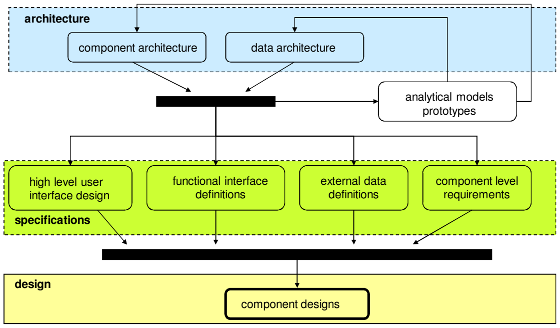
Component Level Design

- 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
  - Data Flow models

Model Hierarchy/Succession



Where do <u>routines</u> 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 <u>private</u>
- 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 <u>private</u> 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
  - operations on huge lists
  - backing out of partially completed operations
- the latter require special attention
  - research, design, review, verification, …

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

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”

Pseudo Code*

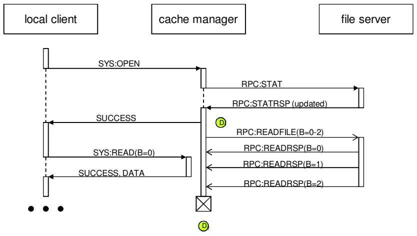
```
    If local request
            find the record
    else
            do remote
    if error
            return failure
    create new transaction
            including record
    return transaction ID

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 <u>level of intent</u>
- 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

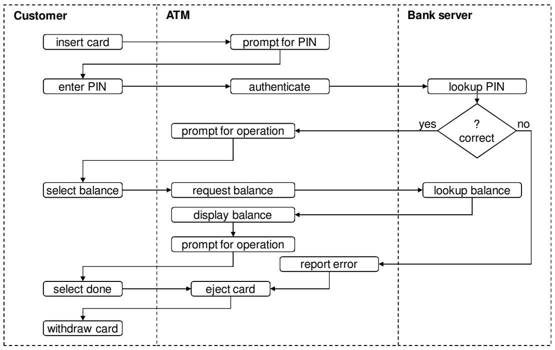
UML Object Interaction Diagram*



(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 ☒

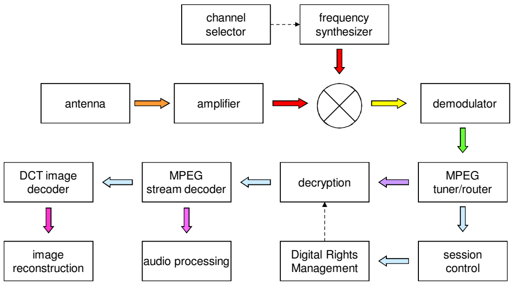
UML Swim-lane Diagrams*



(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

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

Creative Table use

- data uses are fairly obvious
  - information about *n* instances of something
  - where *n* is either fixed or bounded
- tables can also represent algorithms
  - functions on a limited integer range
  - separating code from its parameters
  - the table encodes the algorithm
  - a federation mechanism (in non-OO systems)
- often faster, smaller, more maintainable

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

- the table actually contains the algorithm
  - 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];
}
```

Exercise

- Form teams:
  - prose, pseudo-code, UML diagrams, decision tables, some formal language
- Answer:
  - What are strengths of this representation?
  - What are weaknesses of this representation?
  - What have you done/seen where this would be a good choice, and why?

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

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?

For Next Lecture

- McConnell 23
  - good overview of debugging techniques
- Kampe: Forensic Debugging
  - finding problems w/the available information
- Kampe: Root Cause Analysis
  - why did we create this bug?
- Wikipedia: Defect Tracking
  - overview of defect tracking systems
- Black: Writing a good bug report
  - better reports lead to quicker debugging

Supplementary Slides

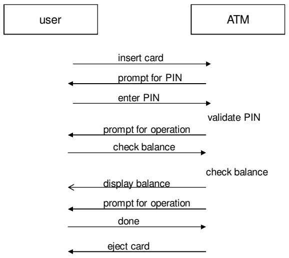
(Routine Level Designs)

- 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

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*



user

ATM

insert card

prompt for PIN

enter PIN

validate PIN

prompt for operation

check balance

check balance

display balance

prompt for operation

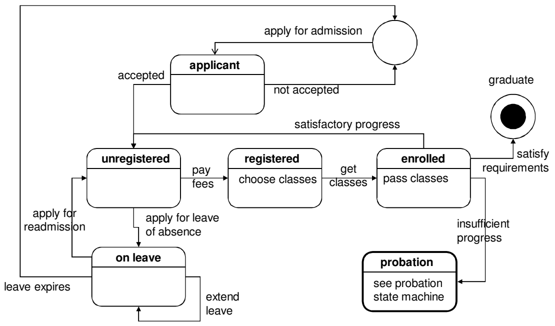
done

eject card

(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*



apply for admission

applicant

accepted

not accepted

graduate

satisfactory progress

unregistered

pay fees

registered

choose classes

get classes

enrolled

pass classes

satisfy requirements

apply for readmission

apply for leave of absence

insufficient progress

on leave

leave expires

extend leave

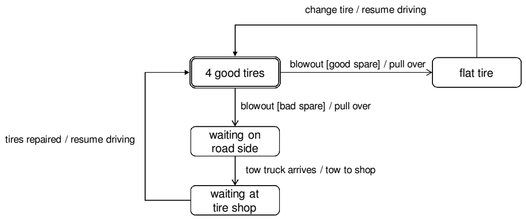
probation

see probation state machine

(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

Functions on an integer range

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

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)