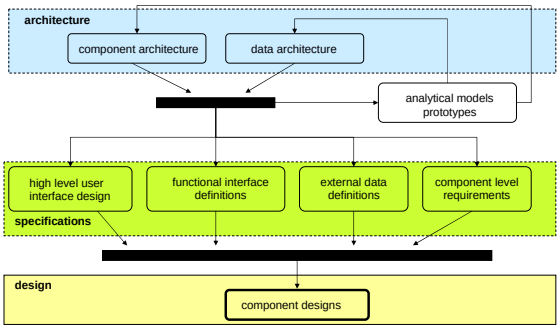
Component Level Design

- Routines
  - where routines come from
  - elements of good routine design
  - algorithms, patterns, and tricks
- Representing Routine Designs
  - pseudo-code
  - UML interaction diagrams
  - UML swim-lane diagrams
  - UML state diagrams, finite state machines
  - Data Flow models

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 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

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 how and why they work
  - they will give you inspiration and alternatives

Representing Routine Designs*

- there are many possible representations
  - 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”

(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
- such designs can still be high level
  - they need not spell out simple/obvious steps

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
```

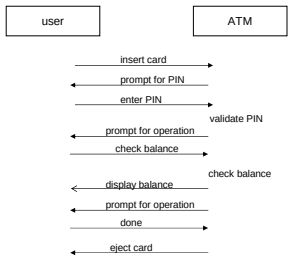

(pseudo-code)

- 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

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

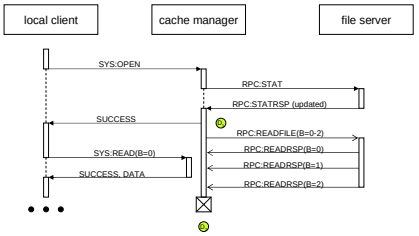
UML User Interaction Diagram*



(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

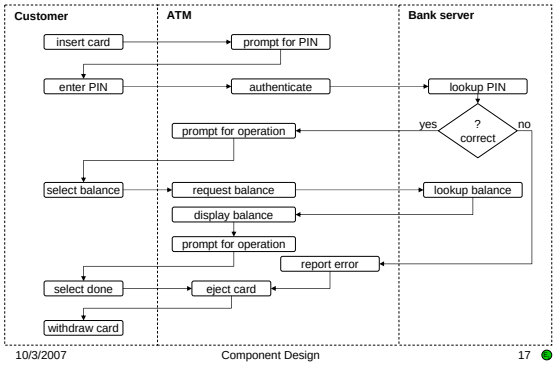
UML Object Interaction Diagram*



(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 

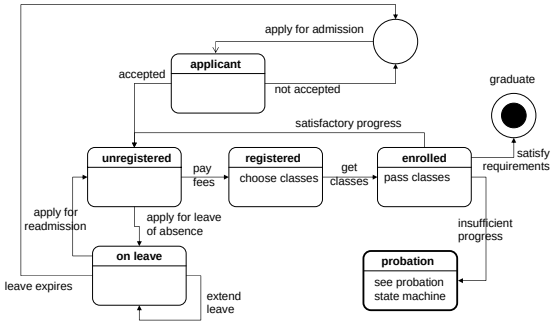
UML Swim-lane Diagrams*



(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

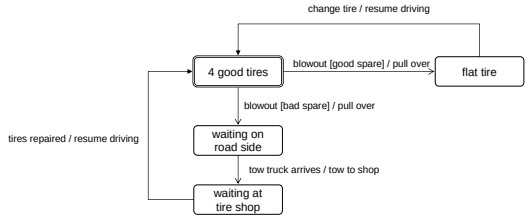
UML State Diagrams*



(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

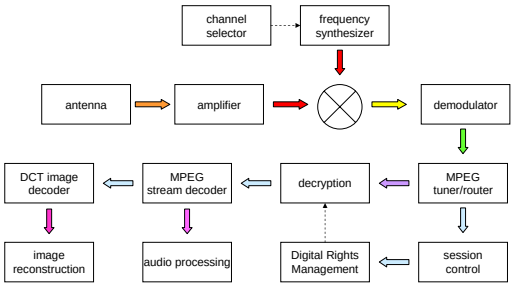
Finite State Machine*



(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

Digital TV Data Flow Model*



(Data Flow Models)

- UML is designed for OO-software
  - components are comprised of related objects
  - objects have properties and methods
  - methods have associated algorithms
  - interactions are via discrete messages
- Data Flow Models take different view
  - follow input, through processing, to output
  - all components are processing in parallel
  - processing is continuous rather than discrete
  - view system in terms of data transformations

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?

mid-term – 20% of course grade

- format
  - closed book
  - 10 questions w/short essay answers
  - 1 more difficult extra credit problem
- scope
  - all lectures & reading assignments thus far
- studying for this exam
  - review key learning objectives
  - posted lecture slides (+ Q&A slides)
  - all on course web site