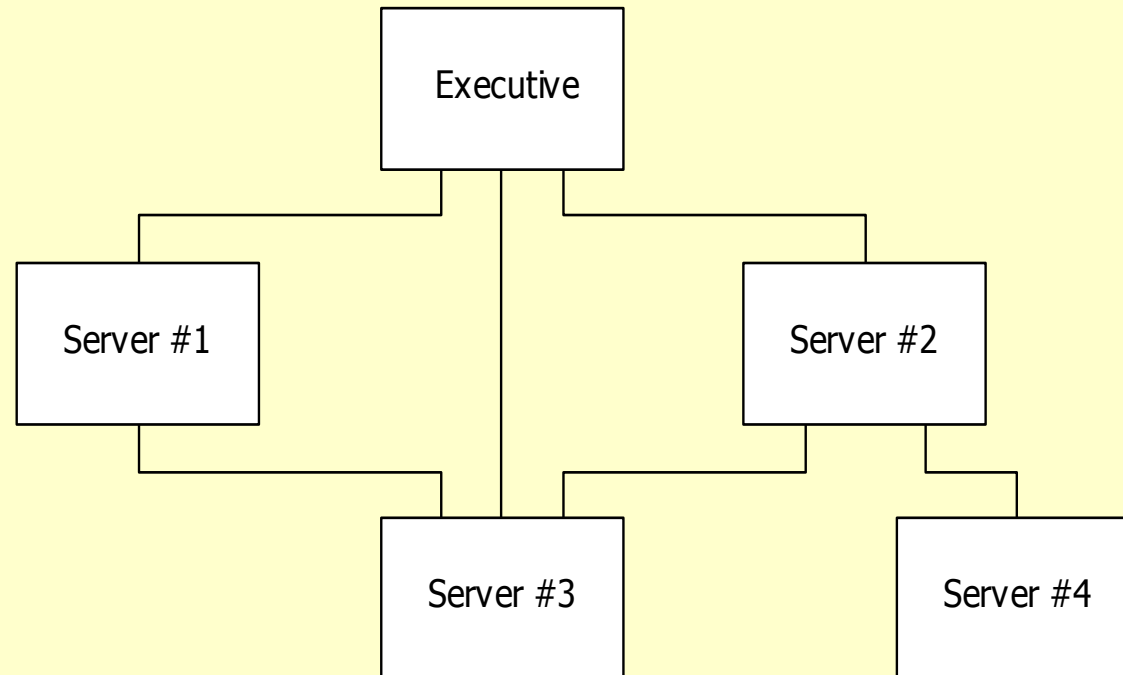
# Package Diagram

- Package Diagrams serve the same purpose as Module Diagrams. They represent the calling relationships between packages.
    - Package is a synonym for Module, e.g., means the same thing
- Their structure is a bit more free form however. Instead of showing calling relationships by above/below positioning, the package diagram uses arrows to show the direction of the call.
- The package icon is slightly different as well, as shown on the next slide.
- I use module diagrams when there are just a few – the diagram is simpler.
- I use package diagrams when there are many packages. It's easier to draw the diagram accurately.
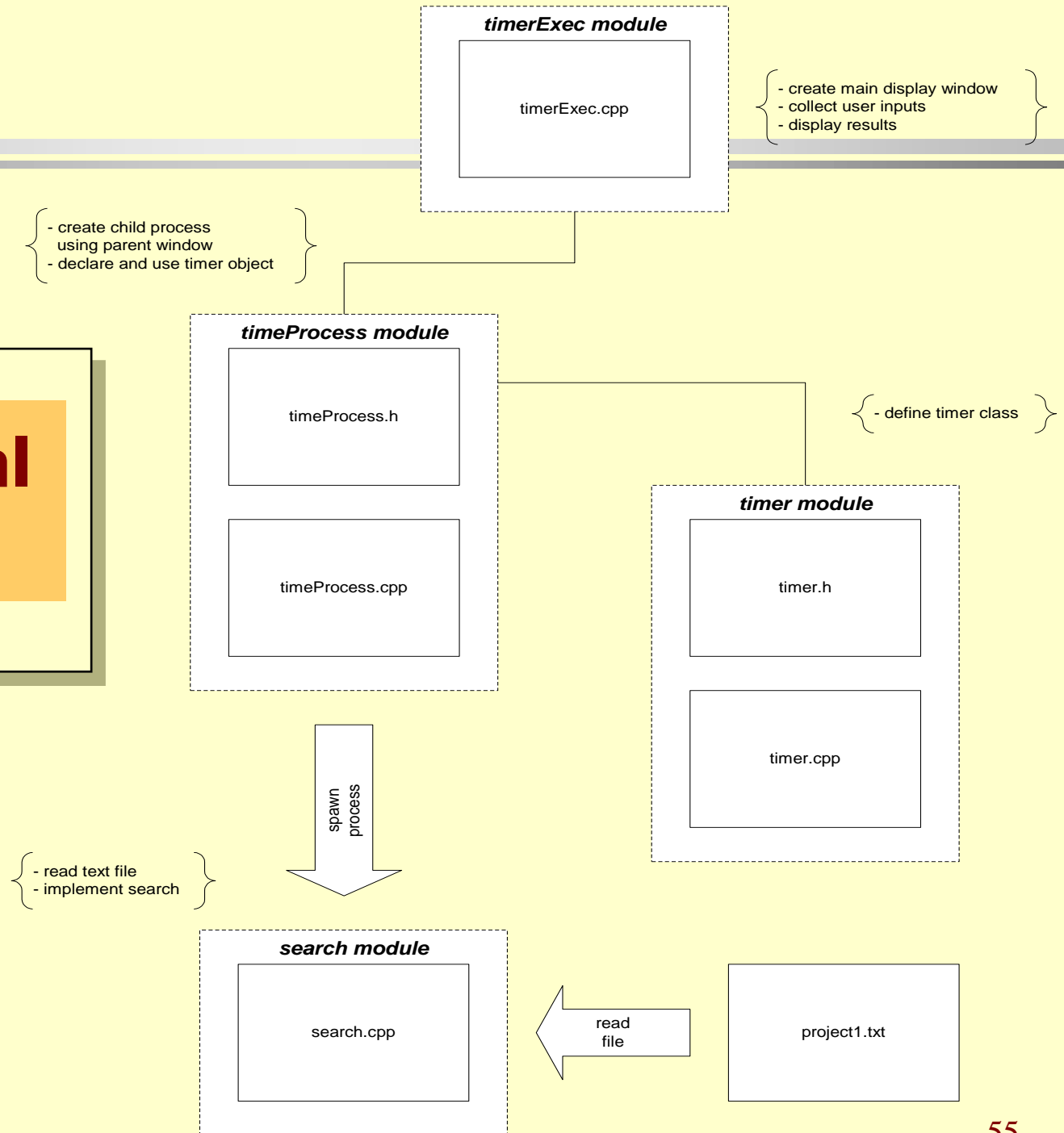
# **Package Diagram** <span>Contents</span>

# Package Diagram for Real System



Peer-To-Peer File Synchronization - Package Diagram
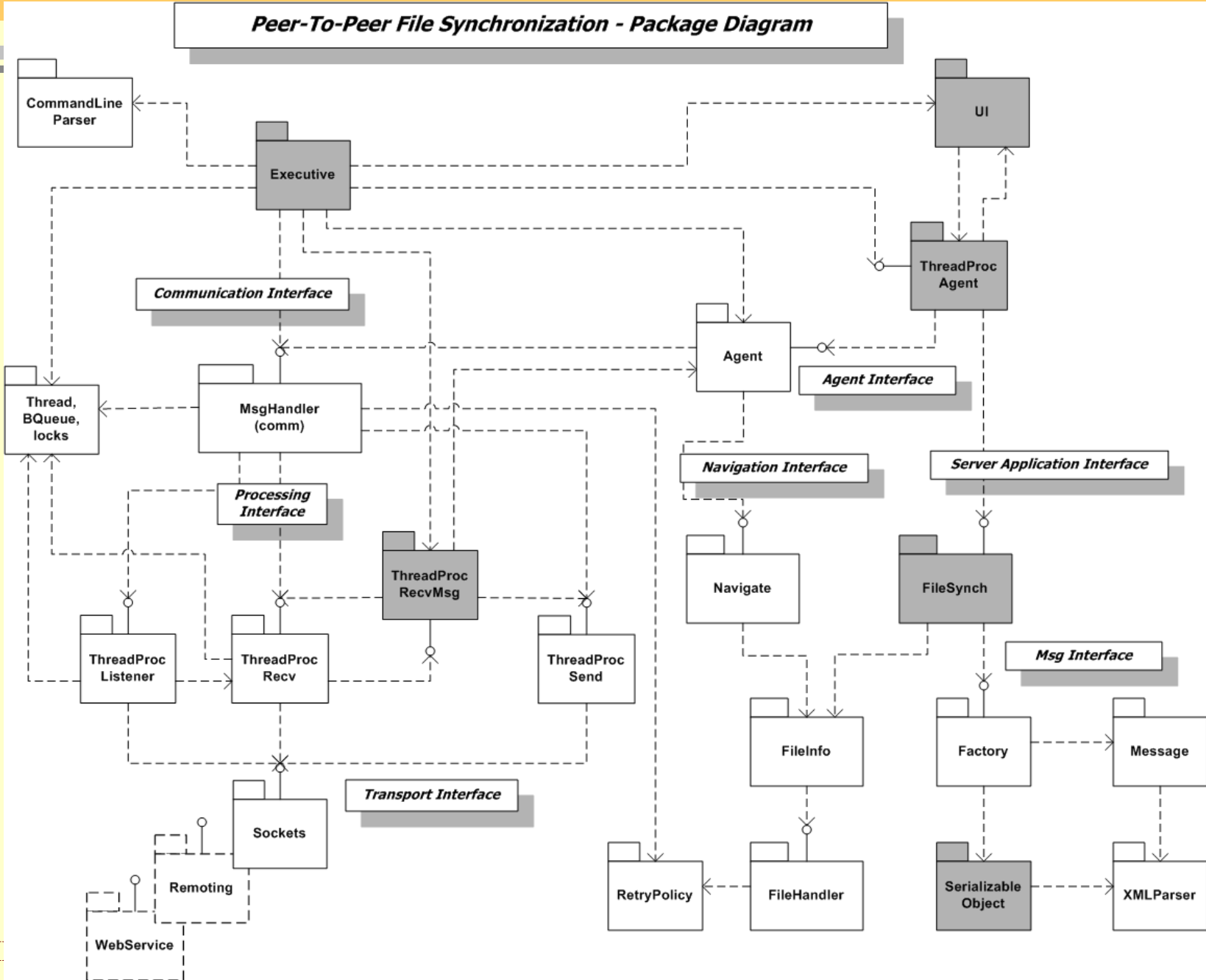
# Architectural Diagram

- For some programs we may wish to provide additional details in the module diagram.

  - If we use code generators like the Microsoft Founctation Classes (MFC) and resource editors (Visual C++ IDE) some files will be generated which do not fit nicely in the standard modular structure, e.g., resource headers and scripts. In this case we may wish to show these additional files on an extended Module Diagram that we shall call an Architectural Diagram.

  - This diagram is a module diagram to which we add the generated files and may annotate with brief statements about processing required of each component.

# Architectural Diagram

**timerExec module**

timerExec.cpp

- create main display window
- collect user inputs
- display results

- create child process
  using parent window
- declare and use timer object

**timeProcess module**

timeProcess.h

timeProcess.cpp

- define timer class

**timer module**

timer.h

timer.cpp

spawn process

- read text file
- implement search

**search module**

search.cpp

read file

project1.txt

# Structure Chart

- The structure chart shows calling relationships between every function in a module and calls made into and out of the module.

  - Callers are <u>always</u> shown above callees.

  - Lines without arrow heads are drawn from the caller to the callee.

  - All data flowing between the invoking and invoked function are shown with labeled arrows.

  - These arrows are called data couples and are usually labeled with the name shown in the argument list of the called function.

  - If a control signal is passed between functions it is shown with a hollow ball.  Note however, that what one function may consider data another function may consider control, e.g., used to make a decision.  If in doubt about how to draw a couple show it as data.

  - Recursive calls or calls which would result in many crossing lines are shown with lettered circles instead.

  Often one Structure Chart is made for each module in a program.  The gray boxes are calls to, or by, functions outside the module.
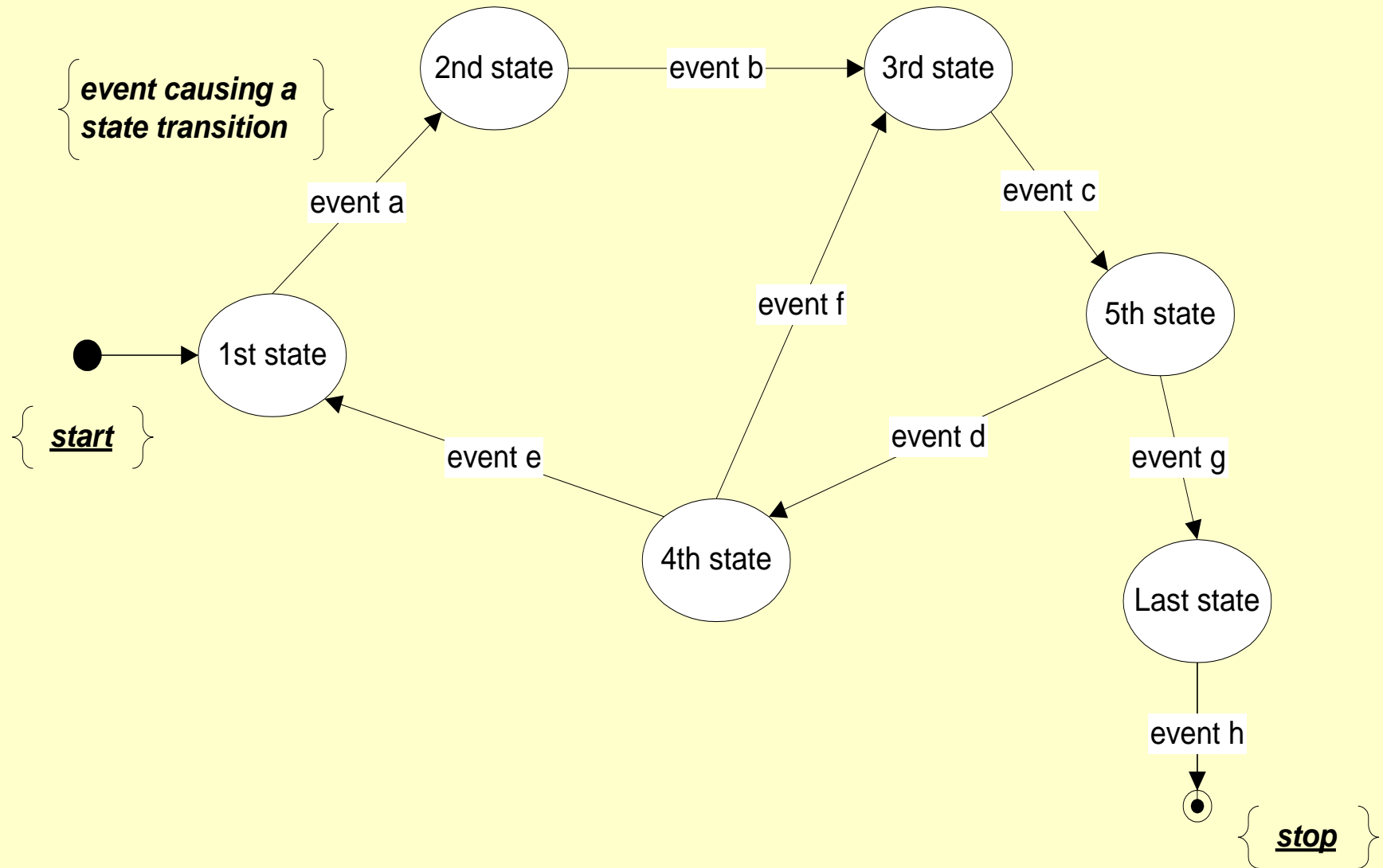
# Structure Chart Contents
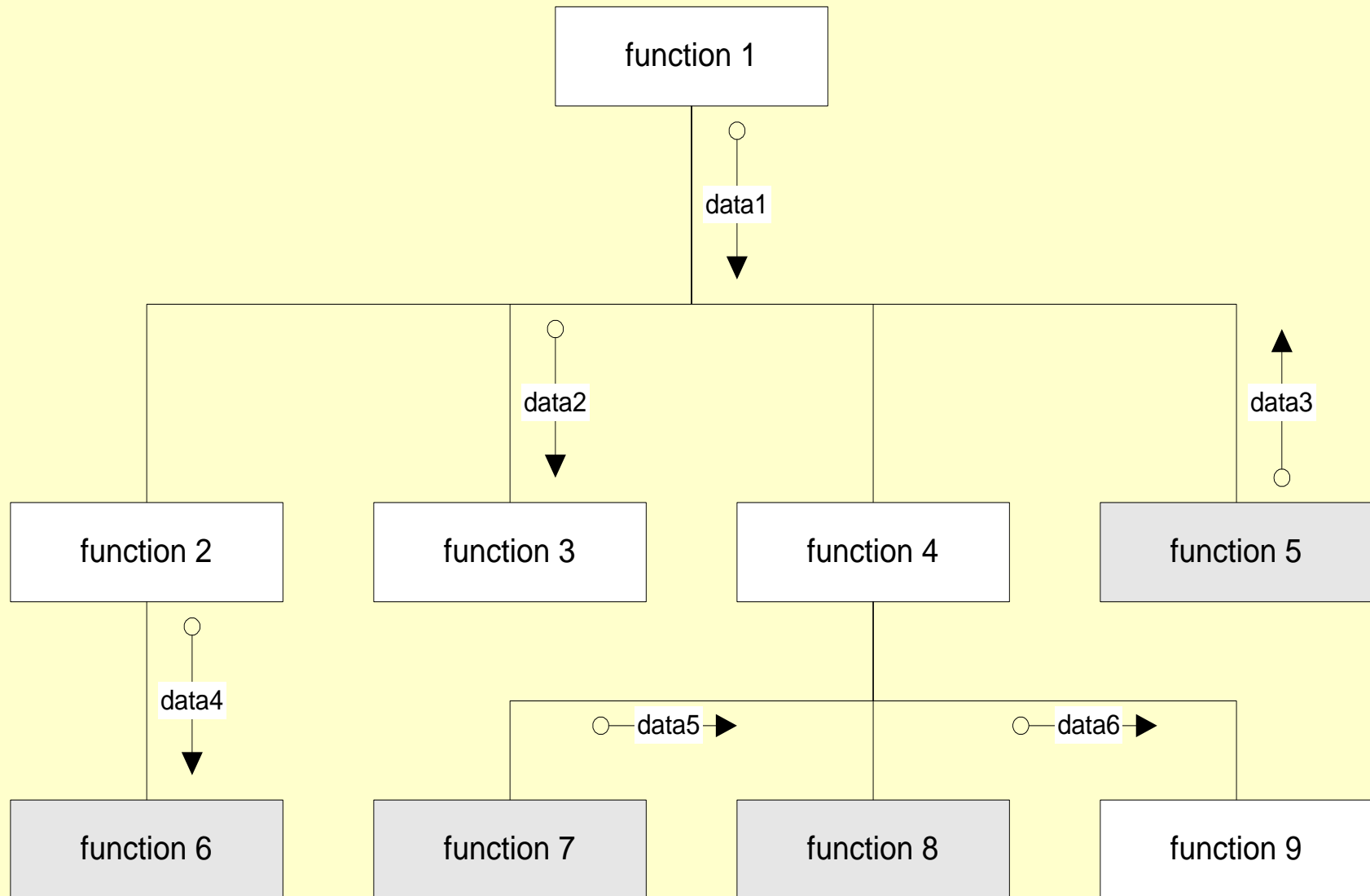
# An Example Structure Chart <u>Contents</u>
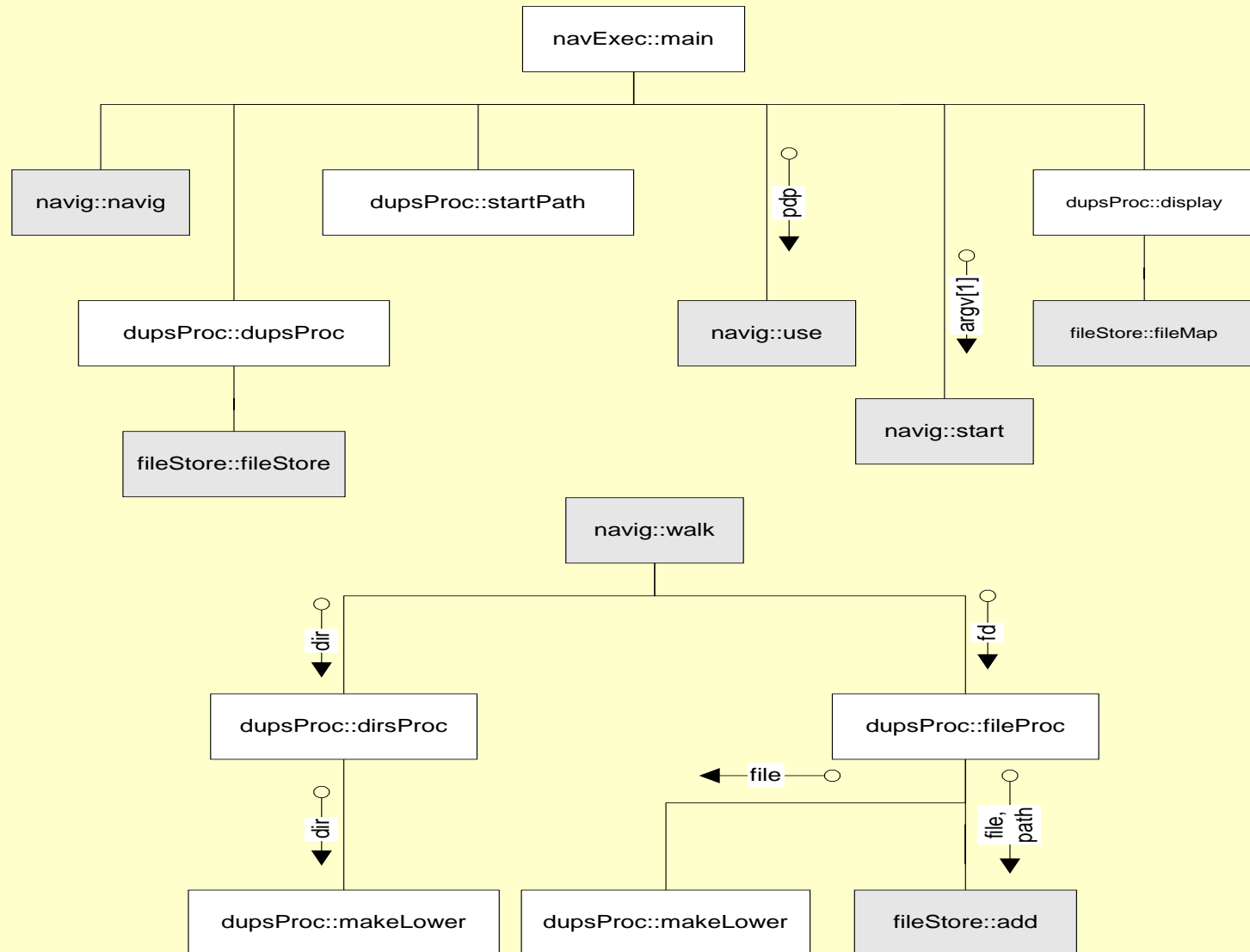
# State Diagram

- A State diagram shows the dynamic behavior of a finite state machine.  Programs which incorporate language grammar processing or controller activities are often represented by state diagrams.

  - A state diagram contains a set of labeled bubbles, one for each state of the machine.

  - Labeled lines are drawn between states showing transitions from state to state.  The labels indicate the event that triggered a transition from the source state to the destination state.

  - start and terminal states are shown with filled circles.

- In a sense, state diagrams are activity diagrams where the transition conditions have been emphasized and no synchronization or parallel activities are shown.

# State Diagram <span>Contents</span>



*event causing a state transition*

event a

event b

event c

event f

event d

5th state

event e

event g

2nd state

3rd state

1st state

*start*

4th state

Last state

event h

*stop*

# State Transition Diagrams

State transition diagrams are usually used to represent low level design details, particularly when representing the processing of a grammar.

I have used them to represent the operation of a tokenizer and to show how code analysis grammar works.

# State Diagram Example

- This diagram represents processing required to analyze C or C++ source code, looking for function definitions, class or struct declarations, typedefs, and global data declarations.
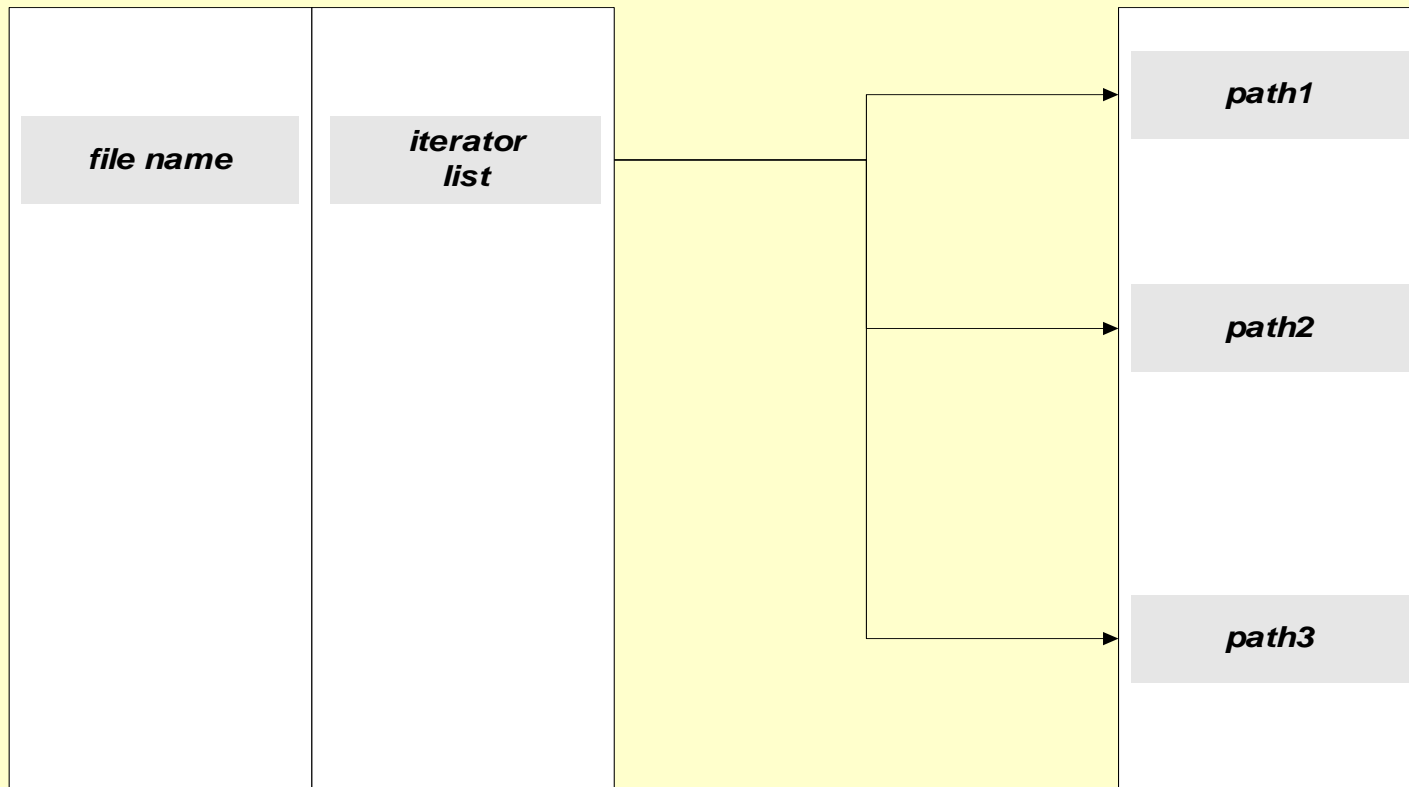


UML Notation
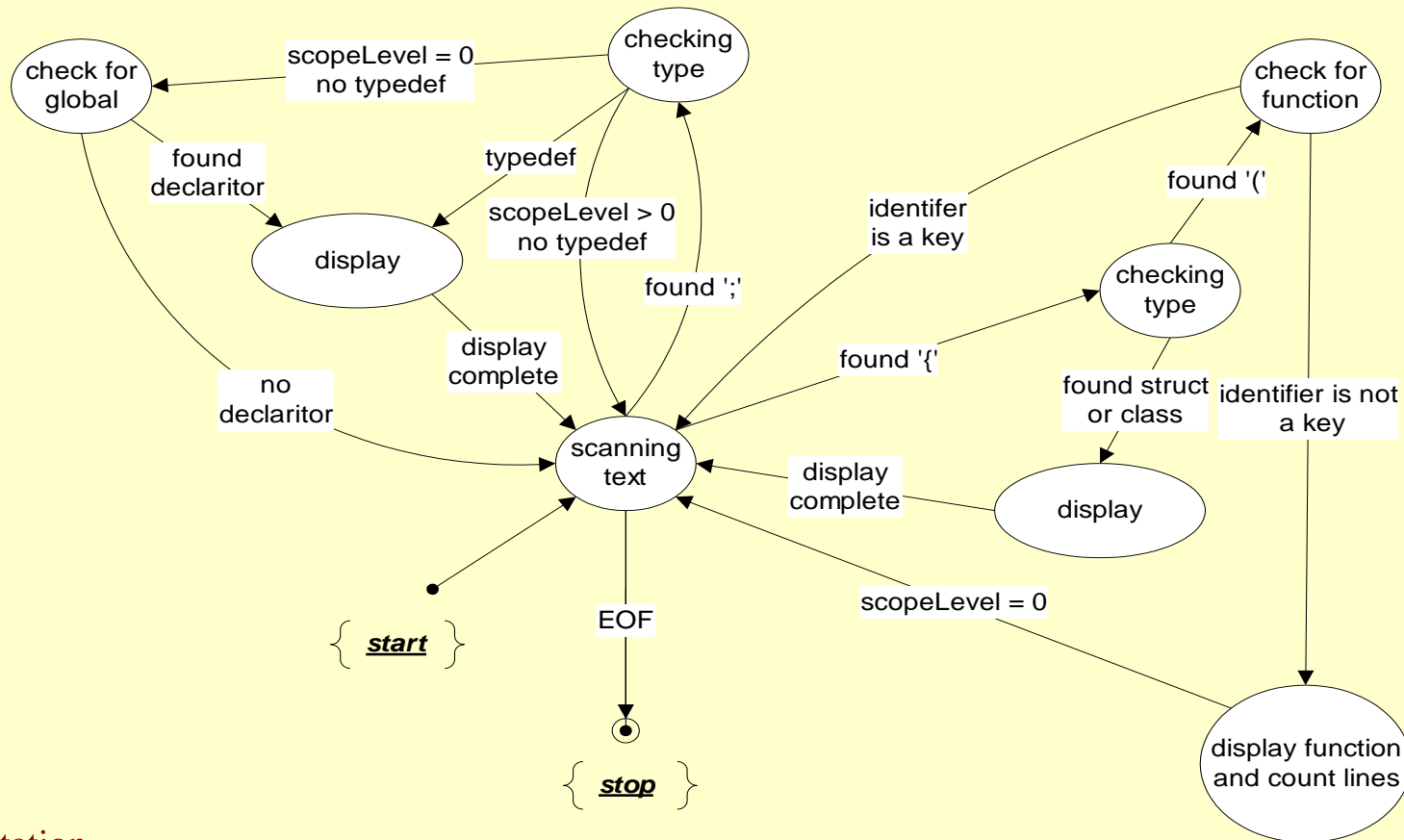
# Data Structure Diagrams Contents

- Data structure diagrams have no special syntax.
  - Their structure is defined to show the layout and relationships between data items in a program.
  - There are diagrams used for data base design called entity-relationship diagrams which do have a syntax formalism. We shall not be concerned with them in this course.
- Data Structure diagrams are often used to document the design of modules and classes which manage complex data for a program.

# Duplicates Program
# Data Structure Diagram <span style="color:blue;text-decoration:underline">Contents</span>

*typedef map< string, list<pathSet::iterator> > fileMap*     *typedef set< string> pathSet*

| file name | iterator list |
|---|---|

path1

path2

path3

Note:     Only one copy of each file name is stored in the fileMap
          and only one copy of each path is stored in the pathSet.

UML Notation                                                                              64

# End of Presentation