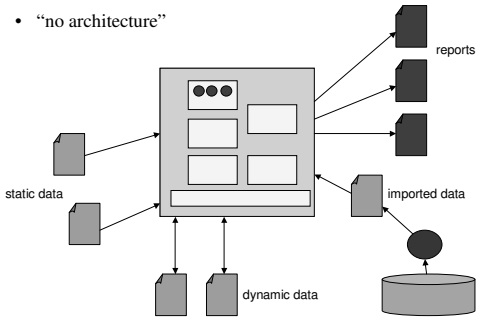
# Systems Architecture

## Monolithic Systems



## Monolithic Systems

- "no architecture"






## Examples

- Most programs you deal with day-to-day
  - word processing
  - spreadsheets
  - powerpoint
  - e-mail (?)
  - inexpensive accounting packages
  - development environments
  - compilers
  - most games
    - (not *Combat Flight Simulator*)
- Large, corporate batch systems
  - payroll
  - reports
  - Descartes route planning



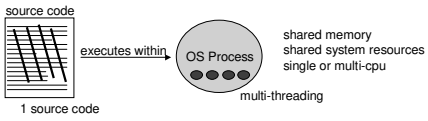
## Characteristics

- Usually written in a single programming language.
- Everything compiled and linked into a single (monolithic) application
  - as large as 1 MLOC C++
  - as large as 100M loaded executable
  - as large as 2G virtual memory
- May operate in both
  - batch mode
  - GUI mode
- Data
  - load into memory
  - write *all* back on explicit save
    - No simultaneous data sharing
- May have concurrency
  - multi-threading
  - multi-processing (but only one executable)



## Concurrency

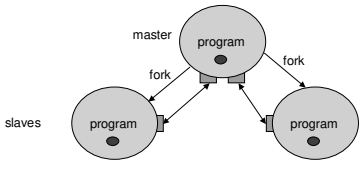
- multi-threading






## Concurrency

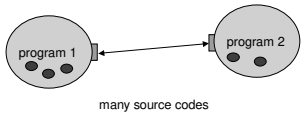
- symmetric multi-processing




1 source code

## Concurrency

- distributed processing




many source codes



## Concurrency

- Why multi-threading?
  - performance (when you have access to multiple CPUs)
  - A design philosophy for dealing with asynchronous events
    - interrupts
    - GUI events
    - communications events
  - Maintain liveness
    - can continue to interact with user despite time-consuming operations
      - e.g., SMIT "running man"
  - performance
    - pre-load, network initializations
  - multi-tasking (lets the user do many tasks at once)
    - e.g., downloads from the net
- You WILL have to multi-thread your program
  - start early in the design process



## Concurrency

- Why symmetric multi-processing?
  - you need parallelism
    - performance
    - liveness
    - …
  - a program is not written to be multi-threaded
    - temporarily modifying shared data
  - fork cost is inexpensive relative to amount of work to be done by slaves
    - fork typically implemented with COW
- Tricks:
  - special allocators to group modifiable shared data to contiguous memory
- Using memory management hardware to switch volatile data based on "thread"



## Monolith Architecture

- A monolithic system is therefore characterized by
  - 1 source code
  - 1 program generated
  - but… may contain concurrency



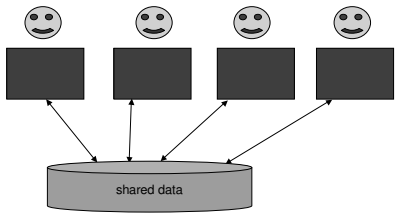
## Data

- In a monolithic architecture
  - data is read into application memory
  - data is manipulated
  - reports may be output
  - data may be saved back to the same source or different
- Multi-user access is not possible



## Multi-User Access

- Can changes by one user be seen by another user?
  - not if each copy of the application reads the data into memory
  - only sequential access is possible



## Multi-User Access

- Allowing multiple users to access and update volatile data simultaneously is difficult.
- Big extra cost
  - require relational database expertise
- More on this later.



## Advantages

- performance
  - accessing all data
    - disk is disk!
    - either
      - read data more directly from the disk via file system
        - » highly optimized
        - » caching and pre-fetching built-in
      - read data less directly from the disk via layers of intervening software (e.g., RDBMS, OODBMS, distributed data server).
  - modifying data
    - in-memory is massively quicker
    - caching is not an option for shared data systems
      - delays while committing changes to a record
  - No IPC overhead
- simplicity
  - less code to write
  - fewer issues to deal with
    - locking, transactions, integrity, performance, geographic distribution



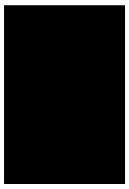
## Disadvantages

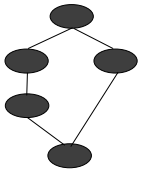
- Lack of support for shared access
  - forces one-at-a-time access
  - mitigate:
    - allowing datasets that merge multiple files
    - hybrid approach
      - complex monolithic analysis software
      - simple data client/server update software
- Quantity of data
  - when quantity of data is too large to load into memory
    - too much time to load
    - too much virtual memory used
  - Depending on which is possible
    - sequential access (lock db or shadow db)
    - selective access



## Red Herring

- Monolithic systems are “less modular”

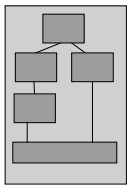


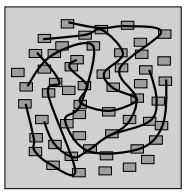




## Red Herring

- The code for distributed systems will need to share common objects.
  - This “module” organization could be terrible.







## Red Herring (sort of)

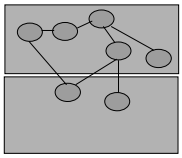
- Distributed systems *require* architects to define and maintain interfaces between components
  - cannot do anything without this
  - even for RDBMS systems
    - relational schema + stored procedures define an important interface
  - by default: nothing is visible
    - must work to expose interface
- For monolithic systems, this is “optional”
  - because there are no process boundaries, any tiny component can depend on (use, invoke, instantiate) any other in the entire monolithic system. *e.g.,*

```
extern void a_routine_I_should_not_call(int a, int b);
```

  - default: everything is visible
    - must work to hide non-interface

## Module Structure

- To preserve the architectural integrity of a monolithic system, we must work to define and maintain (typically) extra-linguistic sub-system boundaries.
  - recall façade pattern





## Module Structure in the C Language

- `.h` & `.c` together define a "translation unit"
  - `.h` file is the interface
  - `.c` file is the implementation

edit.h

```
#ifndef _EDIT_
#define _EDIT_
extern void edit();       /* opens a new edit session */
extern void panic();      /* backup-save any unsaved files */
…
#endif
```

edit.c

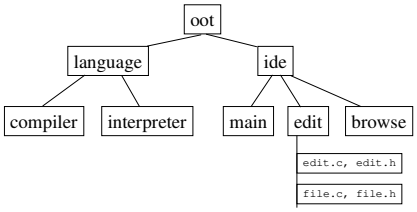
```
#include edit.h
void edit() {
   …
   editHelper();
   …
}

static void editHelper() {
   …
}
```



## Module Structure in the C Language

- Group a set of translation units into a subsystem
  - directory = subsystem
- Group sets of subsystems into higher-level subsystems
  - use the directory structure






## Module Structure in the C Language

- Rule:
  - all routines not declared in the .h file should be defined static
    - "AntiPattern" is a violation of this
      - a routine not declared in the .h file that is not defined static
  - to use an exported routine from outside the translation unit:
    - must #include the .h file
    - can only call routines and use data structures declared in the .h file



## Module Structure in the C Language

- Rule:
  - each directory (=subsystem) must contain a README file
  - the README file
    - explains the concept of the subsystem
    - indicates which .h files can be called from outside the subsystem
      - translation unit export
    - indicates which subsystems can be called from this subsystem
      - subsystem import
  - Higher-level directories
    - explain the concept of the subsystem
    - indicates which subsystems can be called from outside
      - subsystem export
    - indicates which (higher-level) subsystems can be used

(no C language concept of subsystems)



## Issues

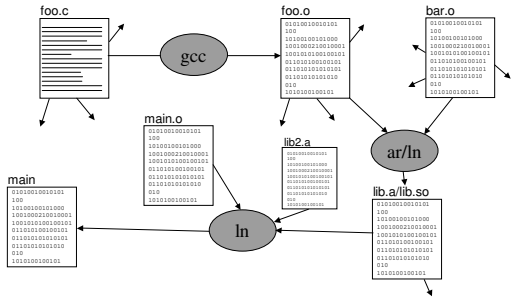
- Communications
  - conventions must be written down as "coding standards"
  - must educate new employees on local coding standards
- Staleness
  - README files get out-of-date w.r.t. the code
- Enforcement
  - typically no automated checking
  - must enforce via code review
- Change Management
  - assignment of groups / programmers to subsystems
  - process for requesting an interface change must be documented
    - is the change architecturally sound?
    - can the owner group handle the workload?
    - by when is the change required?
    - for what purpose (feature, bug fix)
- How to change the subsystem organization itself?

## Module Structure in C++

- Additional considerations:
  - Can a .H file declare more than one class?
    - Naming convention for .H files
  - Use of friends
  - Access via interface only?
    - hides private parts
    - reduces compilation dependencies
    - less efficient (all calls are virtual, less chance to inline)
    - Foo.H
      - public only interface
    - FooImpl.H
      - class declaration
      - inherits from Foo interface
      - adds private data/members
    - FooImpl.C
      - class member definitions



## Library Structure






## Library Structure in C/C++

- Decide
  - how many libraries to have
  - their names
  - which subsystems go into which libraries
    - wise to align library structure with a subsystem
    - not necessary to do so
      - e.g., could be a base level of utilities that rarely change whose TU's belong to unrelated subsystems (stretching it).
    - rationale
- Why?
  - reduce compilation dependencies
    - can be changing a bunch of .c's and .h's and others can keep using the library
    - but… don't change any.h's exported beyond the library
    - "poor man's" configuration management system
      - often most practical
  - Reduces link time (libraries often pre-linked)
  - Shipping libraries
    - Common library supports many apps



## Module Structure in Java

- class
  - nested classes, inner classes
- file
  - 1 public class
  - may have several package access classes
  - no such thing as file-only visibility
  - import one-at-a-time/all public classes from another package
- package
  - multiple files containing public and package access
  - package = directory (enforced)
- hierarchical packages
  - naming convention only
  - carries no access control meaning
  - necessary directory structure required
  - (mapping of linguistically named hierarchical subsystems to directory hierarchy)



## Module Structure in Java

- higher-level-subsystem import and export not supported
  - need to specify in
    - architecture document
    - and/or directory README

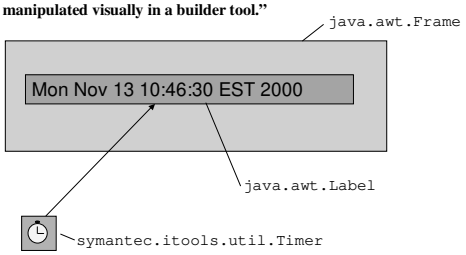


## Library Structure in Java

- jar file
  - (somewhat) analogous to a library
    - a collection of classes of various packages (can be a subset of classes)
    - bundle other resources (images, sounds, properties, …)
    - downloaded all at once when over a network
    - cached all at once

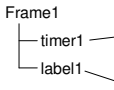
## Java Beans

- Java defines a standard for packaging a component in such a way that it can support visual programming.
  - **"A Java Bean is a reusable software component that can be manipulated visually in a builder tool."**






## Java Beans




| symantec.itools.util.Timer properties | |
|---|---|
| Name | timer1 |
| Repeat Automatically | true |
| Time (ms) | 1000 |

| java.awt.Label properties | |
|---|---|
| Alignment | LEFT |
| Background | orange |
| Text | text |
| Bounds | [12,60,355,74] |
| Enabled | true |
| Font | [Dialog, 12, Plain] |
| Foreground | black |



## Java Beans

- Java Beans have
  - features:
    - properties
      - simple
      - indexed
      - bound
      - constrained
    - events (that can fire interactions)
    - methods
      - by default all public methods
      - can limit
  - design time v. run-time



## Java Bean Simple Properties

- Property 'foreground'
  - accessors:
    - `public void setForegrouond(java.awt.Color);`
    - `public java.awt.Color getForeground();`
- Property 'text'
  - accessors
    - `public void setText(java.lang.String);`
    - `public java.lang.String getText();`
- Property 'URL'
  - accessors
    - `public void setURL(java.lang.String);`
    - `public java.lang.String getURL();`



## Java Beans Indexed Properties

- Indexed property 'items'
  - accessors:
    - `public String[] getItems()`
    - `public String getItems(int index);`
    - `public void setItems(String[] items);`
    - `public void setItems(int index, String item);`



## Java Beans Bound Properties

- Listen to changes in property 'text'
  - accessors (to listen to all)
    - `public void addPropertyChangeListener(PropertyChangeListener pcl);`
    - `public void removePropertyChangeListener(PropertyChangeListener pcl);`
  - accessors (to listen to just 'Text')
    - `public void addTextListener(PropertyChangeListener pcl);`
    - `public void removeTextListener(PropertyChangeListener pcl);`

## Listening for Property Changes

```
package java.beans;
public interface PropertyChangeListener
   extends java.util.EventListener {
    void propertyChange(PropertyChangeEvent evt);
}


package java.beans;
public class PropertyChangeEvent extends
   java.util.EventObject {
    public String getPropertyName()
    public Object getNewValue()
    public Object getOldValue()
}
```



## Java Beans Constrained Properties

- Listen to changes in property 'text'
  - accessors (to listen to all)
    - `public void addVetoableChangeListener(VetoableChangeListener vcl);`
    - `public void removeVetoableChangeListener(VetoableChangeListener vcl);`
  - accessors (to listen to just 'text')
    - `public void addTextListener(VetoableChangeListener vcl);`
    - `public void removeTextListener(VetoableChangeListener vcl);`
- Can listen for both propertyChange and vetoableChange



## Listening for Vetoable Property Changes

```
package java.beans;

public interface VetoableChangeListener
   extends java.util.EventListener {
    void vetoableChange(PropertyChangeEvent evt)
      throws PropertyVetoException;
}
```



## Events

- Listen for actions in 'Timer'
  - accessors ('Action', 'Tick')
    - `public void addActionListener(ActionListener al);`
    - `public void removeActionListerner(ActionListener al);`
    - `public void addTickListener(TickListener tl);`
    - `public void removeTickListener(TickListener tl);`
- PropertyChange & VetoableChange are just events
- Events can be inherited as well
  - many are for GUI events (focusGained, mouseOver, keyDown, …)



## Event Sets

- Events come in sets
  (all delivered as method calls on a single event listener interface)
  - action
    - actionPerformed
  - component
    - componentHidden
    - componentMoved
    - componentResized
    - componentShown
  - focus
    - focusGained
    - focusLost
  - text
    - propertyChange
    - vetoableChange
  - propertyChange
    - propertyChange
  - vetoableChange
    - vetoableChange



## Introspection

- Visual design tools use the class
  - java.bean.Introspection which manufactures a java.bean.BeanInfo class

  to learn about a bean.
  - Reflection and design templates
    - java.lang.reflect package

    Enquires about a bean's methods (names and parameters) and attempts to pattern match with aforementioned methods.
  - Explicit specification
    - Explicitly supply a BeanInfo class
      - » if bean is called 'Timer', attempts to load 'TimerBeanInfo'

## BeanInfo

- Can determine
  - each feature
    - display name, short description, expert, hidden, preferred
  - if the bean has a GUI
    - don't use, avoid, needs, ok
  - properties
    - types
    - string conversions
    - custom property editors (java.awt.Component)
  - events
    - sets and individuals names
    - what methods to call to register events
  - icons to use (16x16 and 32x32)
    - for use in visual development environment