Documenting Architecture and Design

Jim Fawcett

Copyright © 1999-2003

CSE687 – Object Oriented Design

Spring 2003

Levels of Documentation

- Operational Concept Document
  - Overall view of architecture of large, complex system
- Software Specification
  - Complete, unambiguous, consise description of a component's obligations.
- Design Document
  - "As built" description of the means used to design and implement a component.
- Test Report
  - Detailed description of component or system test methods and results.
- Code documentation:
  - Manual page, maintenance page, design notes, prologues, comments

What is Software Architecture?

- "An architecture is the set of significant decisions about the organization of a software system, the selection of the structural elements and their interfaces … together with their behavior as specified in the collaborations among those elements, …"[1]
- "… abstract away some information from the system … and yet provide enough information to be a basis for analysis, decision making, and hence risk reduction."[2]
- "…designing and specifying the overall system structure emerges as a new kind of problem. Structural issues include gross organization and global control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance; and selection among design alternatives."[3]

References

1. Booch, Rumbaugh, and Jacobson, The UML Modeling Language User Guide, Addison-Wesley, 1999.
2. Bass, Clements, and Kazman. Software Architecture in Practice, Addison-Wesley 1997.
3. Mary Shaw and David Garlan. An introduction to software architecture. In V. Ambriola an G. Tortora, editor, Advances in Software Engineering and Knowledge Engineering, volume I. World Scientific Publishing Company, 1993.

What is Software Architecture?

- The architecture of a software system captures major features and design ideas for a software development project.
  - Describes relationship of users with the system.
  - Describes structure and organizing principles of the system.
    - major partitions within the system and their interfaces
    - responsibilities of, and resources needed by, each partition
    - design concepts: data structures, algorithms, data flows, that help developers understand and implement their piece of the system.
  - Identifies major threads of execution
    - A thread is the sequence of activities that result from some system event. Examples are system startup, response to operator requests, and processing of errors.
  - Identities critical time-lines and risk areas
    - A time-line is a time-based budget for critical threads.
    - A risk area identifies objectives and requirements that will be difficult to meet under the current architectural and design concept.

Architectural Issues

- Software architecture is concerned with:
  - ***Goals:***
    - main objectives of the system
  - ***Uses:***
    - how people and other software will interact with the system
  - ***Tasks:***
    - activities for system and its major partitions
  - ***Partitions:***
    - modules and objects that make up the system
  - ***Interactions:***
    - the relationships, data flows, and assumptions that partitions have about each other
  - ***Events:***
    - any occurrence that affects system activities
  - ***Views:***
    - appearance of the system to users and its designers
  - ***Performance:***
    - Efficient use of computer resources – processor cycles, network bandwidth, memory

Uses

- Uses describe the way users and other software components interact with the system.
  - What is the user trying to accomplish?
  - What are the required inputs that the user supplies?
  - What are the system outputs that the user expects?
  - What controls will the user want to affect system operation?

- Uses are often developed as scenarios, called use cases.
  - Each scenario describes on or more of the following:
    - User roles, e.g., developer, manager, quality assurance, …
    - Mode of operation, e.g., data collection, data analysis, data presentation.
    - Responses to specific important events, e.g., initialization, user inputs, computational errors, system output.

Tasks

- Tasks are a high level list of the activities that the system will need to carry out.
  - First developed for the system as a whole.
  - Later, allocated to the major system partitions.

- Tasks are usually presented as lists and in activity diagrams.
  - Activity diagrams are like flow charts, but at a higher level.
  - They describe activities that are important for the system or its major partitions.
  - Activity diagrams show required sequencing and synchronization of tasks.
  - When software is implemented tasks allocated to each module are described in the modules manual page.

Partitions

- Partitions represent the grouping of system activities into logical and physical entities.
  - Data Flow Diagrams (DFDs) represent the partitioning of system activities into processes, showing the flow of information between the external environment and each process.
  - Module diagrams show the physical packaging of system processing into files.
  - Classes show the logical partitioning of system data and processing into low-level program constructs.

- Partitions are the second most important part of the architecture concept, after the definition of its tasks.
  - Sequence of development is often: (1) uses, (2) tasks, (3) partitions, (4) interactions, (5) events, and (6) views.

Interactions

- Interactions describe the relationships between system partitions. They are described by:
  - ***Data flow diagrams:***
    Used in the early phases of architecture and requirements development.
  - ***Module diagrams:***
    Describe static relationships between the system's physical partitions.
  - ***Class diagrams:***
    Describe the static relationships between the system's logical components.
  - ***Event trace diagrams:***
    Show the dynamic relationships between system components.
  - ***Structure charts:***
    Describe the relationships between the system's functions.

Events

- Events describe specific occurrences to which the system must respond, or that affect its modes of operation.
  - Events are critically important for real-time systems, e.g., systems that must respond to asynchronous events from the outside environment.
  - For these systems architecture development may revolve around the definition of critical threads.
    - A thread, as defined by the architecture concept, is all the processing that results from a specific event, e.g., a radar detection, user input, power on, computational error, etc.
    - Many threads are defined, then sorted by importance, relative to the system requirements. The architecture isn't complete until processing is defined that will support system requirements for each of the critical threads.
  - Threads are usually described by event trace diagrams.
  - In some non-real-time systems events play only a minor role in developing the system architecture.

Views

- Views are used in two ways:
  - Views describe the user interface as it appears to the user.
    - Layouts of controls and screens.
    - Screen shots showing what the user will see when entering data.
    - Screen shots showing what the user will see when observing operation.
    - Each of these views is accompanied with text describing how the user interacts with the controls and screens.
  - Views also describe the most important data structures and algorithms:
    - Data structure diagrams are ad hoc diagrams the show how data elements relate to each other.
    - Data structure may be described with xml tree views.

Performance

- Level of communication affects performance by orders of magnitude:
  - Within a process
  - Between local processes on a single machine
  - Between machines in a network
  - Between networks, e.g., across the internet
- Lazy communication:
  - Send information only when needed
  - Send only the specific information needed
- Data caching:
  - Store information locally so that it need not be requested repeatedly
- Minimize remote connectivity:
  - Connections consume threads, CPU cycles, memory
  - Make connection time least necessary to complete request, then disconnect.

What is Design?

- Design is the process of deciding how to satisfy a program's requirements.

- Design has four essential elements:
  - ***client focus***
    Concerned with how the user will interact with the program.
  - ***organizing principles***
    The main design ideas on which a program's implementation is based.
  - ***Structure***
    The physical way that code is formed for implementation.
  - ***Performance***
    How efficiently the program uses machine resources to achieve its design goals.

Structure

- ***Structure***
  the physical way that code is formed for implementation. The design issues here are:

  - the parts into which the program is divided.
  - communication required between parts.
  - ownership of system resources and other parts.
  - visibility of one part by another.

Performance

- ***Performance*** **-** Is determined by:
  - algorithms used in implementing requirements
  - how often the memory manager is called
  - how arguments of functions are passed
  - which objects are static
  - which objects are made local
  - the size of objects and the frequency of their construction and copying

Architecture and Requirements   Contents

- ***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
- ***Module diagram***
  (variant of structure chart – see below)
  One of the main diagrams used to describe software architecture.
  - shows calling dependencies between modules
- ***Architectural Diagram***
  - Like a module diagram but may show non-module files
- ***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

Contents

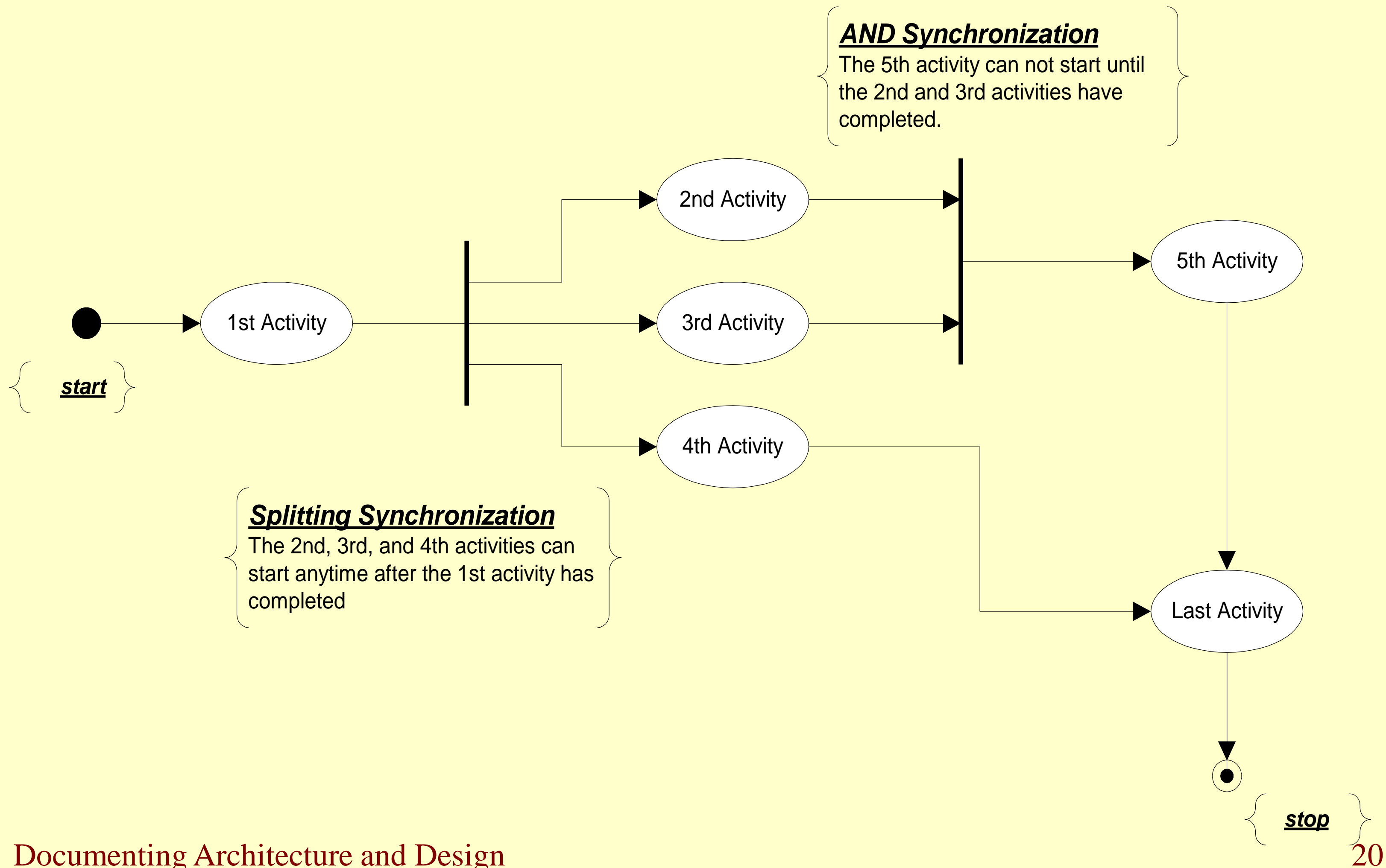
- ***Class diagram* (OMT diagram)**
  - shows classes that are used in a program along with their relationships
  - sometimes also shows their physical packaging into modules
- ***Event Trace diagram***
  - illustrates the timing of important messages (member function invocations) between objects in 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

Activity Diagram (Petri Net)

Contents

- 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 Contents



Activity Diagram   Contents

- 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.

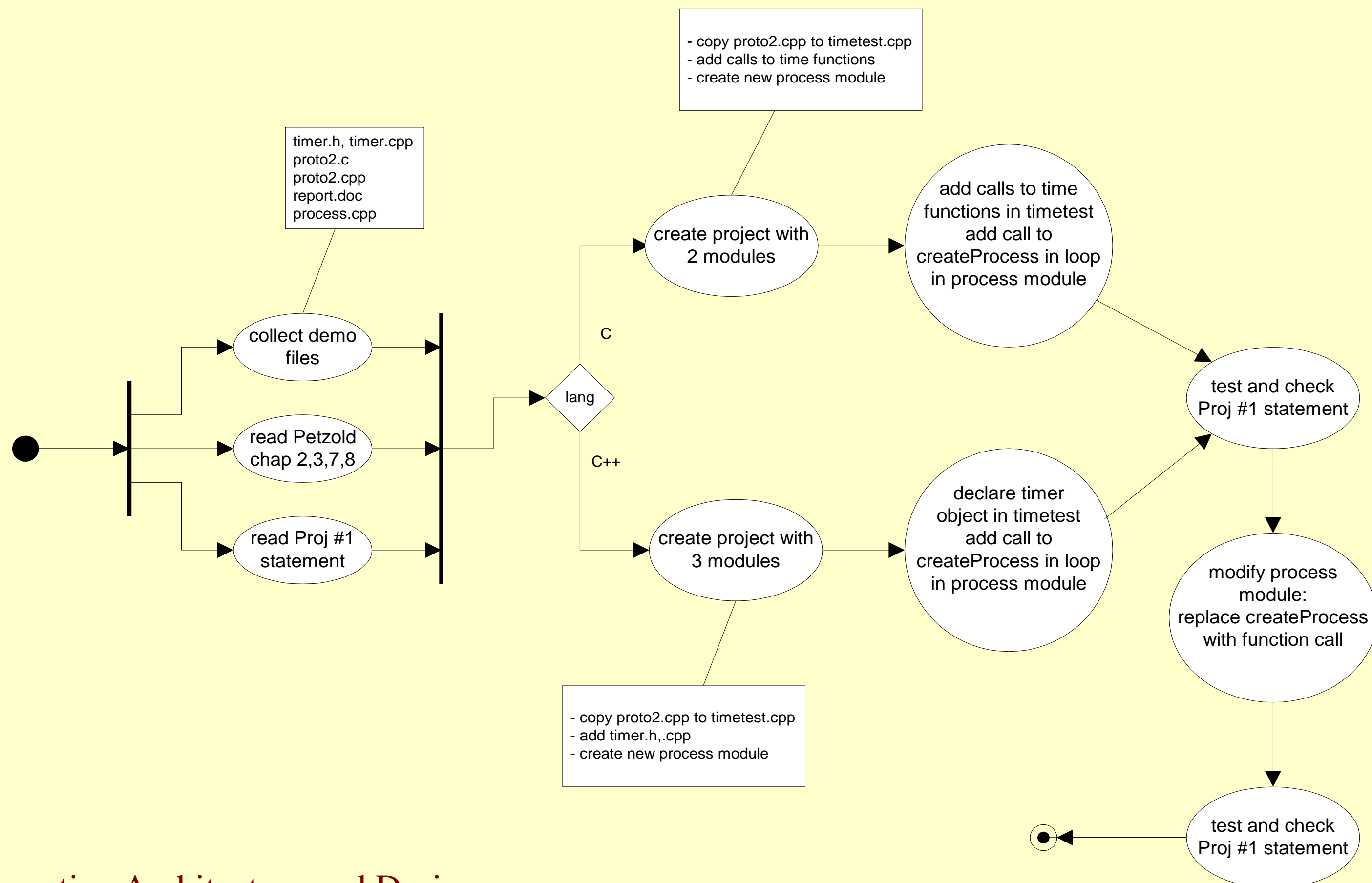
Extended Petri Nets

Contents

- 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 Contents

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

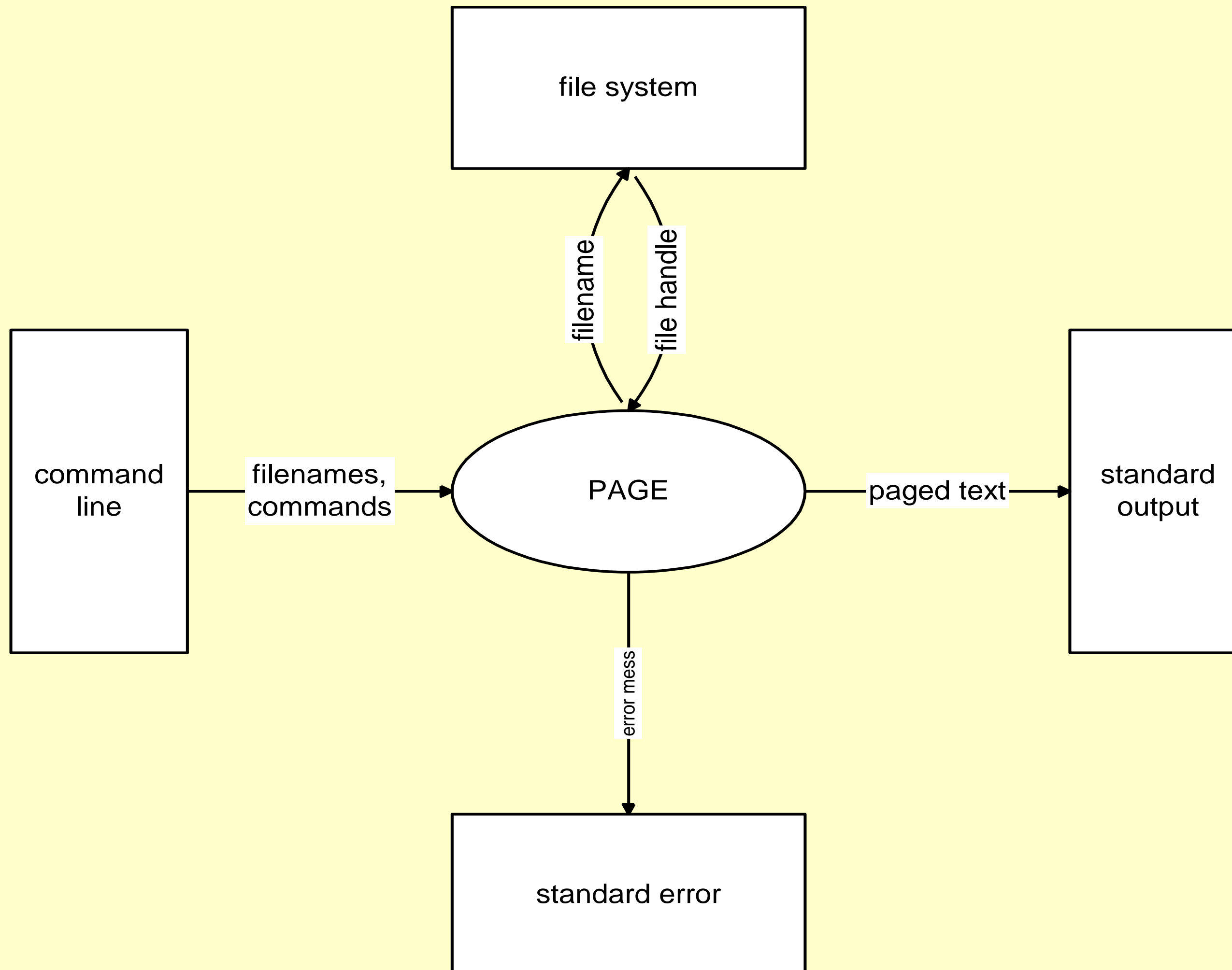


Context Diagram Contents

- 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 all 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 exactly the inputs and outputs on your top level Data Flow Diagram (DFD), described next.

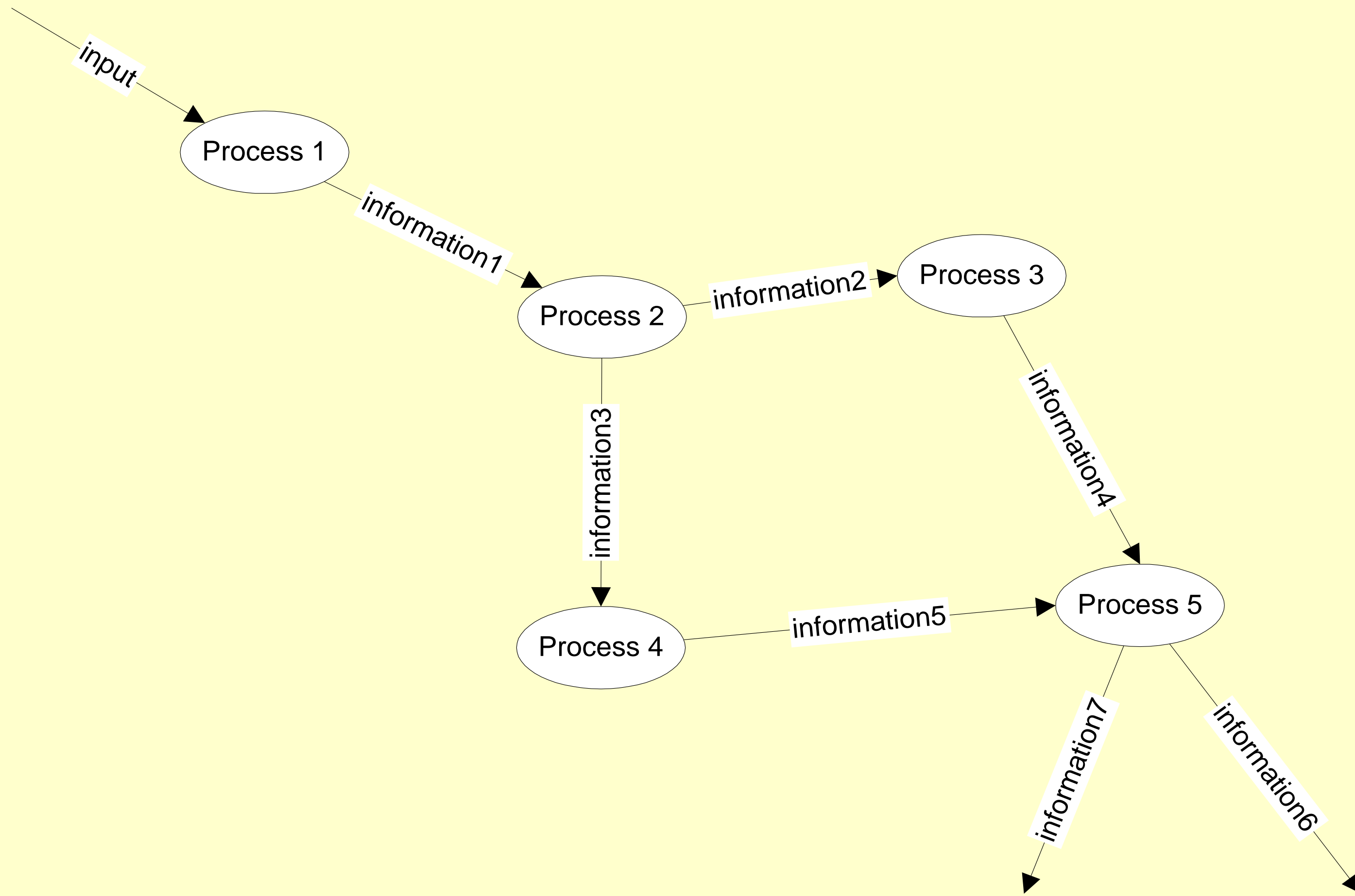
Context Diagram Contents



Data Flow Diagram Contents

- 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

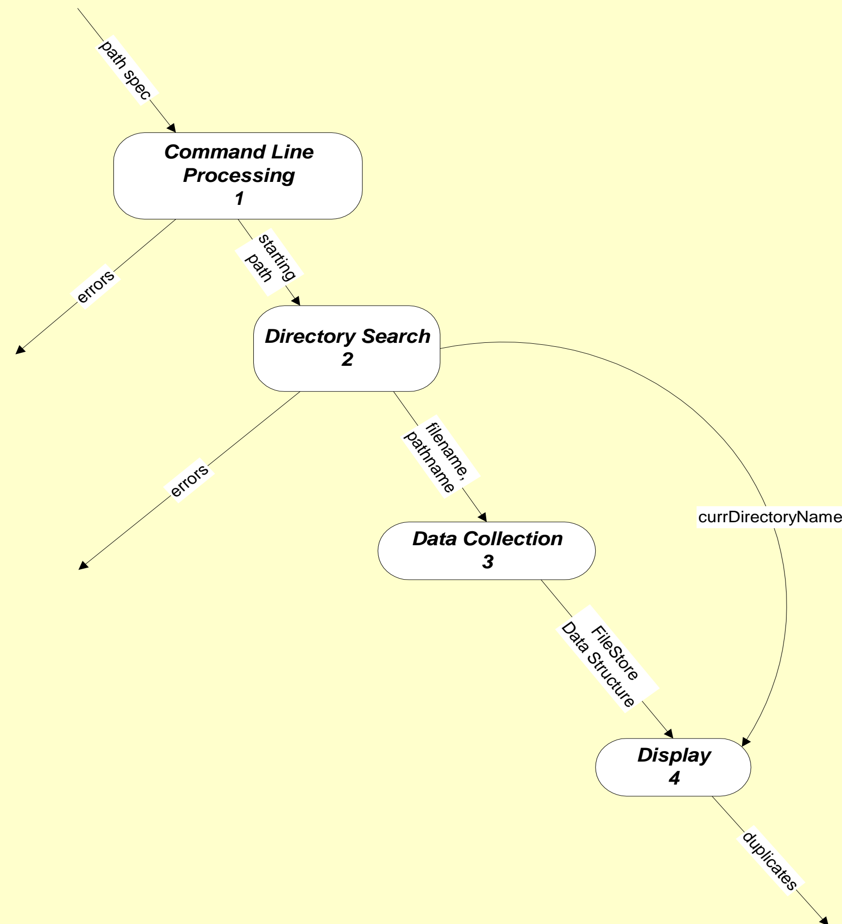
- Data Flow Diagrams (DFDs) are used during the analysis of requirements for complex systems. Each bubble represents a specific process which has been allocated tasks and requirements, so that all of the program's obligations are partitioned among the processes shown on the top level DFD.

- 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.

An Example Data Flow Diagram Contents

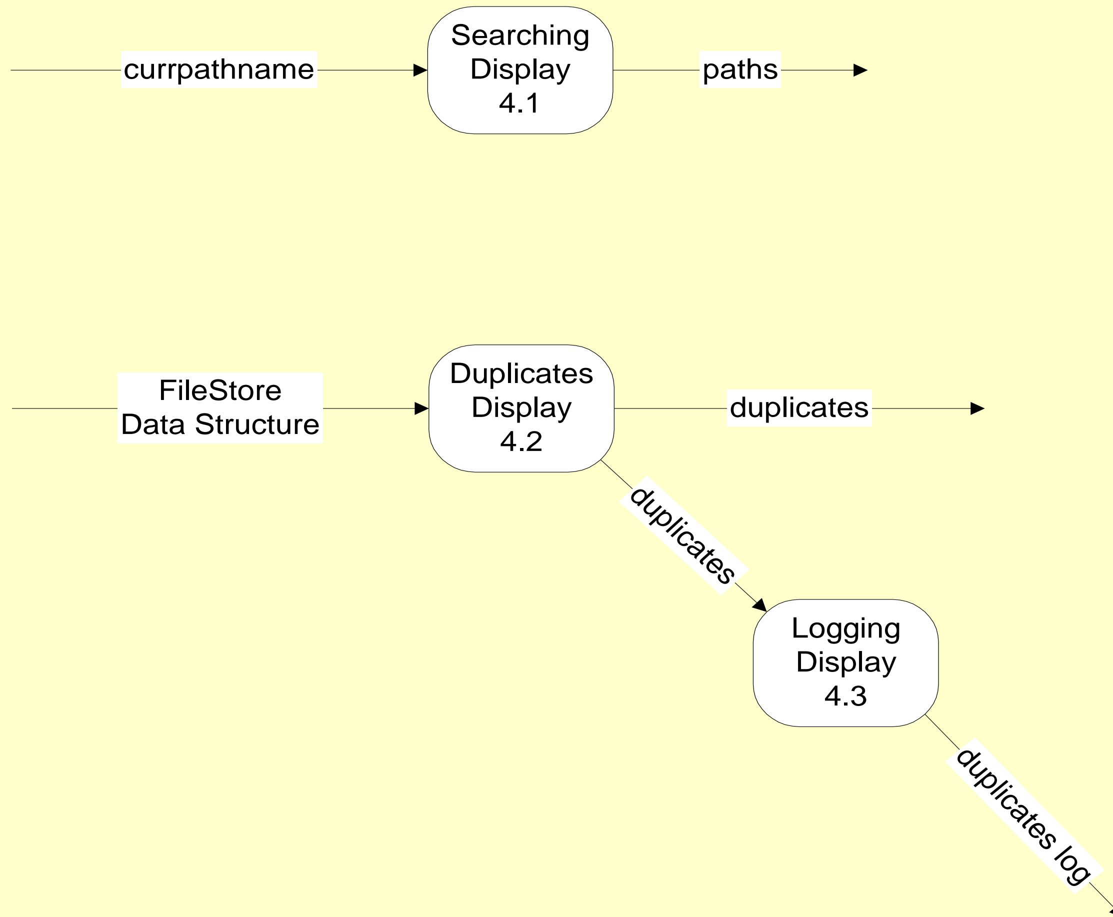
This diagram represents processing in the DUPLICATES program.



Lower Level Data Flow Diagrams [Contents]

- 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 must 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 Contents



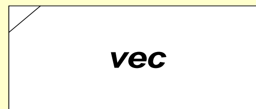
Class Diagrams [Contents]

- 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 [Contents]

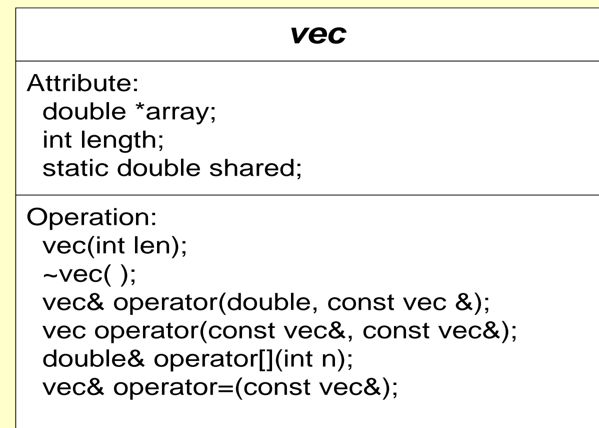
Class : a set of objects of one specific type

class symbol



object of class

class symbol with details



Objects [Contents]

Object: an element of some class

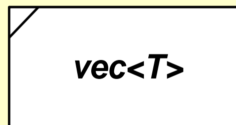
Each class represents a specific collection of data attributes of one or more types (its state) and a collection of functions (behaviors) which modify or disclose the state of an object of the class.

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.

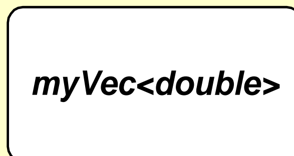
Generic Classes Contents

- 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:

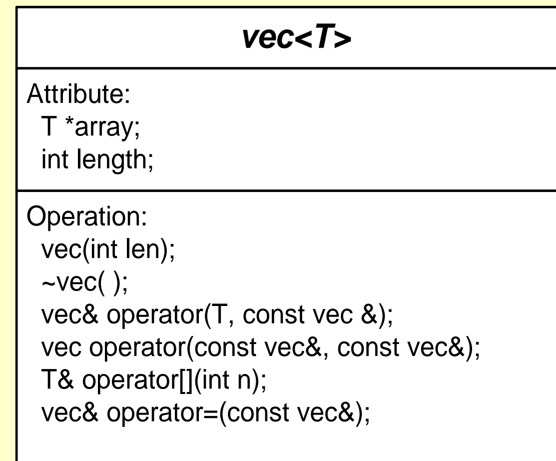
class symbol



object of class



class symbol with details



Template Class Declaration Contents

```
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
};
```

Associations Contents

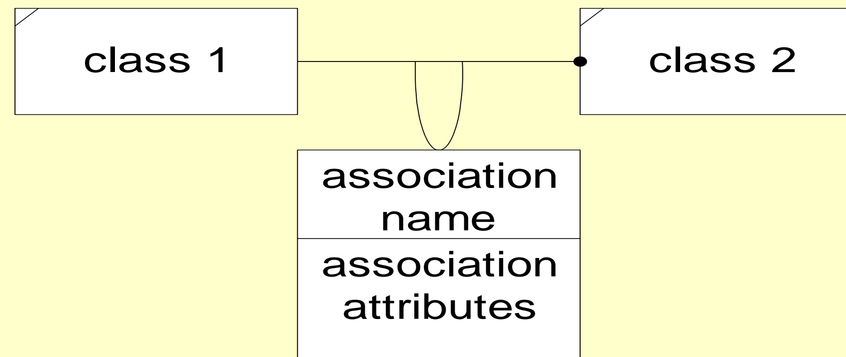
An association is a relationship linking two or more classes or objects.

Relationships [Contents]

- 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 Contents

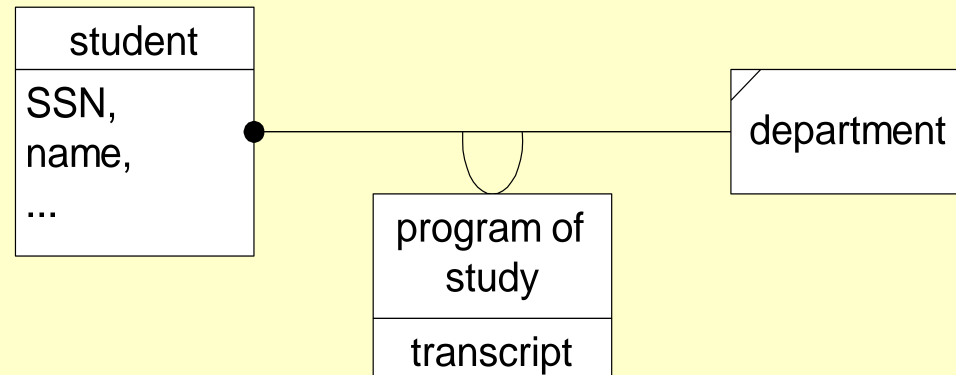
- A model may have attributes which clearly belong to the association relationship rather than to one of the classes in the association. In this case the association is given those attributes, and is denoted as shown below.



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 [Contents]

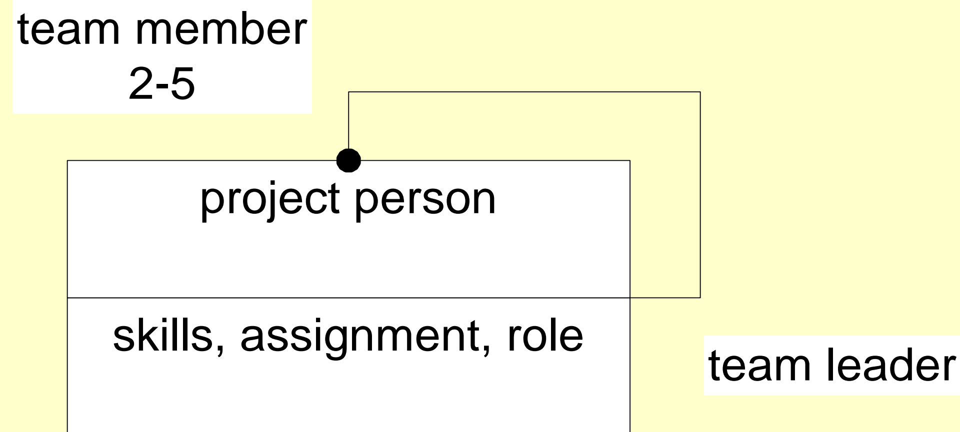
The diagram below captures the relationship between a student and her department.



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

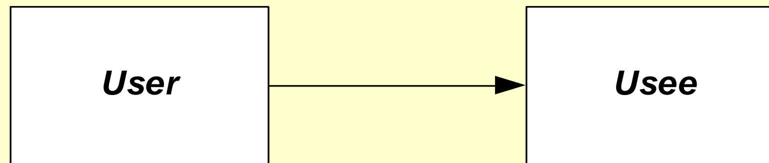
Association Example Contents

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



Using Relationship [Contents]

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

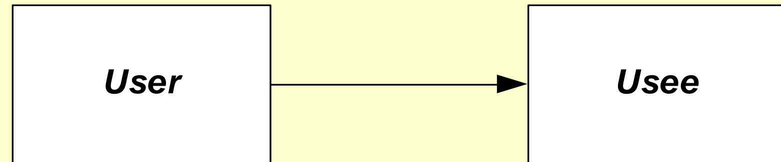


- The using relationship is implemented when a member function of the User class is passed an object of the Usee class or when it creates a local instance of the Usee class.

- 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.

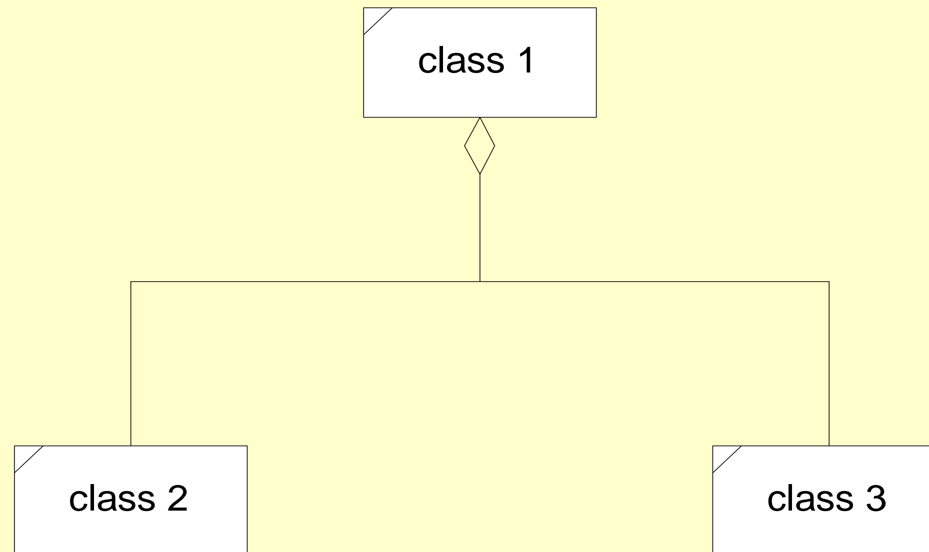
Using Example Contents

```
class User {

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

  private:
    std::string _name;
};

Void User::show(Usee &usee1) {

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



Aggregation Contents

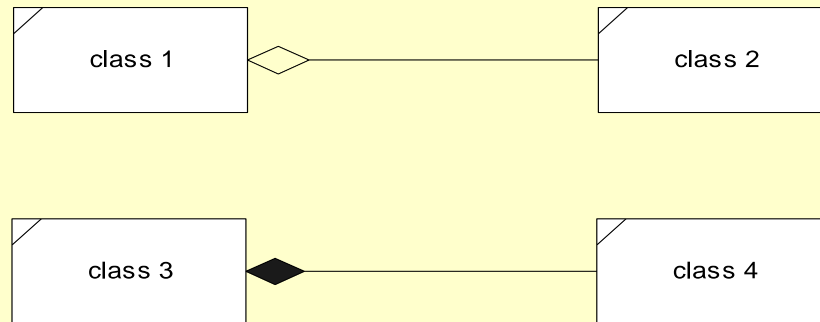
- Aggregations are special associations which model a “part-of” or “contained” semantic relationship.



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

Aggregation and Composition [Contents]

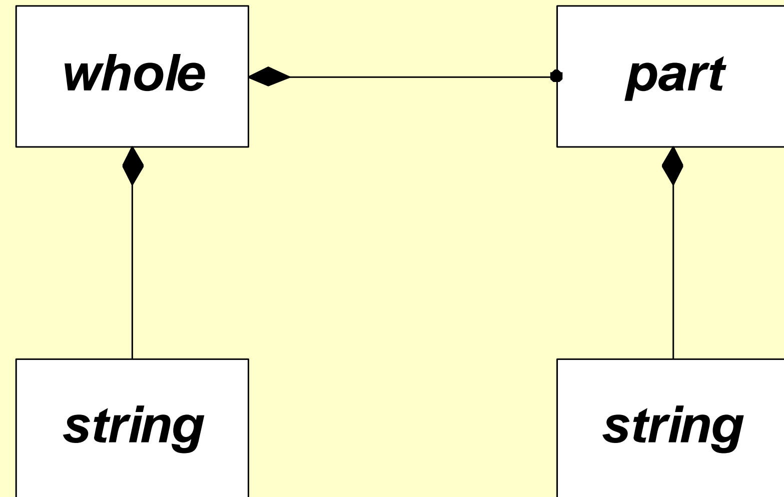
Aggregations are "part-of" or containment relationships. Here, class 2 is part of class 1. A stronger form of aggregation is the composition relationship. This is an aggregation in which the part-of represents an exclusive ownership. An owned object is created when the owner is created and is destroyed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

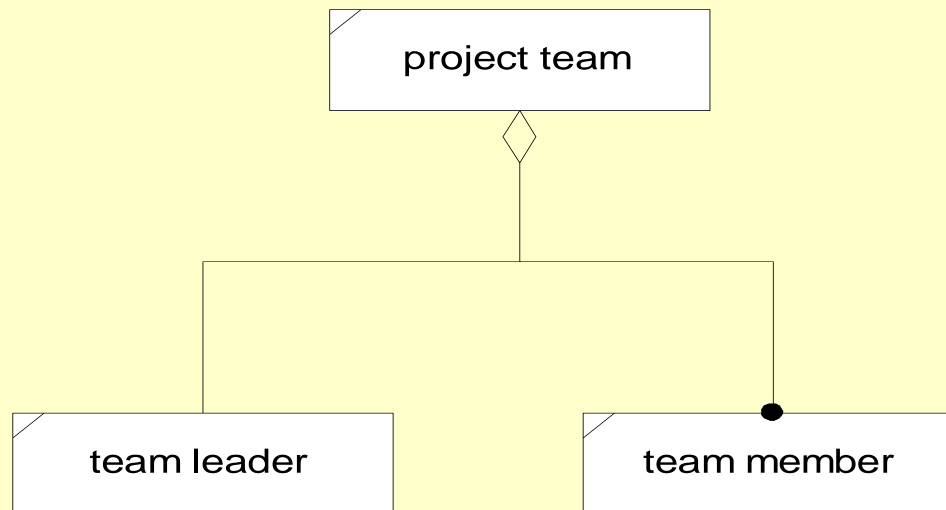
Composition Example Contents

```
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 [Contents]

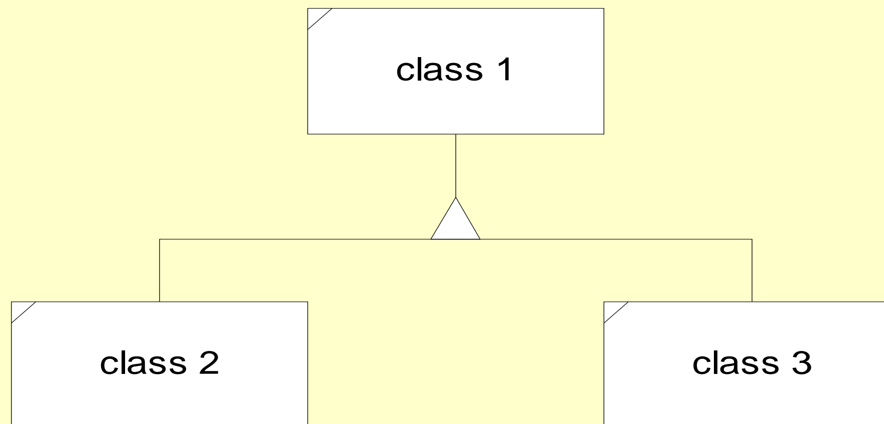
The diagram below illustrates the aggregation relationship inherent in a team.



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 [Contents]

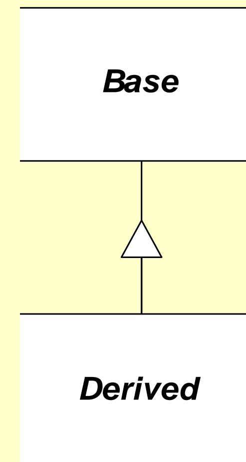
Inheritance models an "is-a" semantic relationship. Here the classes 2 and 3 inherit from class 1. We say that classes 2 and 3 are derived from base class 1.



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.

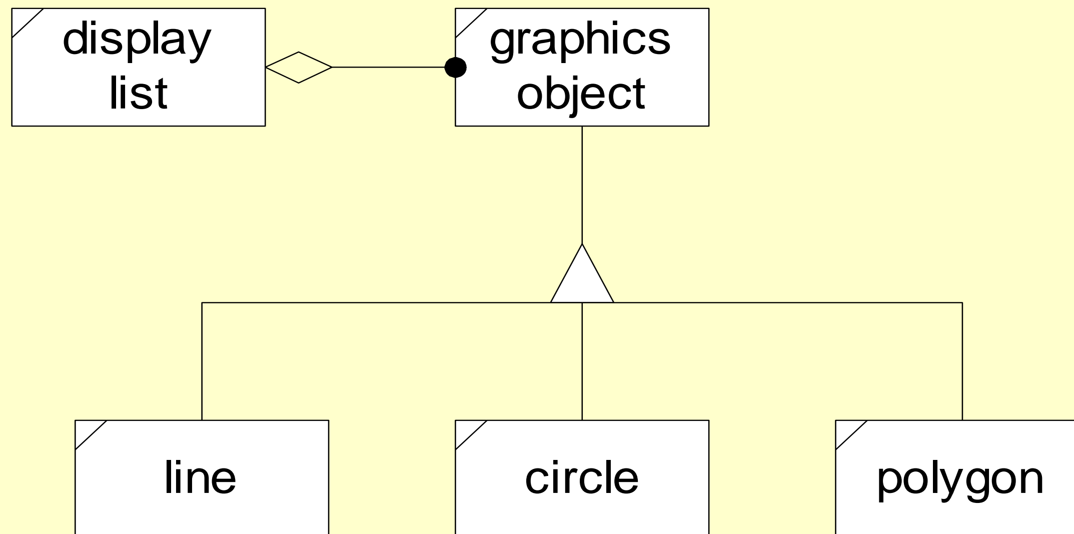
Inheritance Example Contents

```
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 [Contents]

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 [Contents]

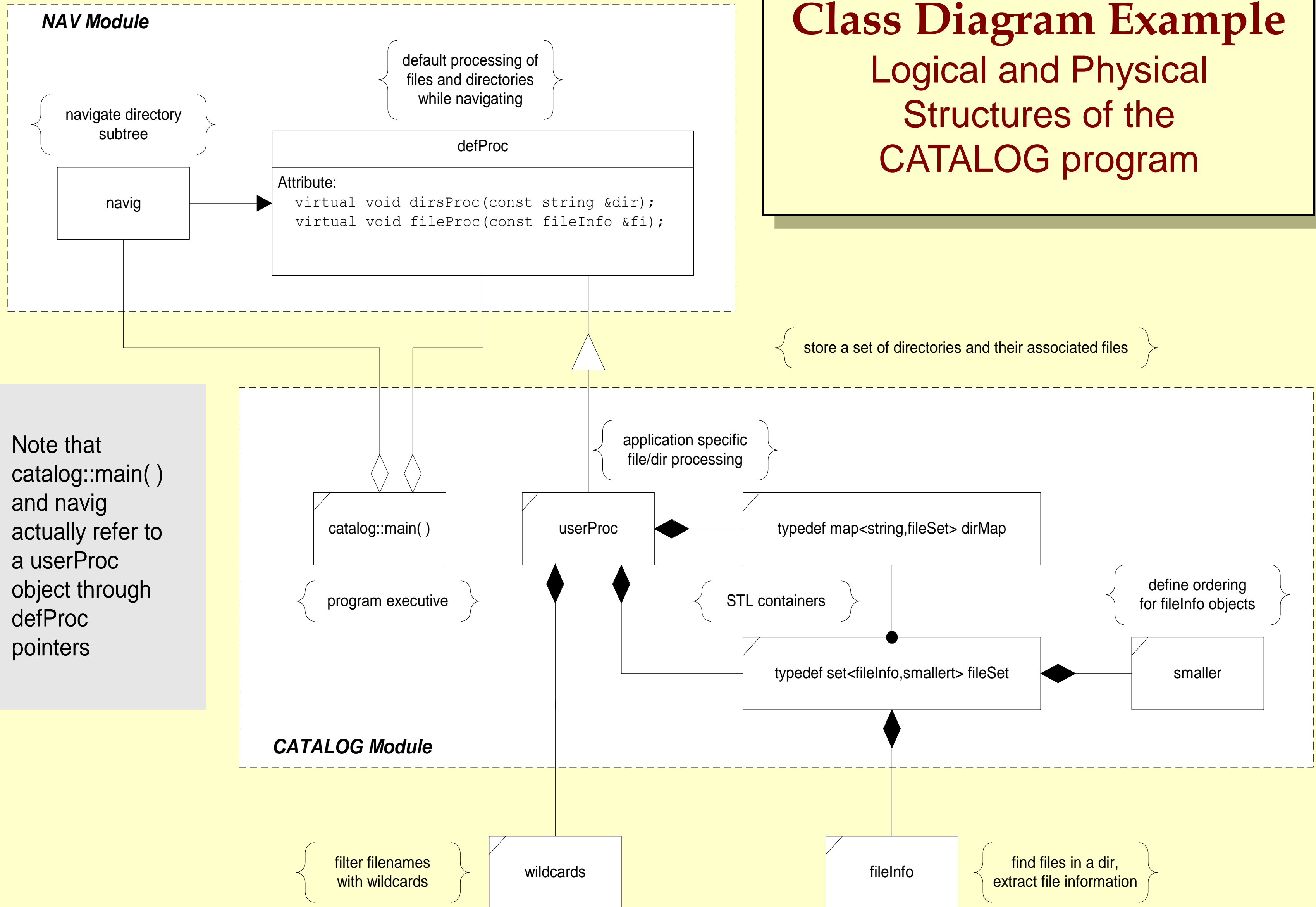
The base class graphicsObject provides a protocol for clients like the display list to use, e.g., draw(), erase(), move(), ... Clients do not need to know any of the details that distinguish one of the derived class objects from another.

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 Contents

- 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

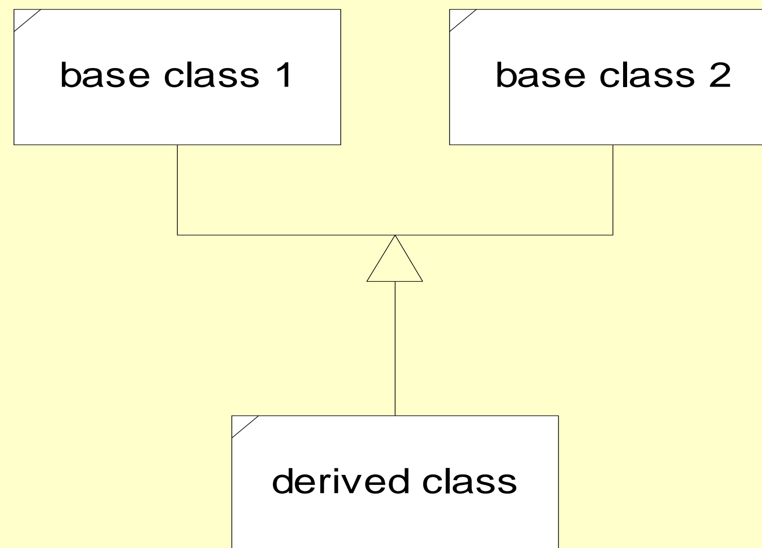


Note that catalog::main( ) and navig actually refer to a userProc object through defProc pointers

Typical Output from CATALOG

Multiple Inheritance Contents

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



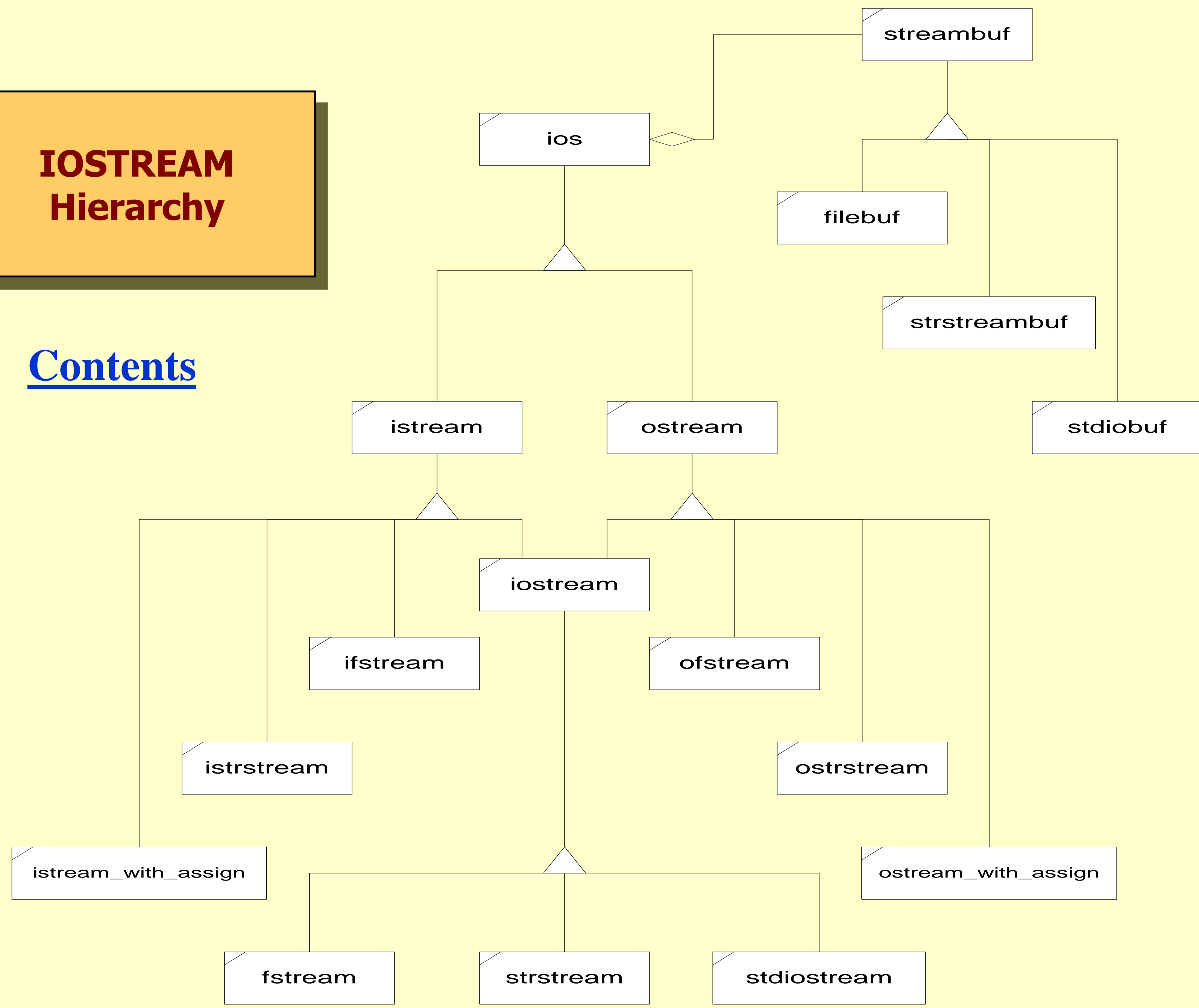
Multiple Inheritance [Contents]

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

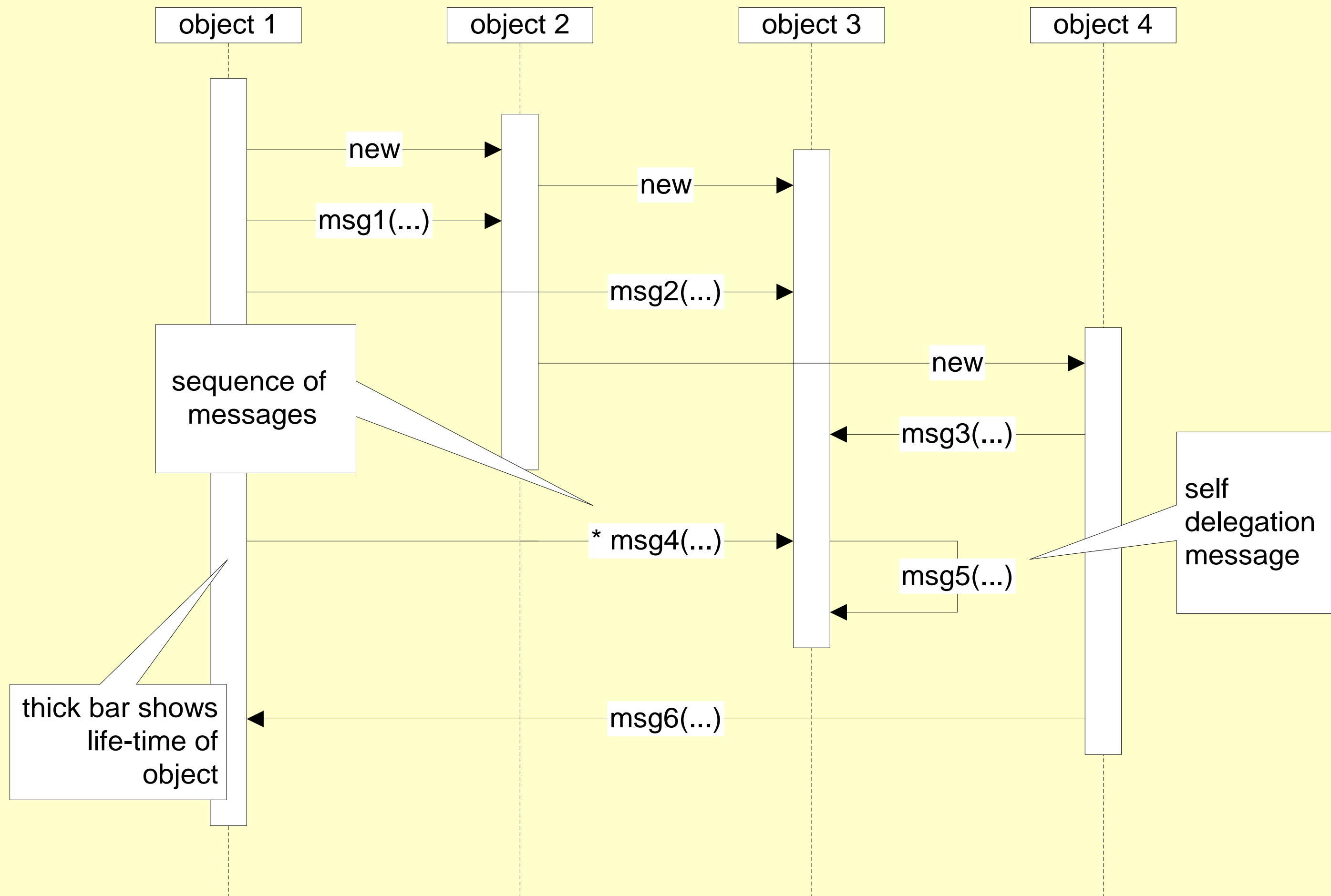
Contents



Event Trace Diagram Contents

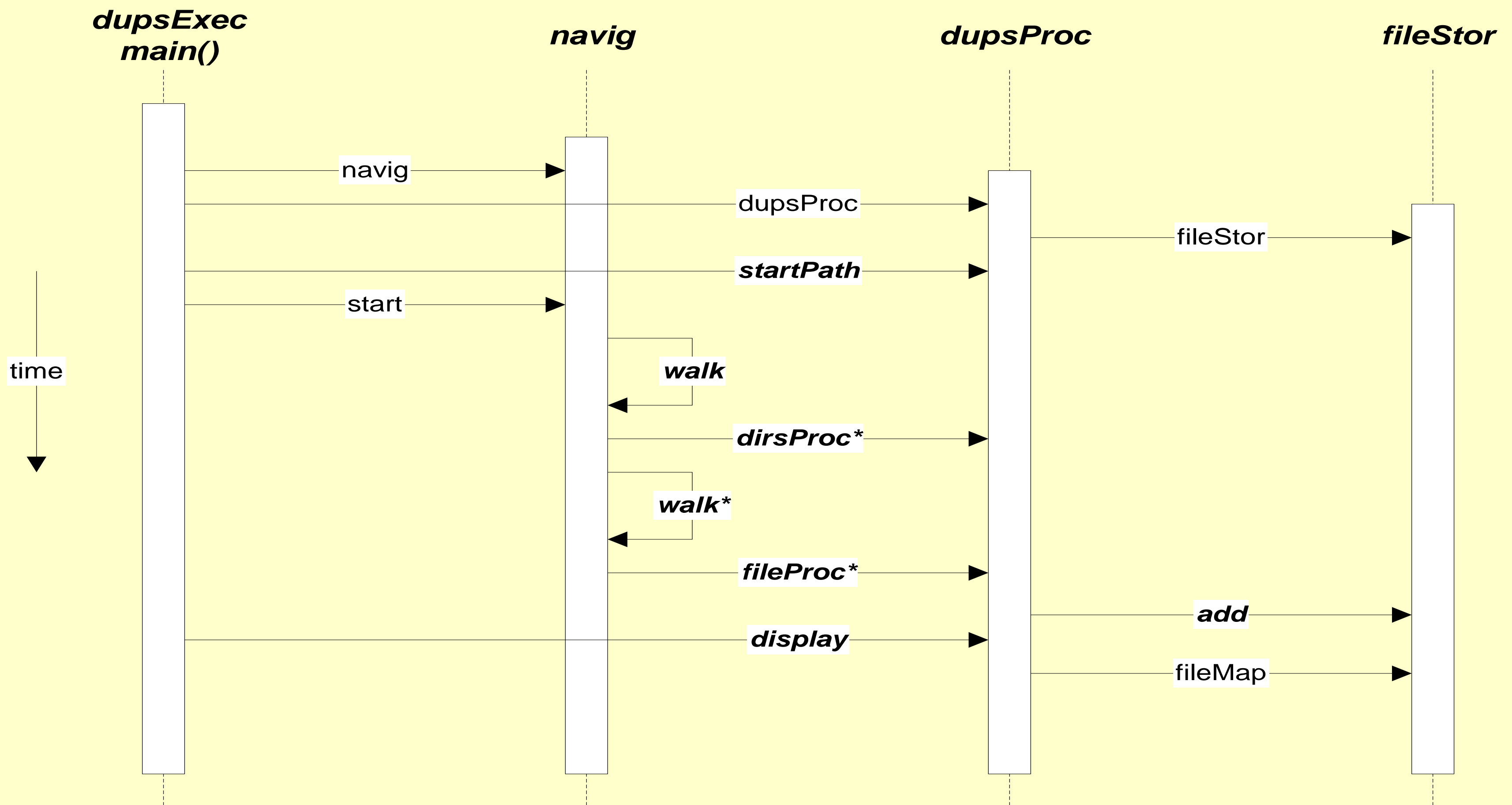
- An Event Trace diagram illustrates the timing of important messages (member function invocations) between objects in a program.
  - Each object is shown by a vertical bar
  - Message traffic is shown by labeled horizontal lines flowing toward the object on which a method was invoked.
  - Time progresses 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 Contents



Event Trace Diagram Example Contents

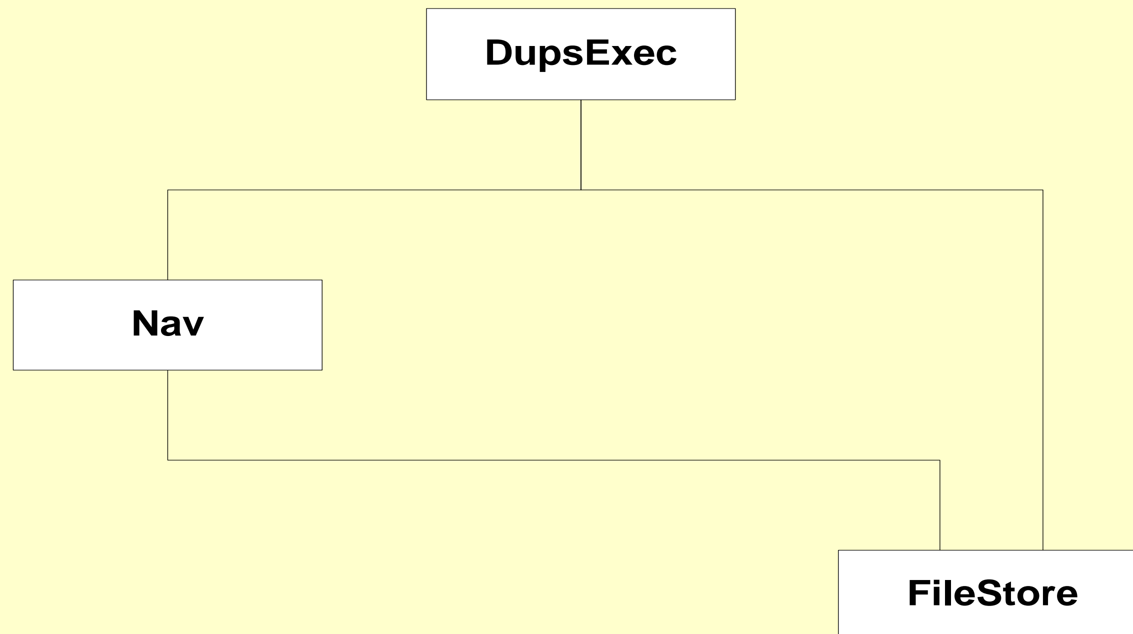
This example is from the Duplicates program.



Module Diagram [Contents]

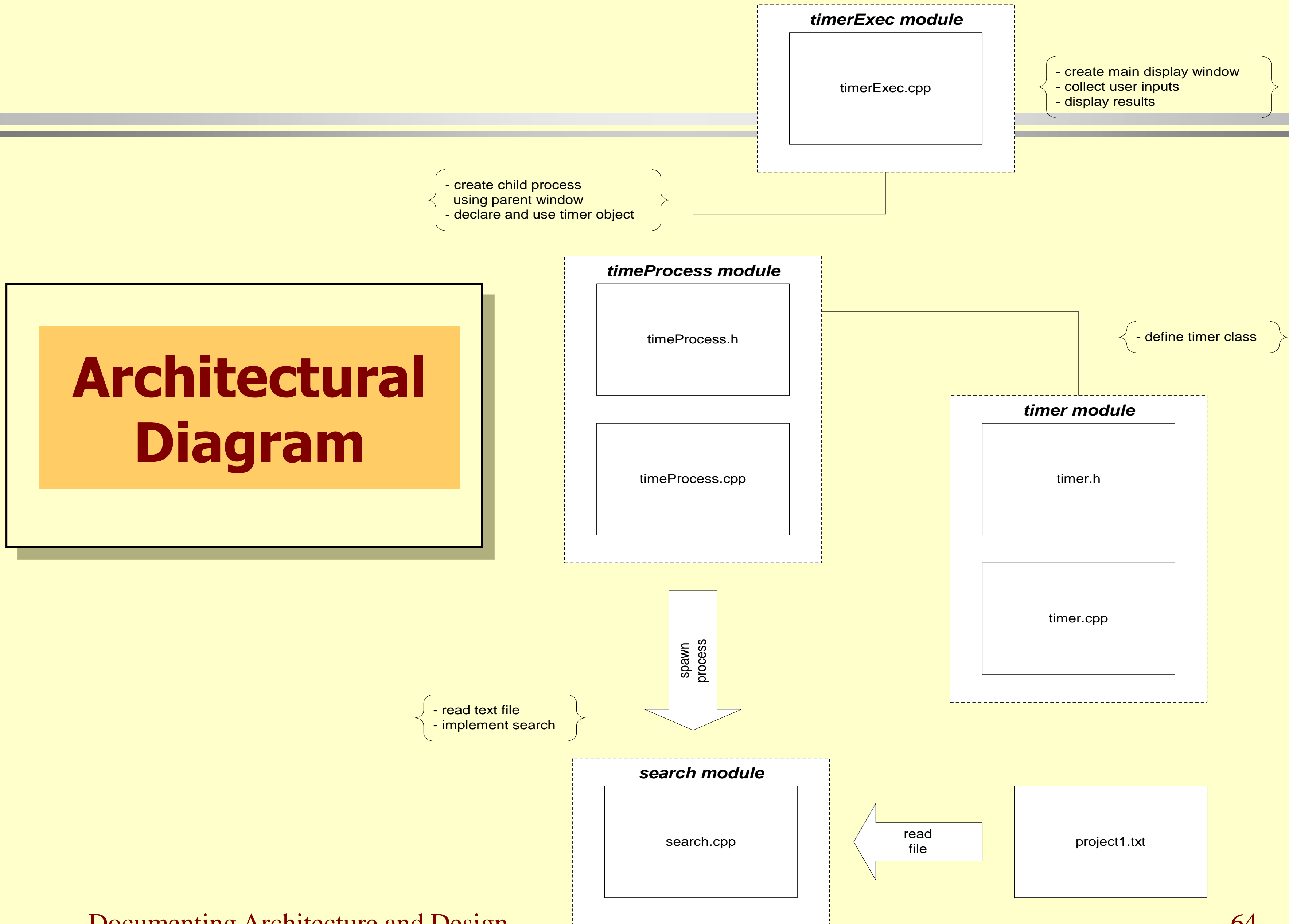
- Module diagrams show function calling dependencies between modules in a program.
  - Each module is represented by a labeled rectangle. Calling modules are shown above the modules they call.
  - A program should be decomposed into a single executive module which directs the activities of the program 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

Duplicates Program Module Diagram [Contents]



Architectural Diagram Contents

- 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.

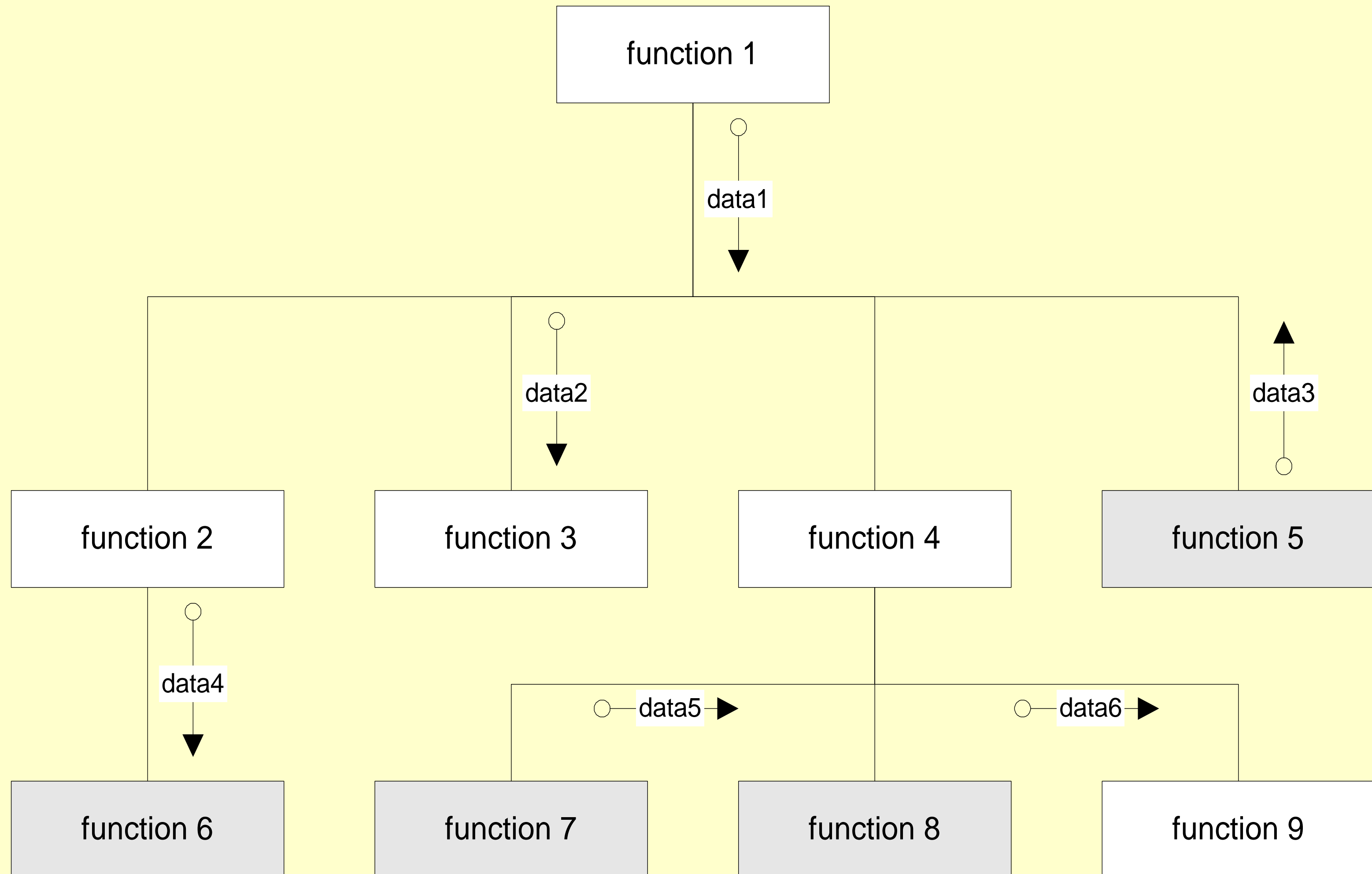

timerExec module
timerExec.cpp
- create main display window
- collect user inputs
- display results
- create child process
using parent window
- declare and use timer object
timeProcess module
timeProcess.h
timeProcess.cpp
- define timer class
timer module
timer.h
timer.cpp
Architectural Diagram
spawn process
- read text file
- implement search
search module
search.cpp
read file
project1.txt

Structure Chart Contents

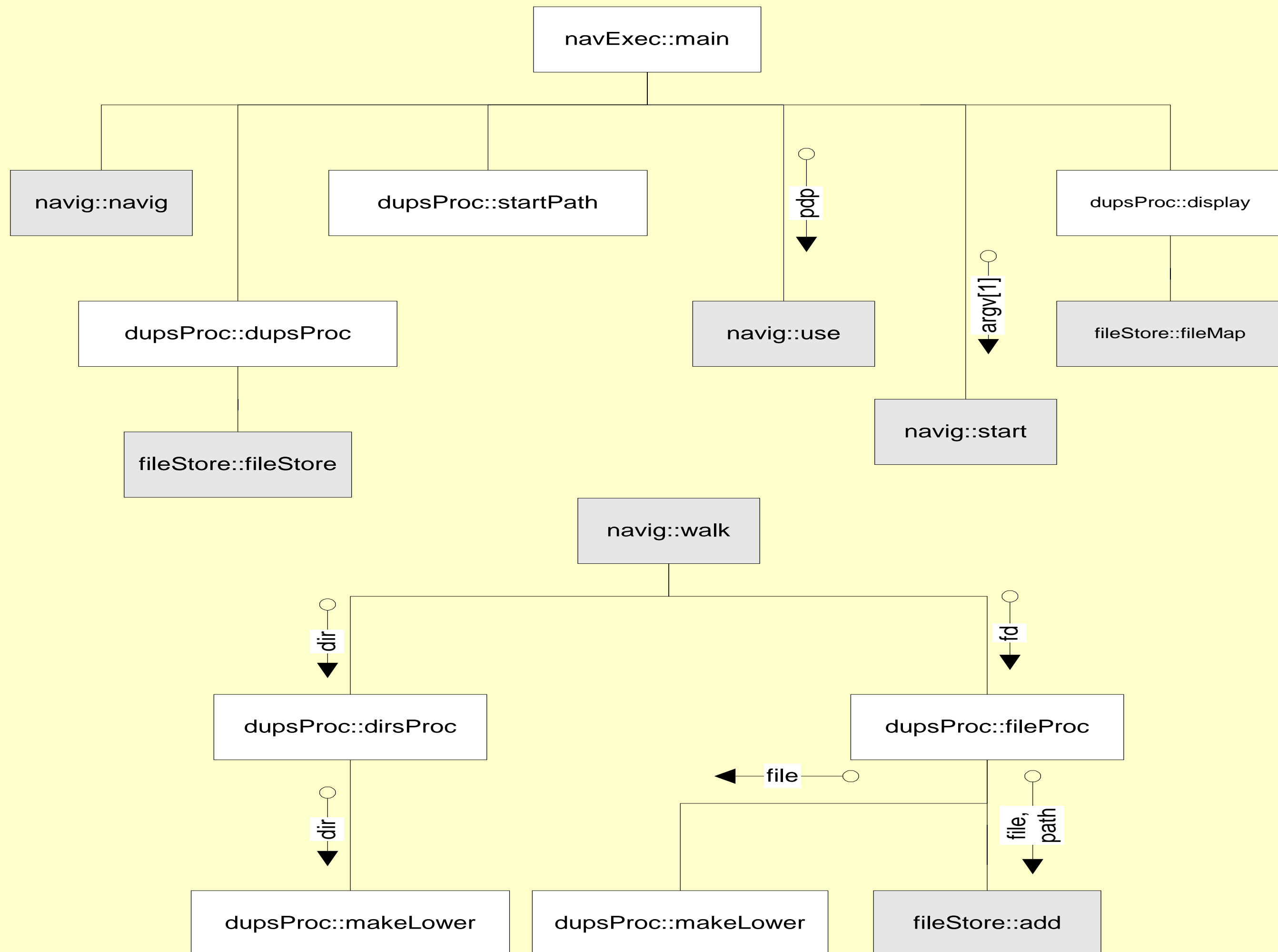
- The structure chart shows calling relationships between every function in a module and calls made into and out of the module.
  - Callers are always shown above callees.
  - Lines without arrow heads are drawn from the caller to the callee.
  - All data flowing between the invoking and invoked function 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



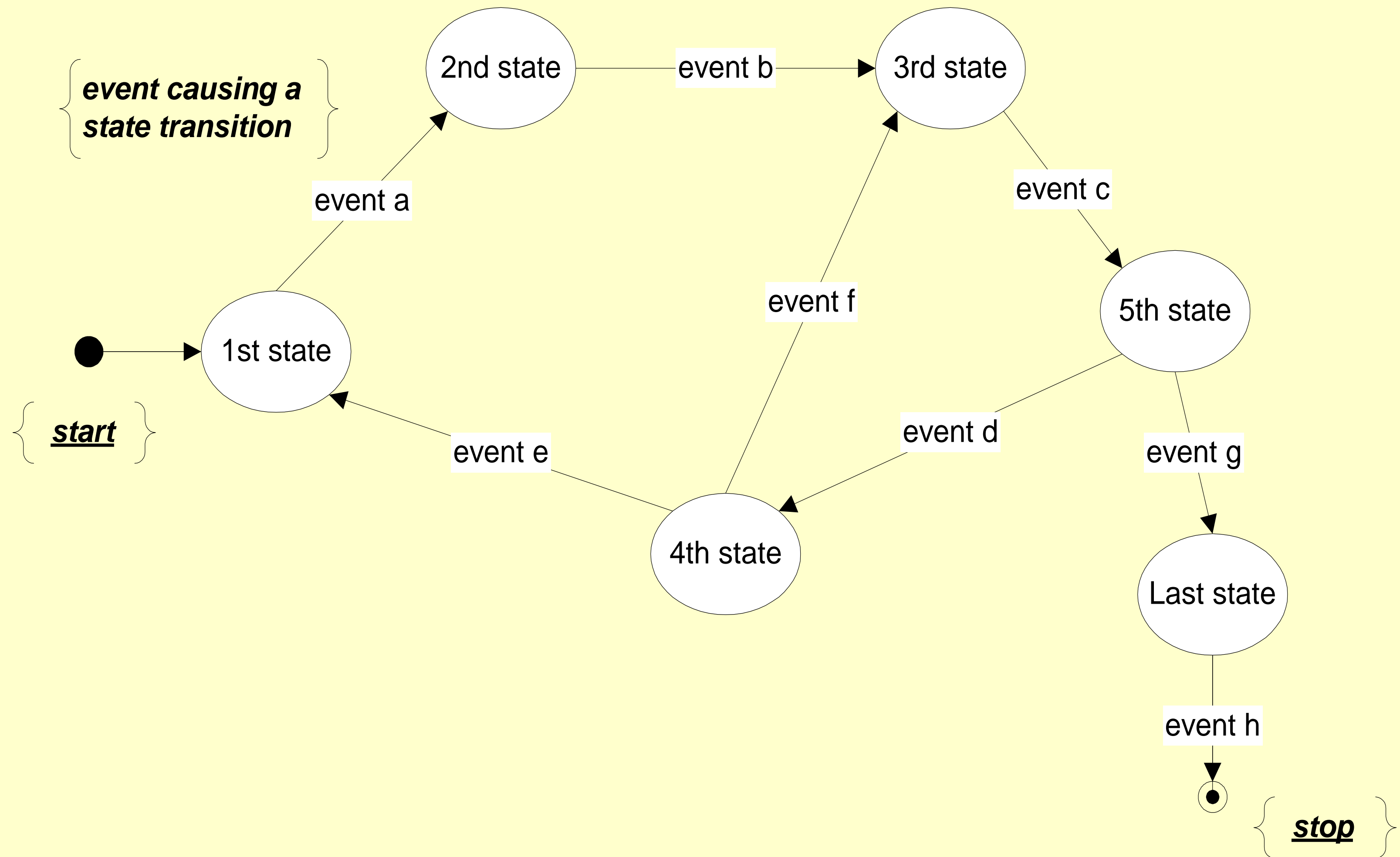
An Example Structure Chart [Contents]



State Diagram Contents

- 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 [Contents]



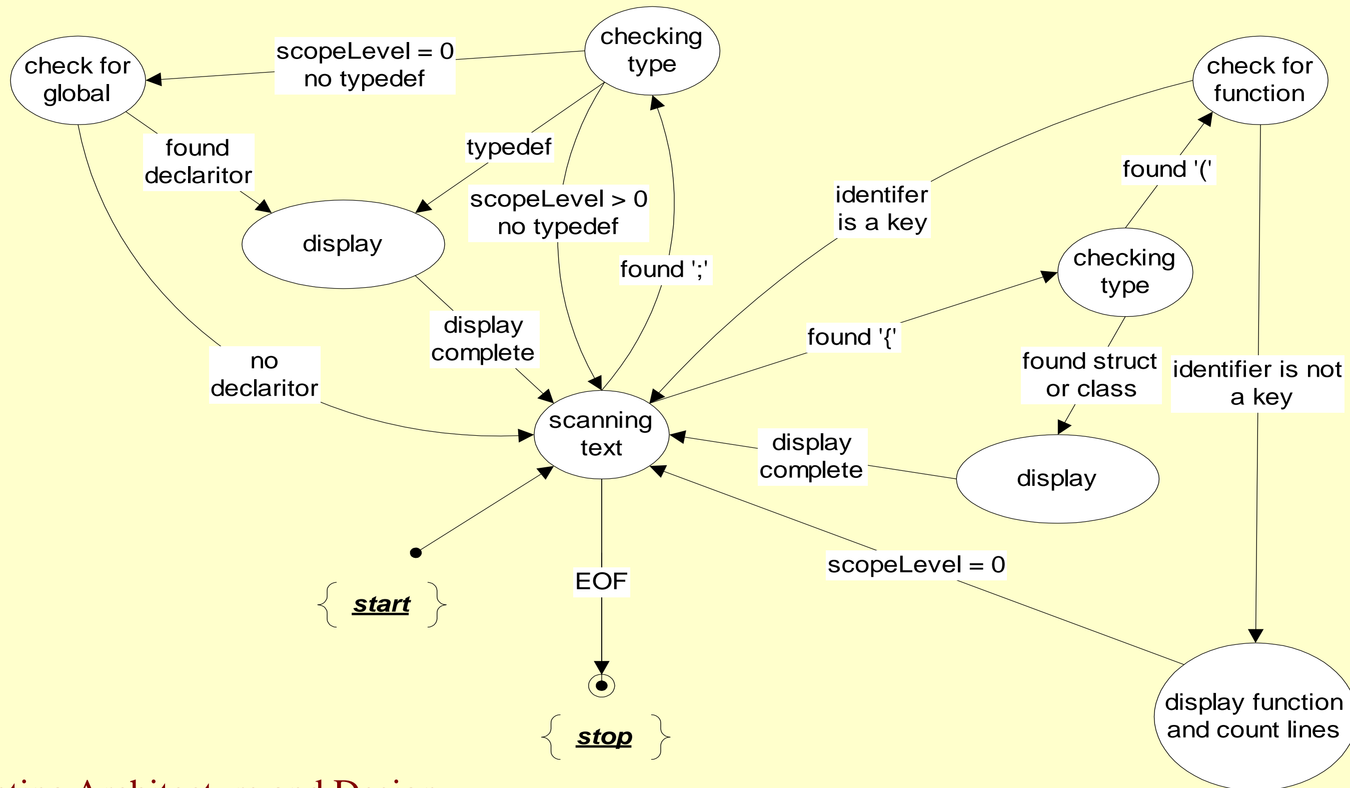
State Transition Diagrams [Contents]

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 [Contents]

- 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.



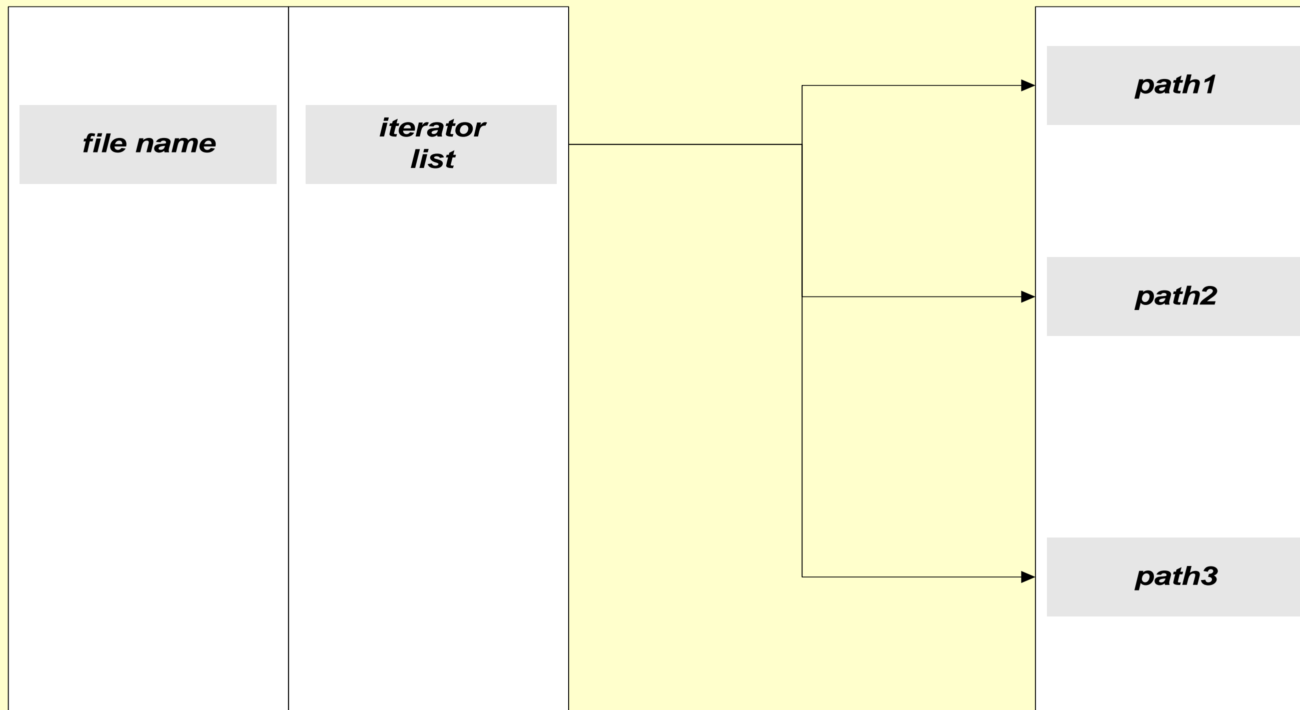
Data Structure Diagrams Contents

- 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 [Contents](#)

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

typedef set< string> pathSet



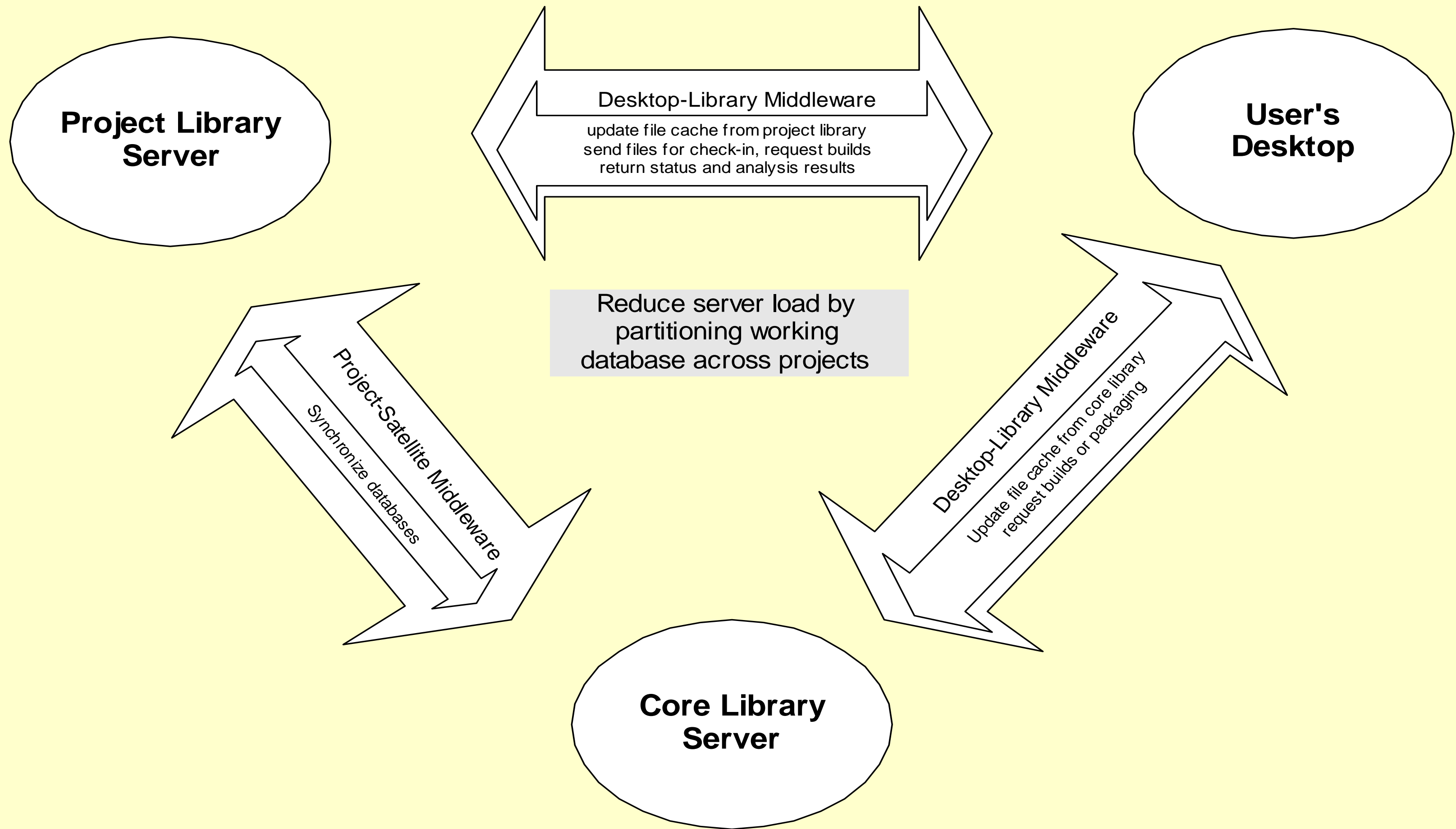
Note: Only one copy of each file name is stored in the fileMap and only one copy of each path is stored in the pathSet.

Software Repository Example

- Background:
  - It is common for software systems to require millions of lines of code for their implementation.
  - The implications of that size and complexity are:
    - Large teams are required in order to complete development in reasonable time.
    - Maintaining conceptual integrity and managing development are extraordinarily difficult.

- Goals:
  - Support massive reuse of existing software so that a large fraction of the total is reused without modification. Issues are:
    - ***Efficient search*** – find a few components out of a collection of many thousands.
    - ***Quality assurance and maintenance*** – support process of maintaining a massive collection of certified components.
    - ***Distribution*** – make components available to all developers anywhere in an organization.

Software Repository Structure - Project and Core Databases



Analysis of Goals

- Assumptions:
  - Quality assurance is primary function of repository server.
    - Reuse implies no change in documentation, code, and test apparatus for the reused components.
    - Very tight quality control is needed for this to be practical.
- Implications:
  - Usability implies that it should be easy to read and extract components.
  - Need for quality guarantees implies that it should be difficult to enter new components and to change existing components.
  - Thus support of analysis tools and role based administration is needed.
  - Support for transition from newly developed component to certified component is needed. A "holding tank" metaphor is appropriate here.

Analysis of Scale

- Assumptions:
  - organization of 2000 developers
  - working on five concurrent projects
  - average software size of 5 million lines of code.
  - ten existing systems fielded and currently operational
  - Each system has a common core of half its total software
  - Average file size is 400 lines of code.
  - Half of the developers are working directly on code at any time.
  - Each developer uses average of five files concurrently.

- Implications:
  - 15 total systems * 2.5 million unique lines of code + 2.5 million lines of common code ⇒ 40,000,000 lines of code
  - 100,000 files in repository.
  - 5000 files being used throughout the work day.

Analysis of Load

- Assumptions:
  - One server holding repository software resource.
  - 1000 user's login between 9:30 AM and 11:00 AM, browsing for average of ½ hour each.
- Implications:
  - 500 user hours of service in 1½ hours $\Rightarrow$ 333 simultaneous users
  - This is an untenable load
- Consequences:
  - Partition into system distributed between server and client desktops.
  - Clients browse on their own desktops, make file requests only when needed.
  - Cache files on desktop to minimize server traffic.
  - Use message based communication to minimize length of connection to server.
- Design strategy:
  - Make distributed file management transparent to user.

Analysis of Function

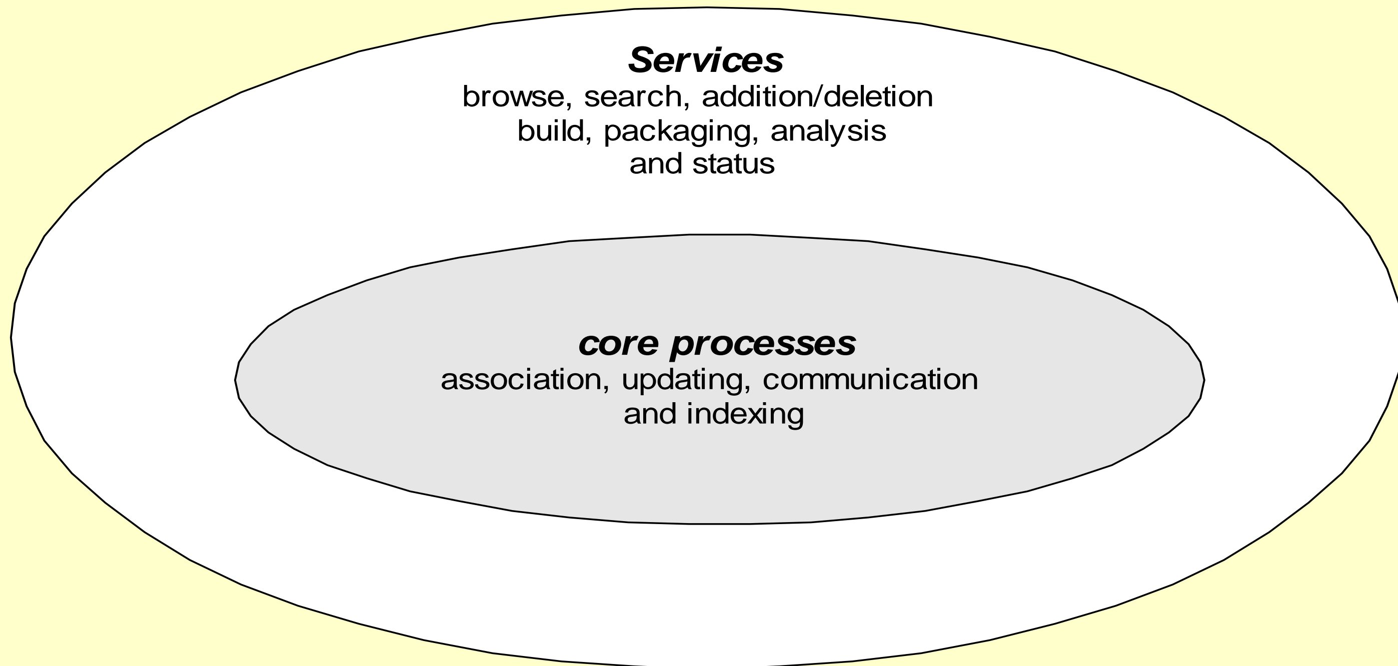
- Need to find a small number of files out of many thousands implies need for:
  - Hierarchical database structure
  - Associations maintained between files, modules, programs, systems
  - Powerful search techniques
  - Support for traversing entire hierarchy on desktop, even though only a small part of the repository's files are cached locally
- Desire to use repository structure for projects, as well as storage for corporate resources implies need for:
  - Status reporting at every level
  - Simple, but effective configuration management

Summary of Conclusions

Need:

- Hold at least 100,000 components in repository
- Distributed file management system with caching on desktop
- Message passing communication
- Hierarchical associations between components
- Powerful search techniques
- Role-based administration
- Support for quality assurance tools
- Status reporting for all components
- Support configuration management

Software Repository - Services and Processes

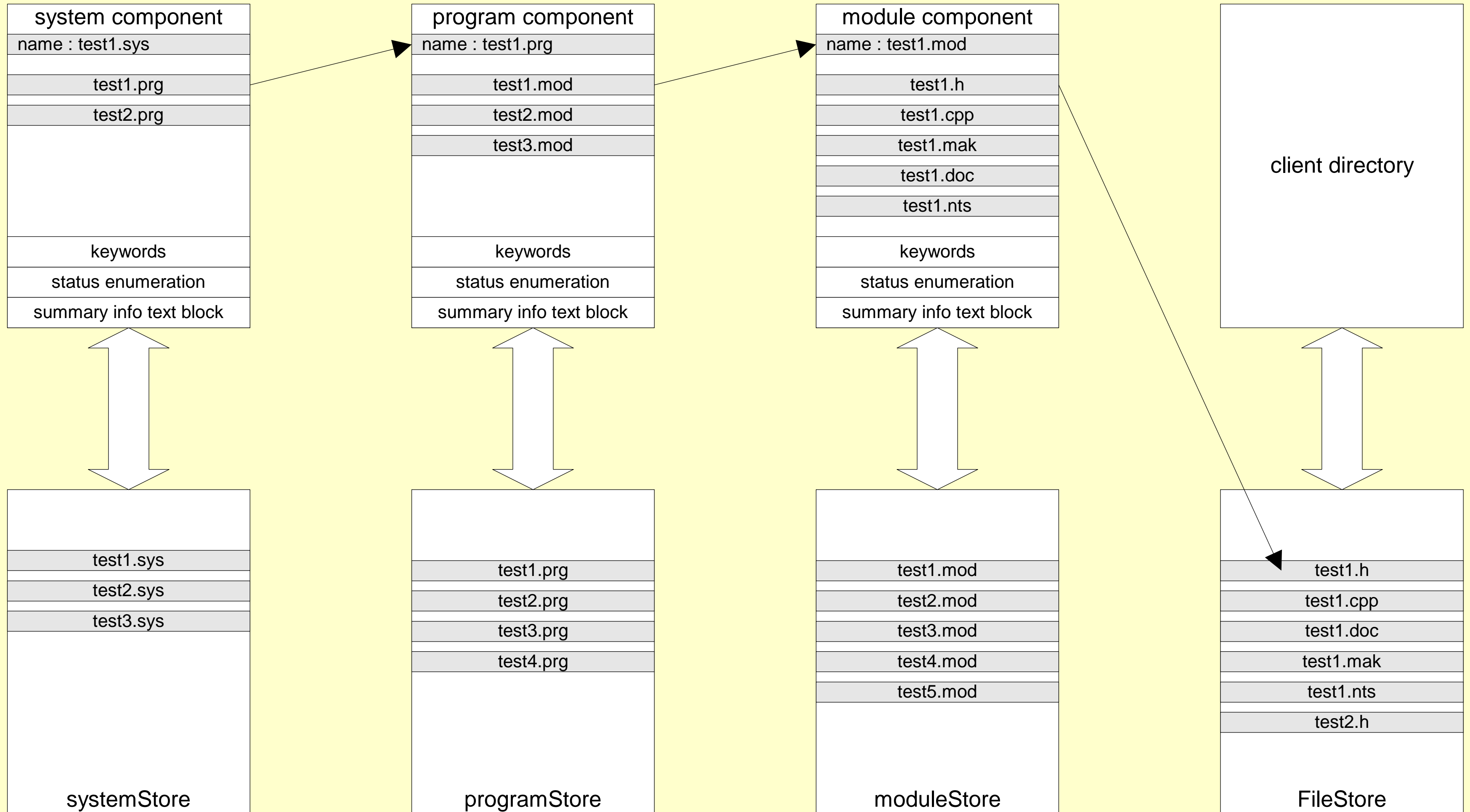


Services are repository processes that the user interacts with

Organizing Principle – Component Structure

- Organize Repository database into:
  - File store: holds exactly one copy of each file
    - Distinct versions are considered to be different files, e.g., token.cpp.3 and token.cpp.4 are both stored in file store.
  - Module Store, Program Store, System Store
    - Versioned in the same manner as files.
  - Units of reuse are components:
    - Module is a list of files
    - Program is a list of modules and (documentation and test) files
    - System is a list of programs and files
  - Components are represented in the Repository by indexes.
    - An index is a file containing:
      - Path of each lower level component and file
      - Set of keywords
      - Status information
      - Brief statement of function

Components - Defined by Presistent Repository Association Index Structure



Organizing Principle – Search Indexes

- Create module, program, and system indexes to capture the results of searches.
  - Example: Search for thread-safe queue
    - Returns program index pointing to modules that have queue and threading attributes
    - Returns system index pointing to all programs that use these thread-safe queues
  - Example: Program up-date
    - Given some program, search for latest versions of all its lower level components, e.g., modules and files.
    - Return as next version of the program.
    - Obviously, this type of up-date applies to components at any level.

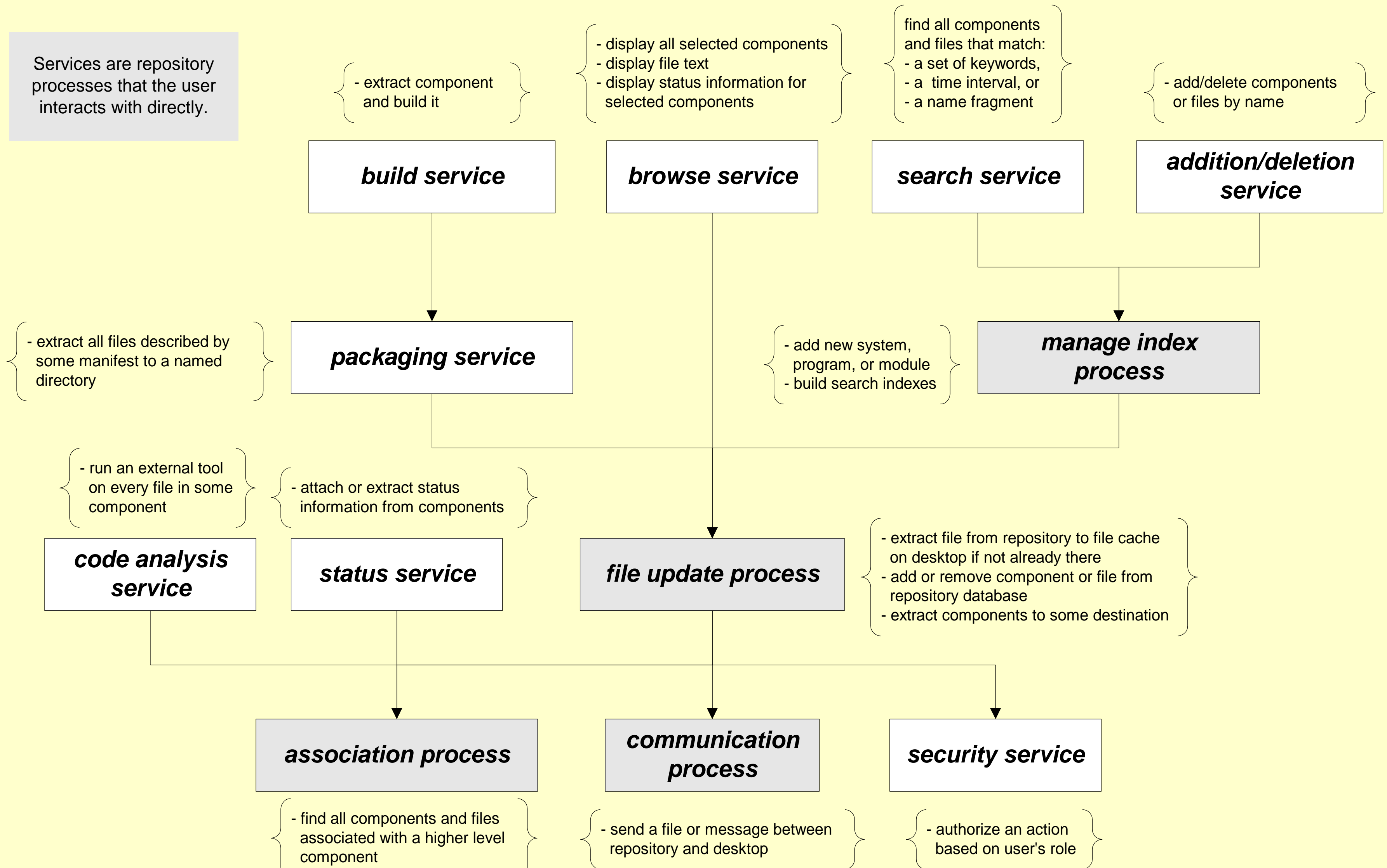
Organizing Principle – File Management

- Operation of Distributed File Management System
  - Every night server sends all its indexes to each desktop.
  - Each index contains keywords, association paths, status information, and a brief text summary of component.
  - Thus client can browse through entire repository structure without making frequent requests of server.
  - Only when client clicks on file link and file is not in local cache will server receive a file request.
  - Client cache is purged only when running low on disk space. Uses least recently used purge algorithm.

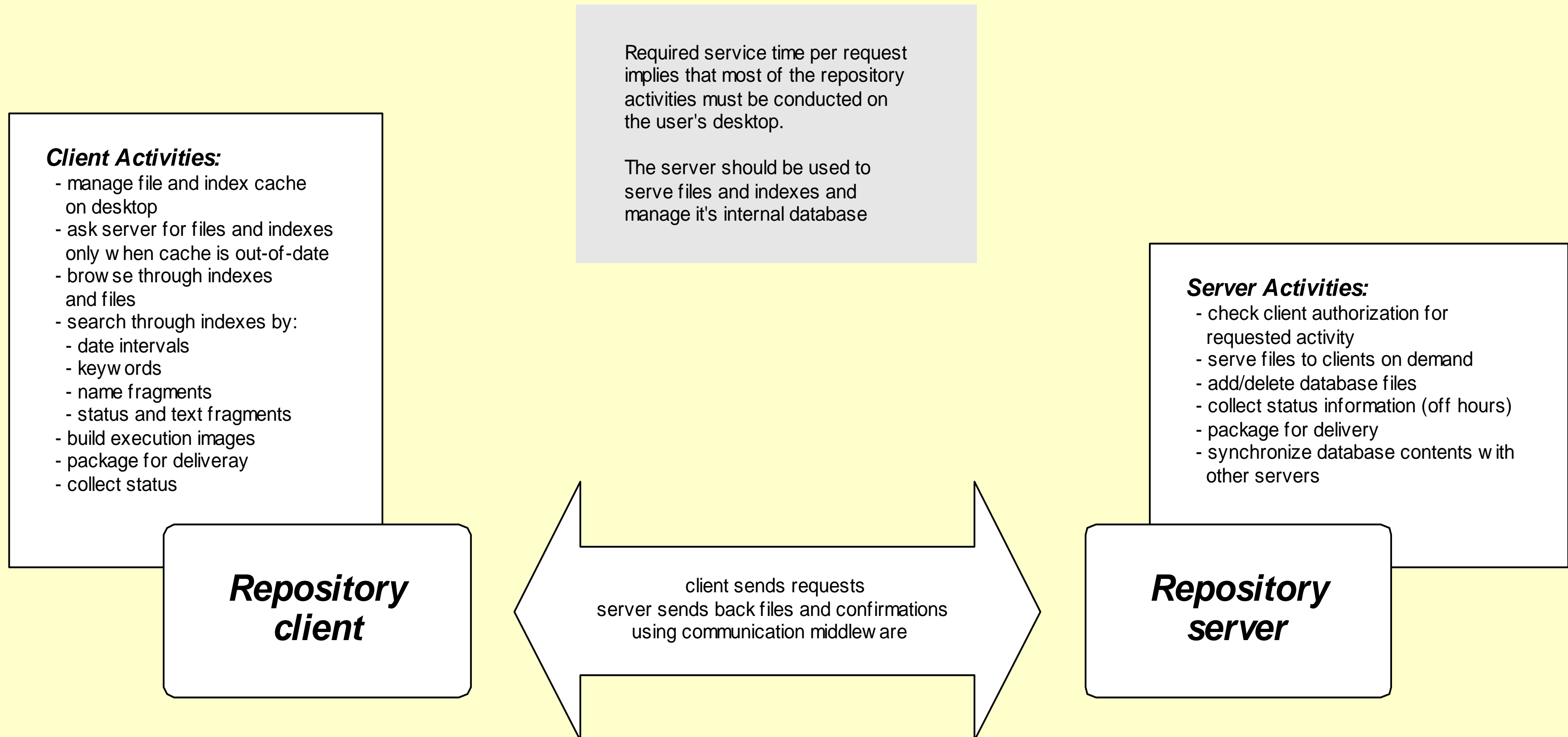
Working Set Size

- Estimate of working set:
  - Number of modules = number of files / 4 = 25,000
    - 2 source code files
    - One test driver
    - One documentation file
  - Number of programs = number of modules / 5 = 5000
  - Number of systems
    = 5 current projects + 10 legacy projects + 4 experimental prototypes
    = 19 systems
  - Number of indexes = 25,000 + 5000 + 19 = 30,019 indexes
  - Size of index set = 30,000 * 1 KByte each = 30 MB of index data
- Could greatly reduce this traffic by sending only new indexes each night
  - Occasionally synchronize by sending complete set.

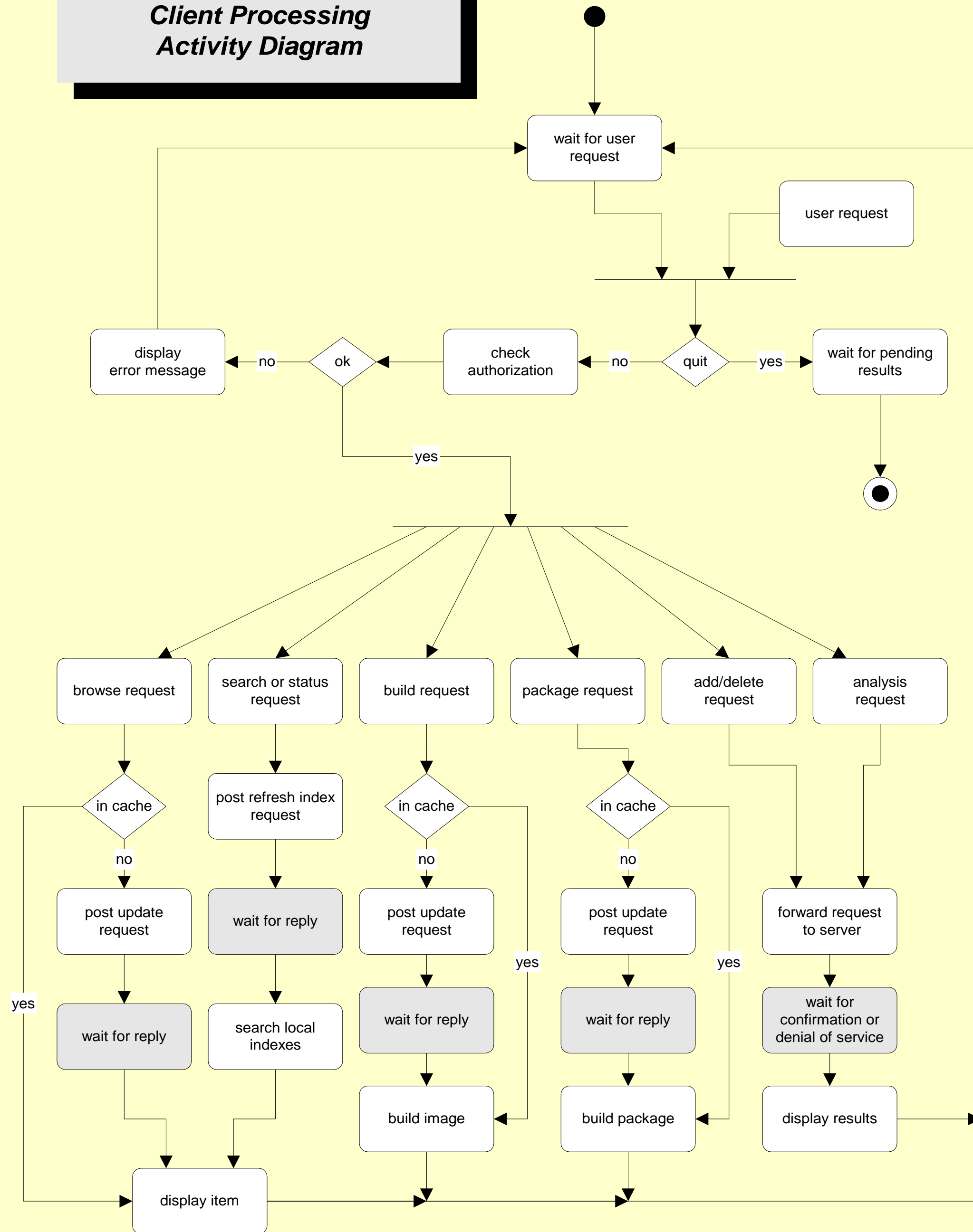
Software Repository - Functional Partitions and Dependencies



Software Repository - Client/Server Structure



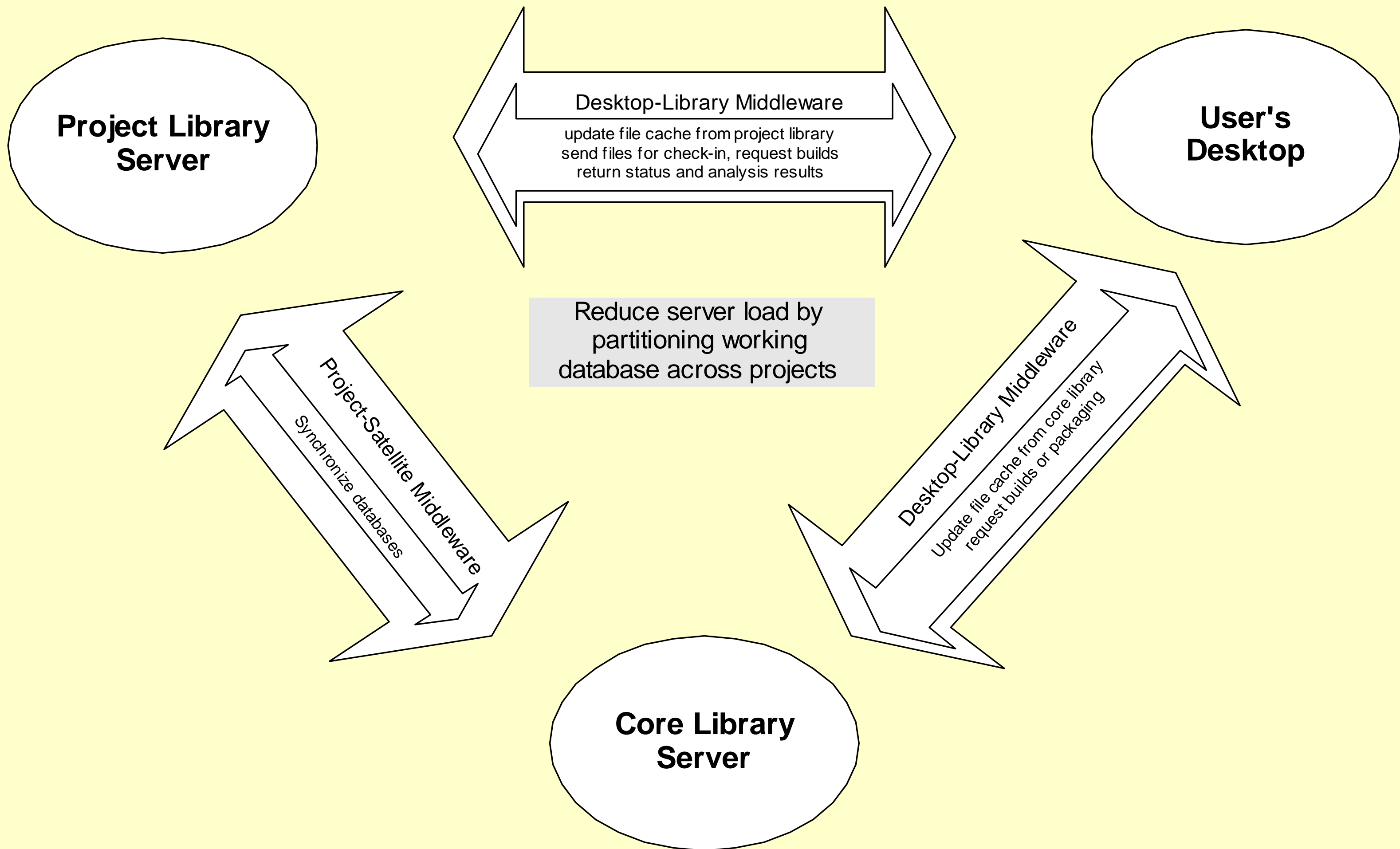
Software Repository - Client Processing Activity Diagram



Organizing Principle – Server Structure

- Organize Repository into a hierarchy of servers and desktops.
  - Primary server holds core repository of reusable components.
  - Project servers are cloned from main repository.
    - Use different rules to manage contents.
      - Easier to add new components and modify existing components.
  - Client desktops are created as subsets of main repository, with contents determined by owner's job functions.
  - Only local browsing is allowed on any server.
    - Only administrators can browse the primary and project servers.
    - Clients only browser the desktop cache.
  - Index distribution supports simulated browsing of the entire repository at client desktops.
    - Distributed file management satisfies file opening requests by first searching the local cache, then downloading from a server if needed.

Software Repository Structure - Project and Core Databases



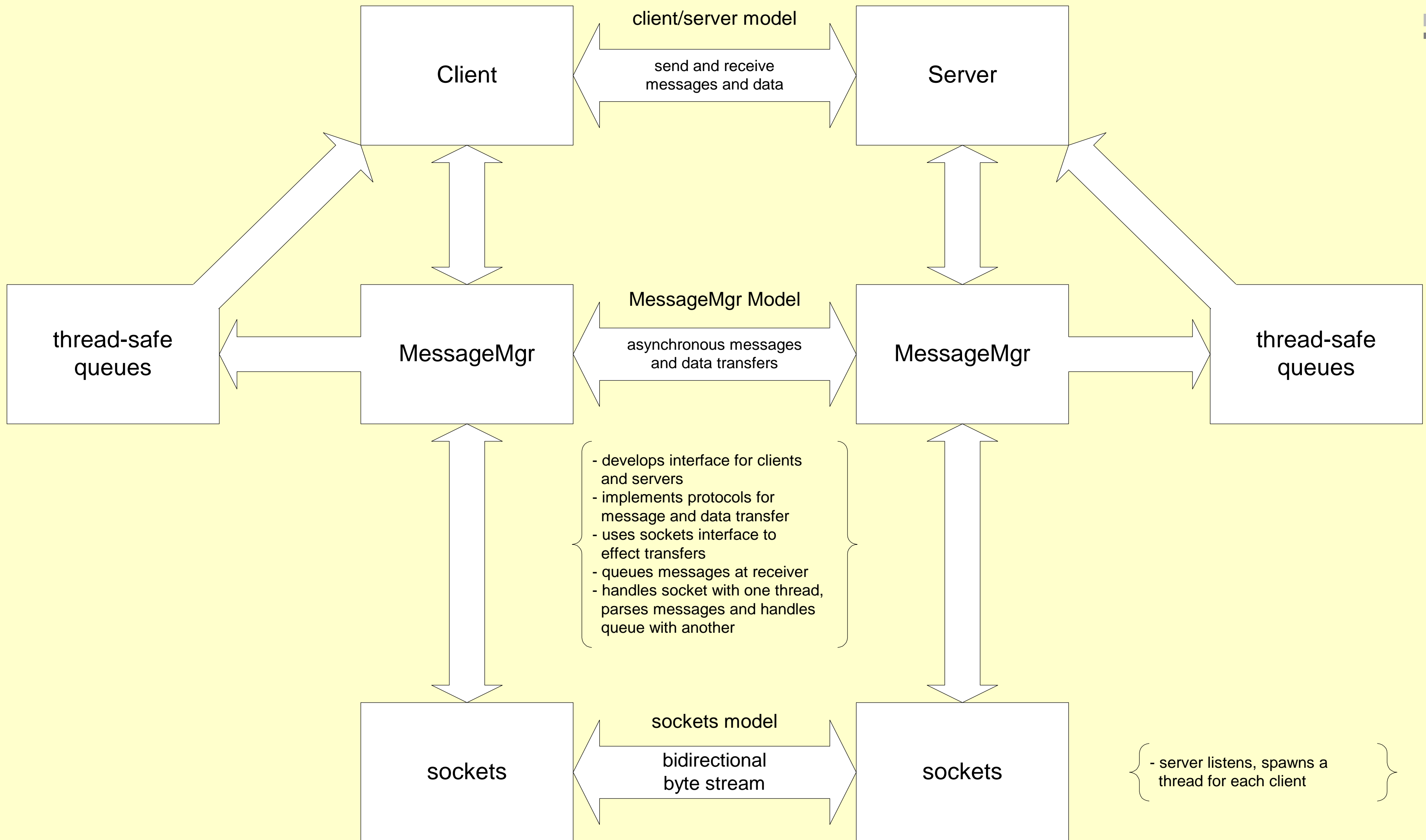
Organizing Principle – Configuration Management

- The proposed repository association index structure can support a very flexible configuration management policy:
  - Every file (source code, document, index) is given a version number, mydoc.doc.3.
  - Indexes refer to a file by name, extension, and version.
  - No file appears more than once in the repository, but all versions are stored.
    - Ancient versions may be retired to an offline archive once no supported system refers to them.
  - Components, e.g., all those objects represented by an index file, are versioned by versioning their indexes.
  - A component can be updated to a new version by updating any one or all of its links to the latest versions of the named files. This automatically increments its version.
  - Since indexes are small files, we can afford to keep all versions under the repository control.
  - Special indexes can be used to point to all the latest changes in the repository, allowing a very flexible updating policy. A team will update to new software, developed by another team, only when they are ready.

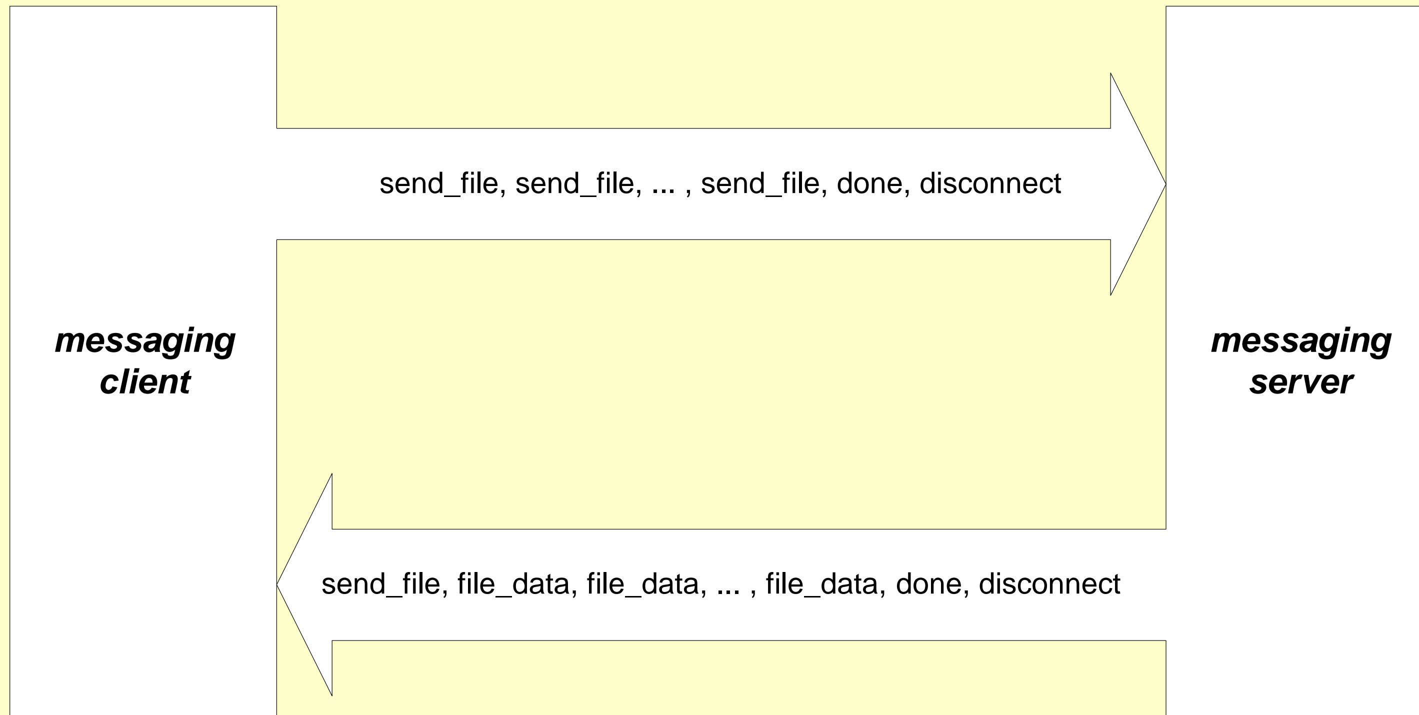
Communication System

- Communications between desktop and servers are based on message-passing.
  - More flexible than Remote Procedure Calls.
  - Client makes request for service:
    - File request
    - Status request
    - Update index request
  - Message is queued on server, serviced based on availability of server CPU cycles and priority – file requests have top priority.
  - Server pushes results to client.
  - Desktop enqueues server results. Queue handler at desktop gets high priority so server does not block.

Project #3 - Messaging System Architecture

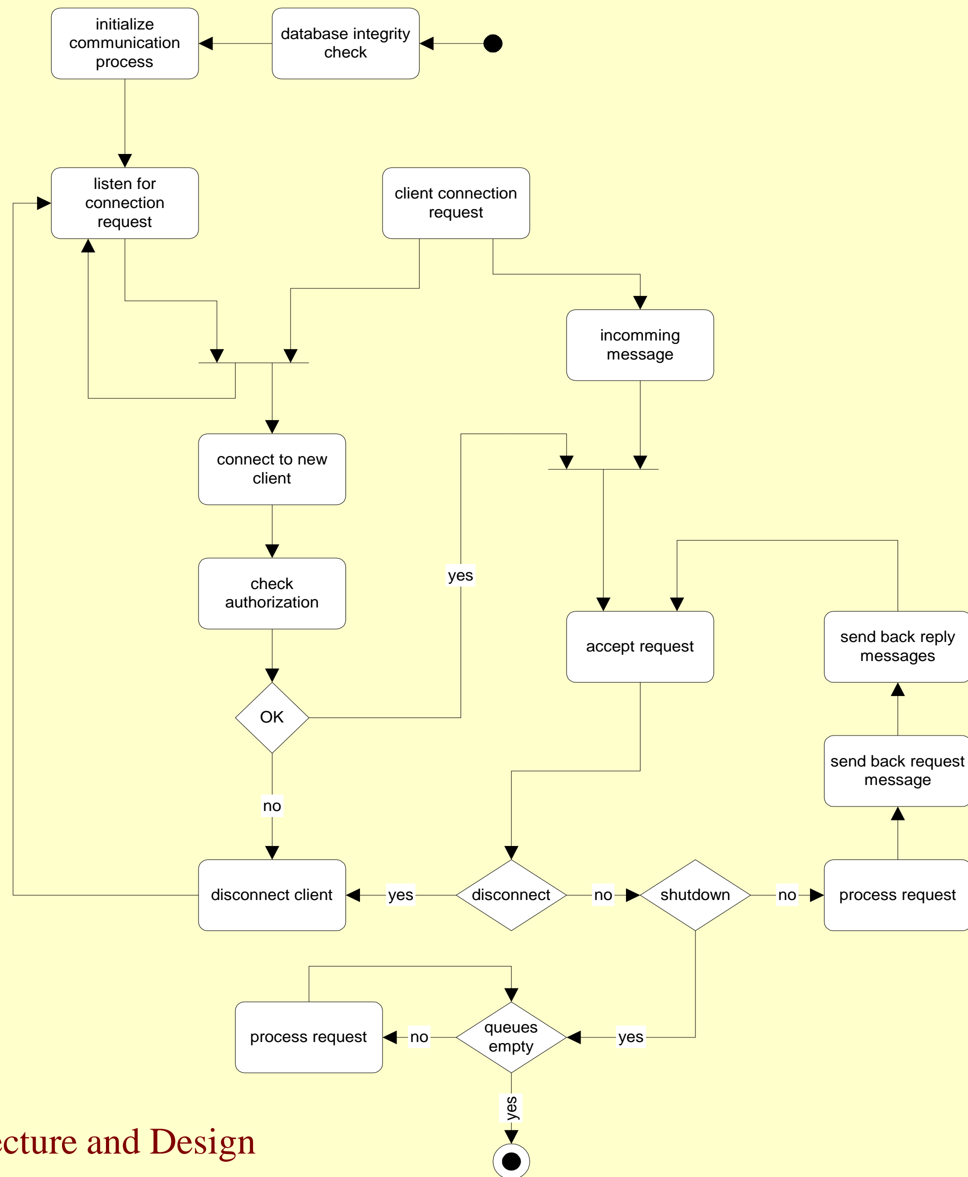


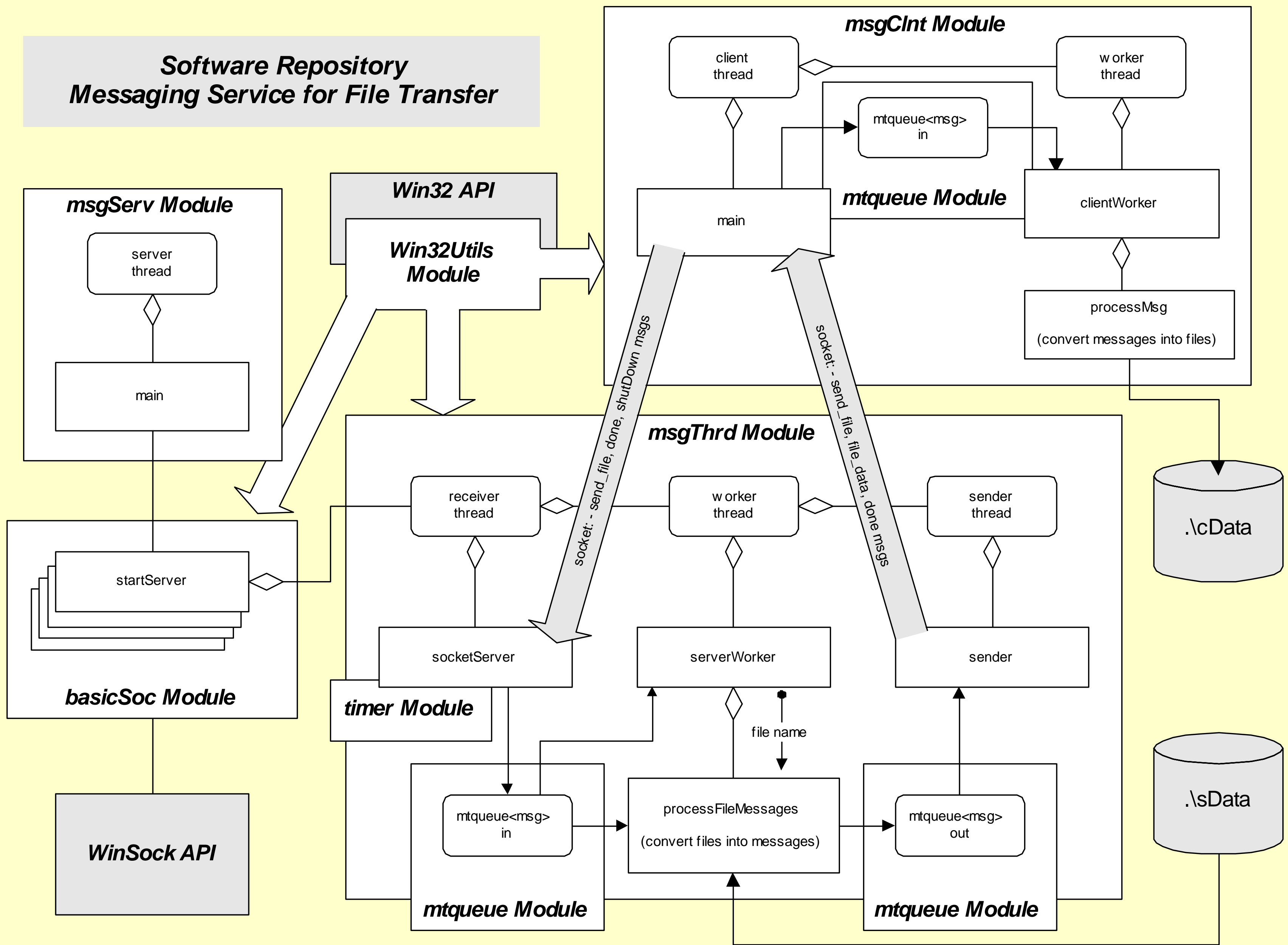
Software Repository - Message Protocol



Organizing Principle:
Server reflects back all of the client's request messages and done message

Software Repository - File Serving Activity Diagram



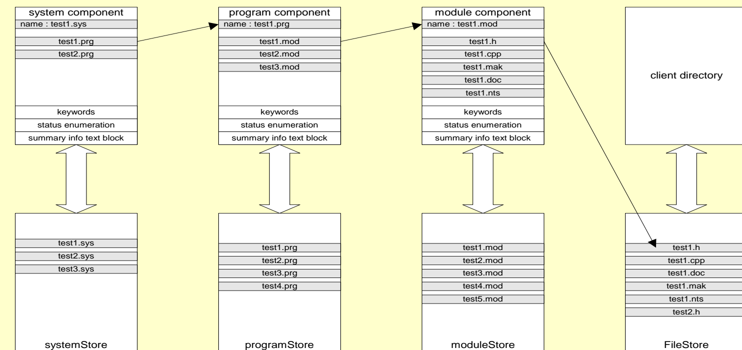

Software Repository
Messaging Service for File Transfer
msgClnt Module
client thread
w orker thread
mtqueue<msg> in
main
mtqueue Module
clientWorker
processMsg
(convert messages into files)
Win32 API
Win32Utils Module
msgServ Module
server thread
main
socket: - send_file, done, shutDown msgs
socket: - send_file, file_data, done msgs
msgThrd Module
.\cData
receiver thread
w orker thread
sender thread
startServer
basicSoc Module
socketServer
serverWorker
sender
timer Module
file name
.\sData
WinSock API
mtqueue<msg> in
processFileMessages
(convert files into messages)
mtqueue<msg> out
mtqueue Module
mtqueue Module

Assessment of Risk

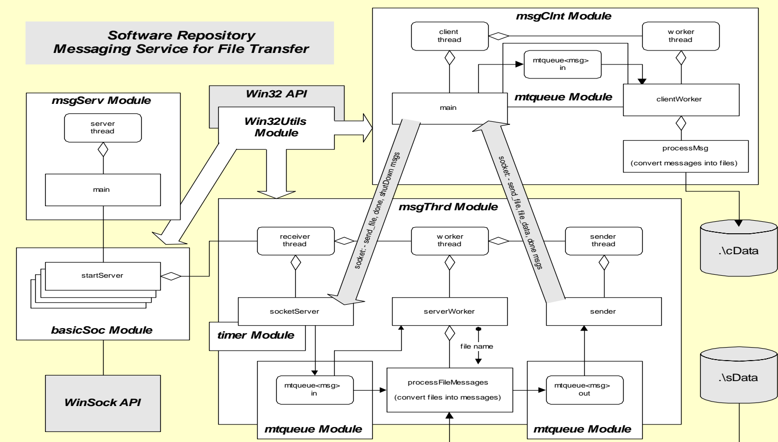
- Risk Areas:
  - Message-passing communication: issues are complexity and performance.
  - Association process, capturing relationships between files, modules, programs, and systems: issues are performance and robustness.
  - Load handling capacity of core repository server: can a single server support demands of a large community of developers?
  - Security: how do we make the repository easily accessible to all our developers, including those at remote sites, while keeping our competitors and malicious hackers from compromising our proprietary software resource?

Risk Abatement

- Association prototype demonstrates that file-based association will support adequate performance. Should investigate other approaches.



- Message passing prototype demonstrates that multi-threaded socket servers support asynchronous file server with required performance.



Risk Abatement

- Repository server performance:
  - Distributed file cache management and index updating service will easily provide adequate performance within a connected network.
  - Should the repository service be provided through web servers, openly accessible world-wide, it will be necessary to provide mirror web hosts. Since the repository contents do not change frequently, this should be easy to manage.
- Repository security:
  - The management plan for the repository calls for anonymous reading, but only administrator write access, conventional security measures should be adequate, e.g.:
    - Firewall controlling the type of access
    - Mirroring contents in multiple sites, including backup sites with no public access.

Risk Abatement

- Protecting proprietary value:
  - Providing strong, conventional network security and allowing outside access only through virtual private networks provide protection against naïve attackers from the outside.
  - Providing read access only through proprietary reader software, different from browser-based access (http and ftp) will add an additional layer of security.
  - There is one gaping security hole in this architecture: the inside job. This system will do nothing to prevent an employee or contractor with repository access from walking out with a box of CDs burned with the repository contents.
  - Two approaches come to mind:
    - Encrypt all contents and decode only on proprietary software keyed to run only on one specific machine, like the Microsoft Activation scheme.
    - Don’t treat the repository contents as proprietary.