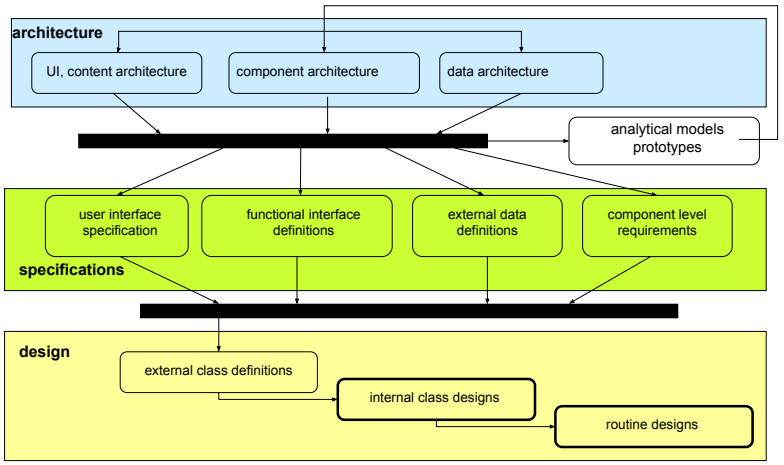
Component Level Design

- Routines
  - why and how we develop routines
  - elements of good routine design
  - algorithms, patterns, and tricks
  - creative table use
- Representing Routine Designs
  - pseudo-code
  - UML interaction/swim-lane diagrams

Model Hierarchy/Succession



Where do routines come from?

- many are already defined for us
  - our public methods (external entry points)
- some emerge naturally from our approach
  - just as ADTs emerge from problem domain
  - some methods and functions will be obvious
  - easier to specify because they are private
- many (most?) are artifacts of our solution
  - we create them to simplify the implementation
  - routines can do this in many ways

Why create a new routine?

- creating useful private pseudo-classes
  - create better abstractions to work with
  - we may even derive private sub-classes
- detail encapsulation
  - move complex sequences out of main code
  - segregate portable from non-portable code
  - hide/wrap global data structures
- centralize a recurring computation
  - one copy of an oft-repeated code sequence
  - enable interception of key operations

elements of good routines

- simplicity and clarity
  - obvious what routine does, how to use it
- good abstraction is still important
  - a well thought out function is easier to use
- information hiding is still important
  - avoid shared data, distributed algorithms
  - interactions mean complexity and bugs
  - encapsulate nasty details within a routine
- cohesion is still valuable
  - shorter routines are easier to understand

All code is not created equal!

- most code is fairly obvious
  - sequential steps in an obvious process
  - implementing well specified decision tree
- some code is complex, subtle, and critical
  - precise data transformations (e.g. DCT, DES)
  - manipulating structure-critical shared data
  - large, common, performance critical ops
  - complex operations or error handling
- these require more care and process
  - research, design, document, review, verify

What needs to be documented?

- purpose, parameters, functionality, returns
  - so code readers understand what calls do
  - so code writers know how/when to use it
- key assumptions, requirements, issues
  - better understand how/why routine works
  - must be understood to write correct code
- non-obvious decisions and algorithms
  - rationale and overview for reviewers
  - bring maintainers & testers up to speed
  - valuable for original creator as well

Routine Level Designs

- all routines are not simple
  - many embody complex algorithms
  - many cases to handle, many decisions
- such designs must be put in writing
  - help designer flesh out, record the design
  - present design to others for review
  - basis for implementation, white-box testing
  - a tool for future training and maintenance
- such designs can still be high level
  - they need not spell out simple/obvious steps

Representing Routine Designs

- there are many possible representations
  - summary comments: e.g. Doxygen
  - prose: e.g. pseudo-code
  - graphical: e.g. UML activity or state diagrams
  - tabular: enumerating cases and handling
  - formal: e.g. Object Constraint Language
- none is intrinsically superior to the others
  - but each has advantages for some problems
  - some may have development tool support
- Choose one that makes sense
  - but, “when in Rome, do as the Romans do”

Pydoc Class Docstring

```
“””
Luhn Sums
author: Hans Peter Luhn

The first level of validation for credit card numbers is a check-sum,
based on the Luhn (or mod10) algorithm:
 * sum the digits
 * double every other digit (mod 10 + 1 in case of overflow)
 * choose lowest-order digit so that the sum adds to 0 (mod 10)
This class contains methods to create, sum and verify such numbers.
“””
```


Doxygen Method Prologue

```
/**
 * Determine whether or not a region is within the defined mesh
 * @param center … location of region in question
 * @return boolean … true if center is within mesh
 *
 * Note: point-inside-polygon is expensive, we use cheap heuristics
 *       1. examine NUM_NEIGHBORS nearest MeshPoints
 *       2. determine the enclosing square
 *       3. see if we are within the max/min x/y coordinates
 *       4. see if mean-distance is wider than the square
 */
private boolean withinMesh( MeshPoint center ) { ...
```


Pseudo Code

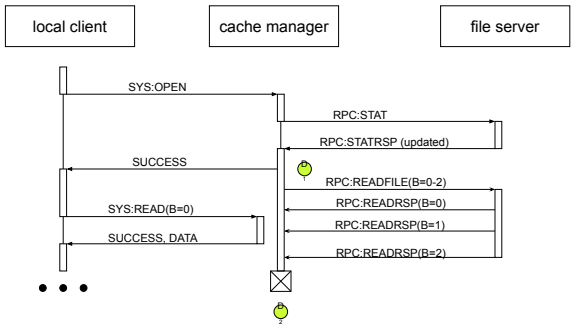
```
If local request
    find the record
else
    do remote

if error
    return failure

create new transaction
    including record
return transaction ID


find record:
    hash the key
    run down the chain
    if end of chain
        allocate new record
        label with this key
    return record pointer
```

```
do remote:
    allocate request
    fill it out
    try 3x til response
        set timeout
        send request
        await response
    extract completion info
    if successful
        allocate new record
        insert into cache
        return record pointer
    else
        translate error
        return error
```

(pseudo-code)

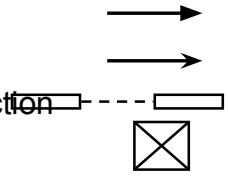
- higher level of abstraction than code
  - programming language independent
  - can be written at the level of intent
- good for roughing out an algorithm
  - faster to write
  - easier to refine/evolve than code
  - easily translated into code
- good for design reviews
  - faster to read and review
  - still contains key algorithmic elements

UML Object Interaction Diagram

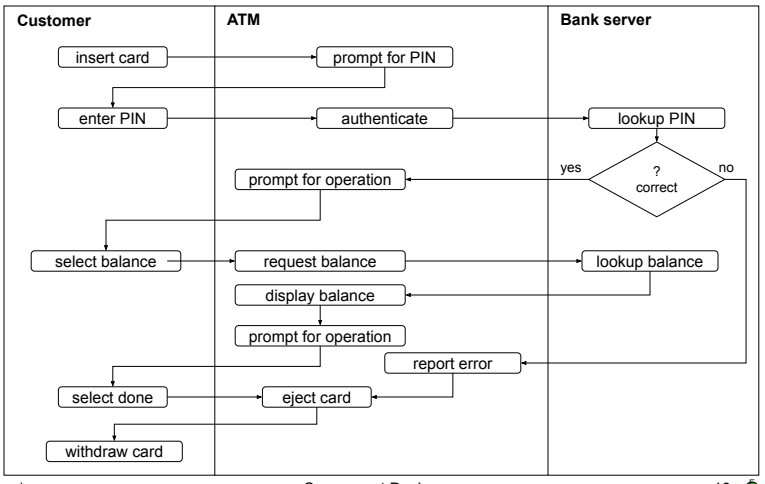


(Advanced Interaction Diagrams)

- Describe system component interactions
  - collaborations between objects
  - remote procedure calls and messages
- Rich vocabulary for describing interactions
  - descriptions of messages and requests
  - synchronous (procedure call)
  - asynchronous (message)
  - active/blocked threads
  - thread creation and destruction



UML Swim-lane Diagrams



(UML Swim-Lane Diagrams)

- combine interaction and activity diagrams
- describe multi-threaded flow of control
  - remote procedure call and return
  - asynchronous message exchanges
- parallel threads in parallel columns
  - each with its own activity diagram
- horizontal lines represent messages
  - from sender to receiver
- horizontal bars represent joins
  - awaiting reception of a message

So many models …

- There are many models to choose from:
  - use case diagrams
  - interaction diagrams
  - activity and swim-lane diagrams
  - state diagrams
  - data flow models
- Each shows very different things
  - which one should we choose?

Team Exercise

- Consider a design-worthy component:
  - what makes it challenging or interesting?
- Consider design representations:
  - prose description, doxygen, pseudo-code, UML diagrams, decision tables
- Answer:
  - What story are you trying to tell?
  - Which form would best tell that story?
  - What makes this the best choice?
  - What else in your project would demand a different choice?

Creative Table use

- data uses are fairly obvious
  - information about instances of something
  - where *n* is a fixed/bounded scalar/vector
- tables can also represent algorithms
  - functions on bounded integer(s) domain
  - separating algorithms from parameters
  - encodes the algorithm (not data, but actions)
  - a federation mechanism (in non-OO systems)
- often faster, smaller, more maintainable

(functions on integer domains)

- direct table look-up
  - a-periodic function, bounded, dense domain
    - int daysPerMonth[month]
    - double insRate[age][gender][smoke][married]
- fudged keys
  - transform a wide domain to bound it
    - max(min(66, Age), 17)
- map a sparse domain into a dense one
  - a faster & simpler alternative to hashing
    - itemDescr[codeMap[partNumber]]

separate code from parameters

- simplify code by pulling out parameters
  - simplify parameters by separating from code
- change parameters w/o changing code
  - parameter table can be read from a file

```
struct {
        int minScore;
        char *grade;
} gradeTable[ ] = {
        95,   “a+”,
        88,   “a”,
        78,   “b”,
        68 ,  ”c”,
        58,   “d”,
        0,    “f”
};
```

```
char *getGrade( int score ) {
        int  i;  /* index into gradeTable */

        /* gradeTable assumes score is positive */
        if (score < 0) return(“f”);

        for( i = 0; score < gradeTable[i].minScore; I++) {}
        return( gradeTable[i].grade );
}
```


table driven code

- values in table represent “actions”
  - usually encoded in some state-language
  - code is a state-language interpreter

```
int stateTable[][NUMEVENTS] = {
/* state      e0     e1     e2     e3  */
/* 0 */        0,     0,     1,    -1,
/* 1 */        1,     1,     1,     2,
/* 2 */        2,     1,     3,    -1,
/* 3 */        3,     3,     0,    -1
};
action_t actionTable[][NUMEVENTS] = {
/* state      e0     e1     e2     e3  */
/* 0 */        A,     A,     C,     X,
/* 1 */        A+X, A+X,  F,     C,
/* 2 */        0,     0,     A+C,  X,
/* 3 */        0,     0,     F+C,  0
};
```

```
int state = 0;        /* initial state */

while (state >= 0) {
        /* get the next input event     */
        event_type = getEvent();

        /* actionTable tells us what to do    */
        doAction(actionTable[state][event_type]);

        /* stateTable tells us our next state */
        state = stateTable[state][event_type];
}
```


Algorithms, Patterns, and Tricks

- know many types of algorithms
  - list maintenance, traversal, searches
  - hashing, sorting, comparing, lexing, parsing
- know many approaches
  - calls, messages, call-backs, pub-sub, threads
  - semaphores, events, signals, exceptions
  - serialized types, locking, transactions, leases
  - caching, guess pointers, table-driven, lazy
- understand <u>how</u> and <u>why</u> each works
  - they will give you inspiration and alternatives

For Next Lecture

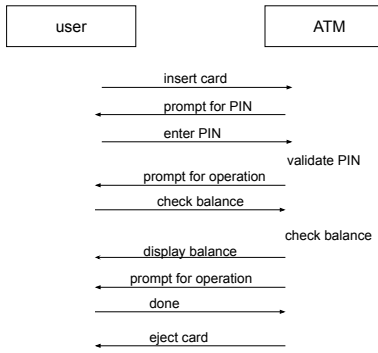
- McConnell: pages 103-104 (review)
  - comments on design patterns
- Garlan: Software Architecture, chapter 3
  - overview of common processing models
- Wikipedia: Client-Server Model
- Object Oriented Design: design patterns
  - creational: *singleton, object-pool*
  - behavioral: *iterator, observer, visitor, strategy*
  - structural: *adapter, bridge, proxy*
  - synchronization: *events, locking, leases, transactions*

Supplementary Slides

Using UML Interaction Diagrams

- Interaction Diagrams show interactions
  - users interacting with system components
  - interactions between system components
  - collaborations between object instances
- They can be used descriptively
  - to illustrate how a system will work
- They can be used prescriptively
  - to define expected behavior
- They are not for expressing algorithms

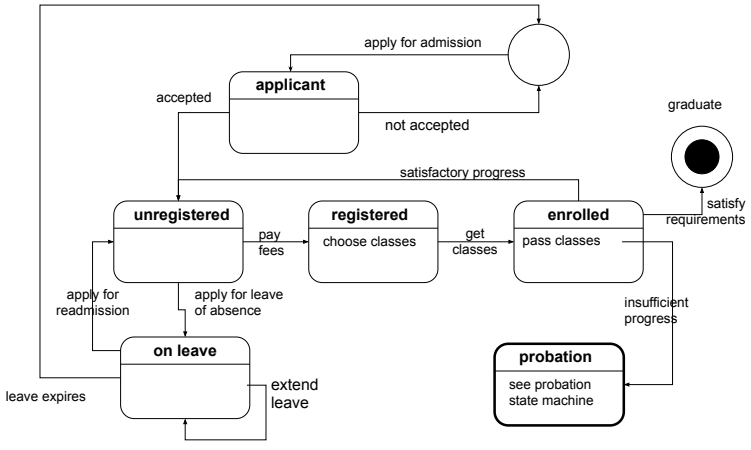
UML User Interaction Diagram*



(UML Interaction Diagrams)

- Describe interactions
  - between multiple actors
  - between actors and the system
  - typically one diagram per task or scenario
- Simple and intuitive representation
  - easy to draw, easy to understand
- Excellent for behavioral requirements
  - illustrative sample usage scenarios
  - additional detail for a use-case or story card

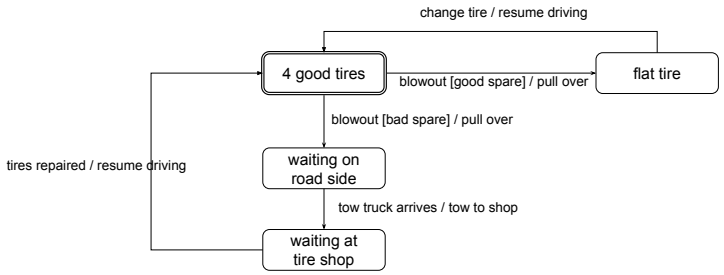
UML State Diagrams



(UML State Models)

- describe state/transition models
  - where events drive state changes
- similar to activity diagrams
  - activity boxes have two compartments
    - state name in the top portion
    - processing steps in the bottom portion
  - arrows represent state transitions
    - from previous state, to next state
    - labels describe conditions triggering the transition
    - processing steps can also be placed on lines

Finite State Machine*



(Finite State Machine Transitions)

- UML defines three parts to an arc label
  triggering event **[** *guard condition* **]** */ action*
- Where
  - *triggering event* is the event that will cause this transition
  - *guard condition* is a boolean test that determines whether or not this arc will be followed
  - *action* is an action that the system will take before entering the next state.
- These make it possible to directly translate traditional finite state machines in UML

macros

```
#define BoundsCheck(s,x,y,z)
    {if (x<y || x>z) log_error("bounds",s)}
```

- compile-time functions
  - expanded at compile time, duplicated code
- advantages
  - can be changed by compile time options
  - faster, no procedure calls, stack maintenance
- disadvantages
  - multiple copies, take up more space
  - can't have local storage (static or dynamic)