CS3215: Software Engineering Project

LN set #6: Software methods

1. Flexible design: table-driven approach
2. Evaluating and documenting design decisions
3. Use of UML diagrams
4. Testing

Flexible design with table-driven technique

Query validation – table-driven solution

What does it take to validate a query?

assign a; while w; variable v;

Select a **such that** Parent* (w, a) **pattern** a (v, _) **with** v.varName="x"

- are design entities assign, while and variable defined in the program design model?
- are arguments to Parent, assign correct?
- does variable have an attribute varName?

Example: what's wrong with this query?

procedure p, q; assign a; cluster c;

Select p **such that** Calls(p,q) **and** Modifies (p, a) **and** Calls(p)

and ReferesTo (p, "x") **unlike** Modifies()

Query validation

A simple-minded solution:

switch (relationship) {

case Calls: expect two arguments; each argument should be either procedure synonym or '_'

case Next: expect two arguments; each argument should be either statement number or synonym of statement, assign, if, while or '_'

etc.

}

Program model definition tables

1	program
2	procedure
3	stmtLst
4	stmt
5	assign
6	etc.

Table 1. Entity table - EntTable

Relationship	# arguments	type of arg 1	type of arg 2	type of arg 3
Calls	2	procedure	procedure	nil
Calls*	2	procedure	procedure	nil
Modifies	2	procedure	variable	nil
	2	stmt, assign, call, while, if, stmtLst	variable	nil
etc.				

Table 2. Relationship table - RelTable

CS3215 Set #6 Methods

5

Table-driven query validation

for each R in a query {

rel = getRelTabIndex (R)

if (#args ≠ RelTab [rel, 2]) Error()

if (arg1 ∉RelTab [rel, 3] then Error()

if (arg2 ∉RelTab [rel, 4] then Error()

… }

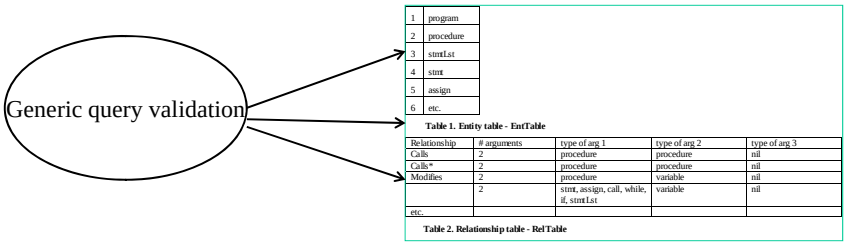
- R is a relationship referenced in a query,
- #args is the number of arguments of R as it appears in the query,
- arg1 is the first argument of R, etc.
- Validation code checks if the actual references in a query agree with their respective definitions in the tables.

CS3215 Set #6 Methods

6

Table-driven technique: summary

- Analyze a design problem and try to capture (some part of) problem description in declarative, non-procedural form (this is "data")
  - Program model definitions for query validation problem
- Work out a design solution as generic, table-driven algorithm



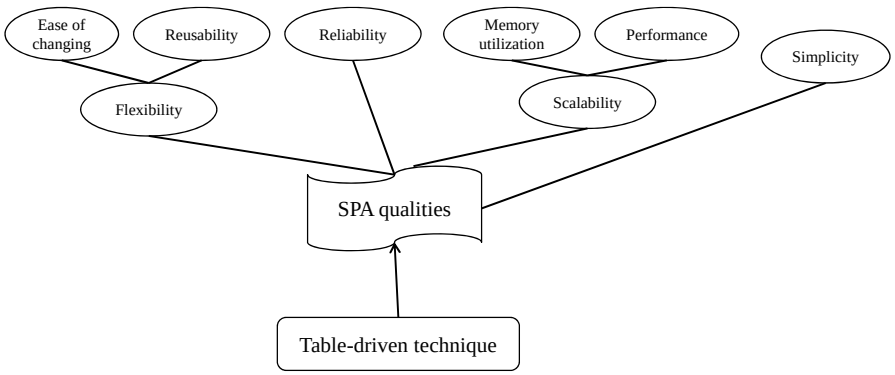
1	program
2	procedure
3	stmtLst
4	stmt
5	assign
6	etc.

Table 1. Entity table - EntTable

Relationship	# arguments	type of arg 1	type of arg 2	type of arg 3
Calls	2	procedure	procedure	nil
Calls*	2	procedure	procedure	nil
Modifies	2	procedure	variable	nil
	2	stmt, assign, call, while, if, stmtLst	variable	nil
etc.				

Table 2. Relationship table - RelTable

Which qualities of SPA the table-driven technique helps us address?



Evaluating design decisions

There are many ways to design SPA

– how do we make right design decisions?

Architectural design decisions

- High level decomposition of SPA into functional components
  - Evaluation: high cohesion, low coupling
- Abstract PKB API
  - The choice of API operations for design abstractions
    - *Completeness*: Do I have all API operations that are needed?
    - *Convenience*: are they convenient to use?
  - Documentation of PKB API

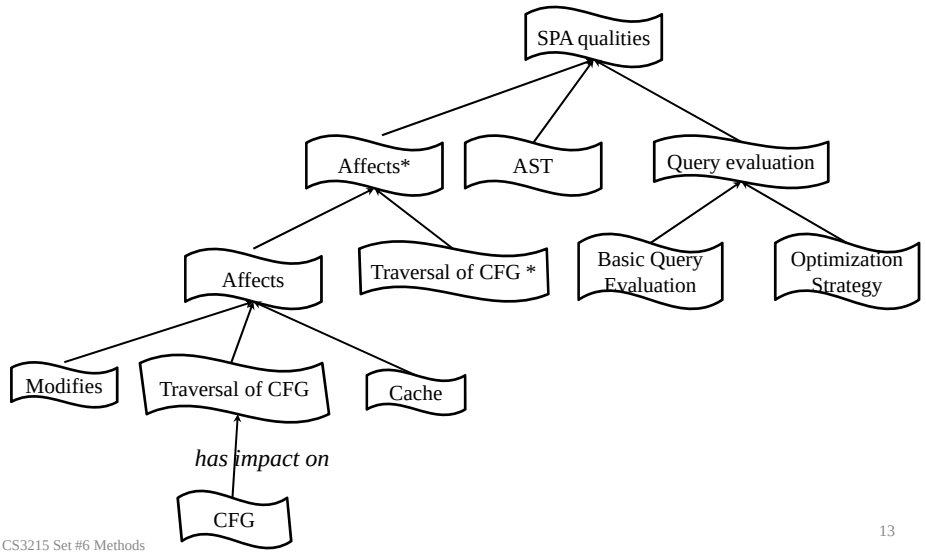
Detailed design decisions

- Selection of data structures for
  - Design abstractions in PKB
  - Queries
- Design of algorithms
  - Computing Next*, Affects, Affects*
  - Query evaluation
- Supporting techniques
  - Mappings between AST and source program
  - Table-driven design
  - Many others

Modifies (stmt, variable)

- Factors and design goals to consider:
  - Memory utilization
  - Performance of API operations
  - Simplicity of implementation
- Alternative design solutions:
  - Linked list
  - Byte array
  - Bit vector (various containers in STL)
- Evaluation:
  - Explain how design solutions affect design goals
  - Justify your choice of design solution

Inter-dependent design decisions



Documenting design decisions

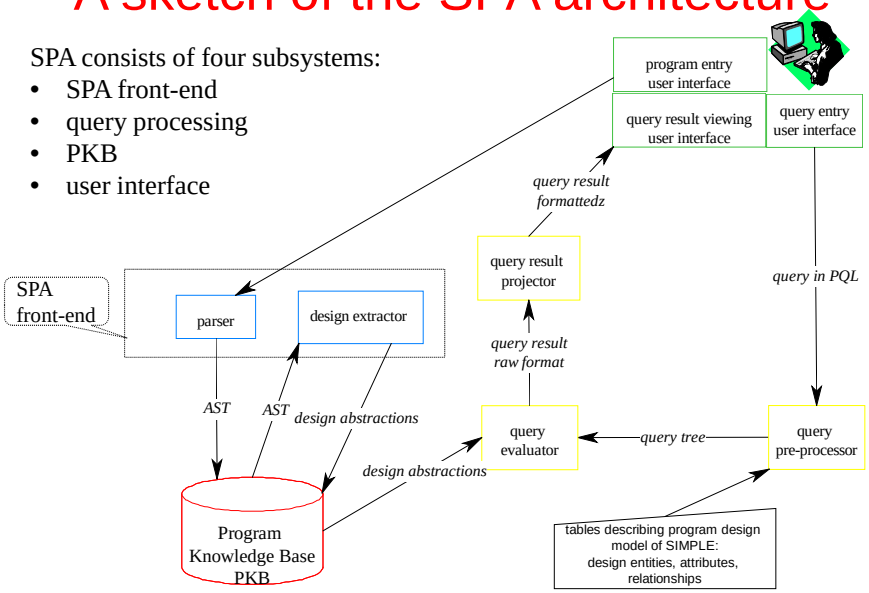
- Depict inter-dependencies among design decisions
- For each design problem:
  - State design goals that matter
  - Consider alternative design solutions
  - Evaluate design solutions in view of design goals
  - Justify your choice of design solution
  - Document the above process
- Use Big O notation to describe complexity of algorithms
  - Next*, Affects, Affects*, query evaluation

Use of UML diagrams

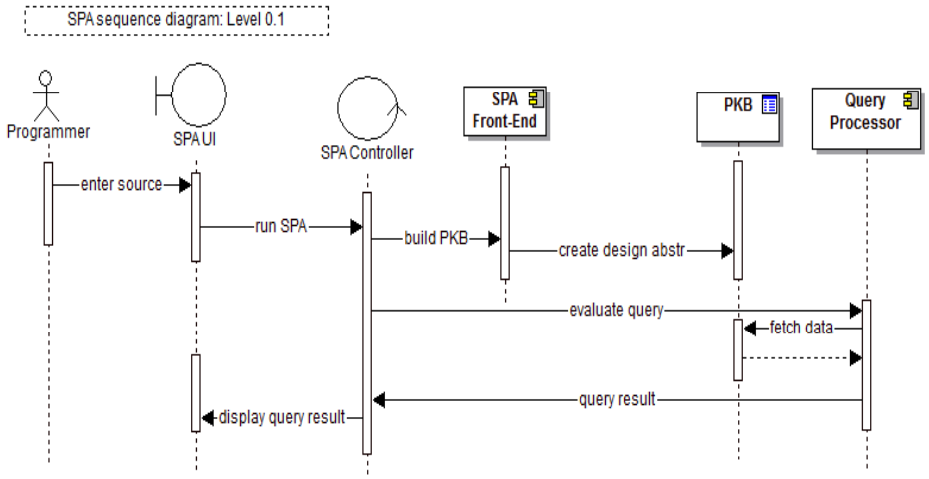
- Class diagrams and sequence diagrams
  - Check examples in Handbook
- Use activity diagrams to describe complex algorithms
- Always be clear about the purpose of a diagram, how you are going to use it
- Let it be one purpose per diagram (cohesion!)
  - *Common error*: using class diagram to describe both system structure and dynamic behavior
- Check modeling guidelines, Handbook Section 10.3

A sketch of the SPA architecture

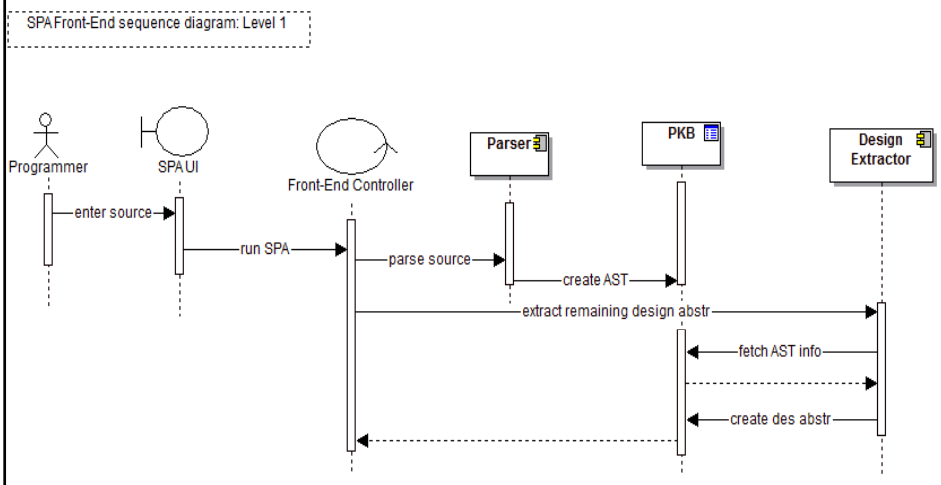
SPA consists of four subsystems:
- SPA front-end
- query processing
- PKB
- user interface



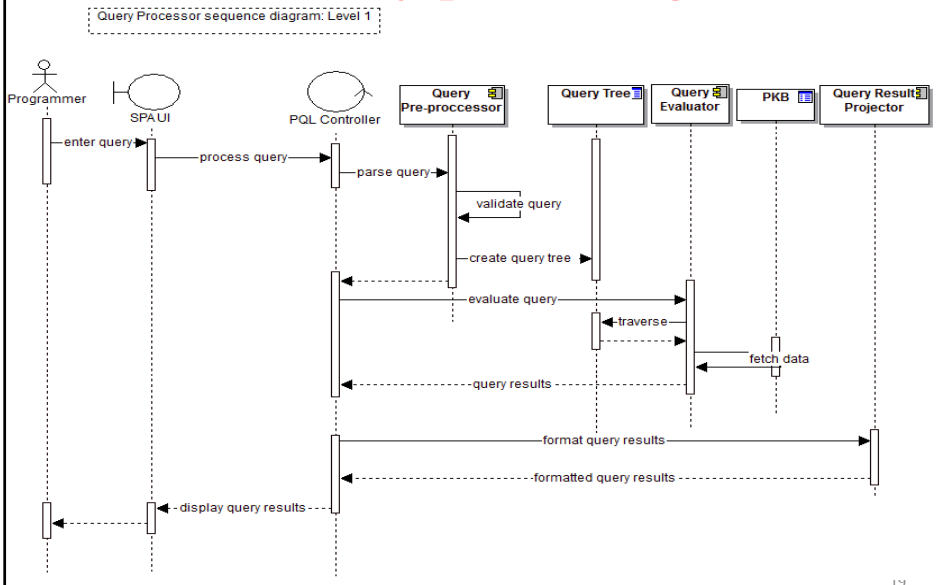
Showing detailed interactions



SPA front-end



Query processing



When sequence diagrams are useful

- Modeling SPA design:
  - Understanding how SPA works
  - Understanding responsibilities of components
  - Understanding chains of interactions among components
- Project management
  - Plan development task
  - Mark the progress of a project
- Testing

Testing

Testing

- To ensure that software meets requirements
- To find and eliminate software errors

Testing may take 40% of development time!

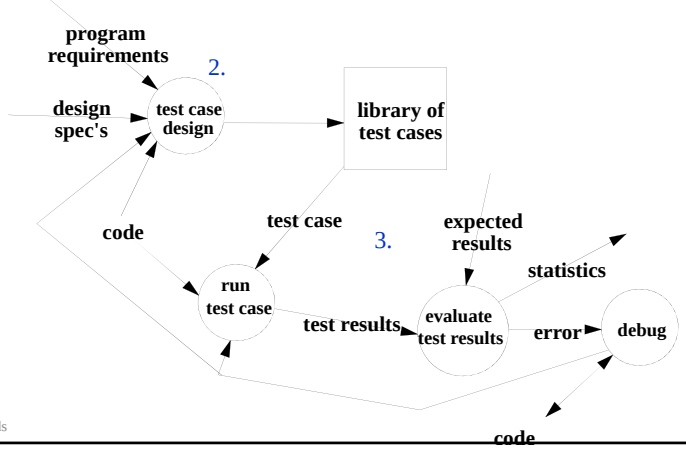
Have I done enough testing?

"Testing can only show the presence of errors"

testing is never complete and no amount of testing can prove program correctness

Testing lifecycle

1. Test planning, testing strategy
2. Test case design
3. Execution of test cases and evaluation of test results



Documenting test cases

- The purpose of a test case and description
- Required inputs to a program
- Expected results produced by a program
- Any other requirements for running a test case

Black-box and white box testing

- Black box testing:

Testing based on specifications, without looking into code

  - Does a program meet requirements?
  - Are all the required functions provided and fully operational according to specs?

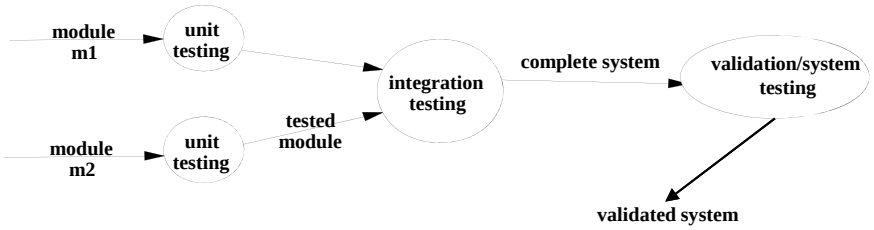
- White box testing:

Testing the way code is implemented

  - Have all program modules been called?
  - Have all statements been executed?
  - Have all braches and control paths been covered?

Testing strategy

- What and when I should test?
  - When to test parts of a program?
  - When to test the entire program?
- Testing from inside out:
  - Individual modules - unit testing
  - Some modules together -integration testing
  - The whole system – validation/system testing

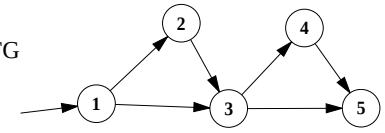


Unit testing (cppUnit)

- Units are functions and class methods
- White-box tests, written by developer
  - Before, during or just after writing code
  - Is program logic correct in respect to specs?
- Force program to execute:
  - each statement
  - each branch (edge in CFG)
  - important control flow paths (CFG)
  - do loops terminate properly?
- Test legal and illegal inputs (exceptions)

Test coverage

CFG

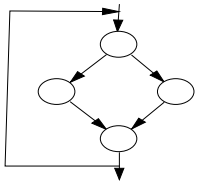


- One test case can cover all statements: 1, 2, 3, 4, 5
- Two test cases cover all branches: 1, 2, 3, 4, 5 1, 3, 5
- Four test cases cover all control paths:

  1, 2, 3, 4, 5
  1, 3, 5
  1, 2, 3, 5
  1, 3, 4, 5

Test coverage

- We cannot cover all the paths during testing:



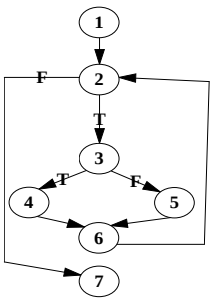
No. of iterations	No. of control paths
0	2
1	4
2	8
....	....
10	2048
N	2**(N+1)

- Try to identify and cover “essential” paths

Covering independent paths

- Start with a test case for typical execution path
- Add test cases as long as they traverse at least one node or edge not traversed before

```
(1)  a = 1;
     b = 1;
(2)  while (i < 100) {
(3)          c = i + 1;
             z = a;
             if (y < 0){
(4)             a =2;
                b = 2;
             else {
(5)             a = 3;
                b = 3; }
(6)  i = i + 1;
     y = y + c; }
(7)  z= a + b
```



Sample test cases

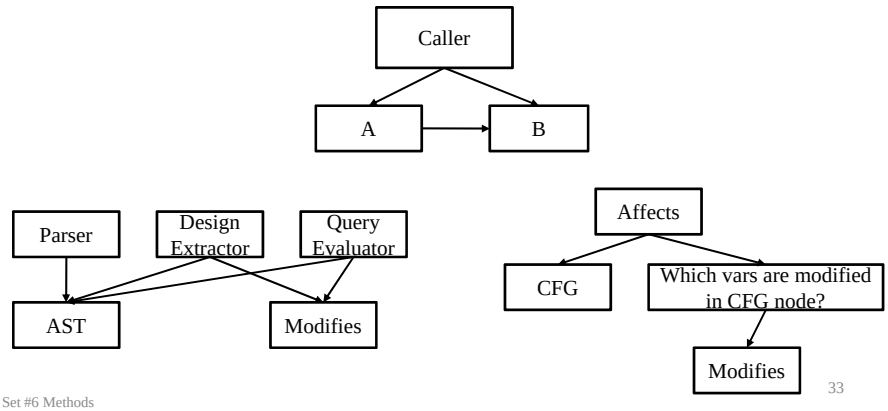
Test case 1, purpose: skip the loop
inputs: i >= 100; any value for y
expected results: a = 1; b = 1
Path covered: 1, 2, 7
Test case 2, purpose: execute loop with y < 0
inputs: i < 100; y < 0
expected results: (a = 2 or a = 3) and (b = 2 or b = 3)
Path covered: 1, 2, 3, 4, 6, 2, ..., 7
Test case 3, purpose: execute loop with y >= 0
inputs: i < 100; y >= 0
expected results: (a = 2 or a = 3) and (b = 2 or b = 3)
Path covered: 1, 2, 3, 5, 6, 2, ..., 7

Unit testing, cont.

- When to (re-)run unit tests?
  - As soon as you can compile code of a unit
  - After changes done to a unit
- What if changes of unit A may affect other units B, C, … ?
  - Have test cases whose execution invokes A, B, C
- Defensive programming helps in testing

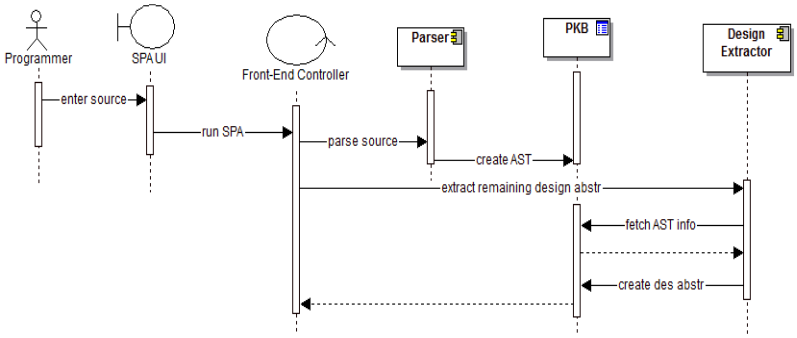
Integration testing

- Modules A and B were implemented separately
  - Will A and B work together ok?



Planning integration testing

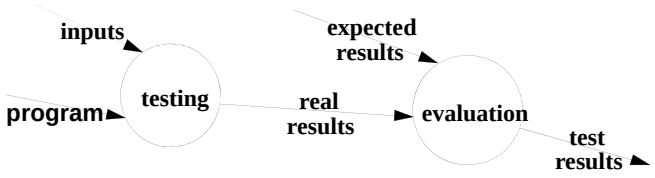
- Have I tested all important module interactions?



- Use sequence diagrams to plan integration tests
  - Refine sequence diagrams given in the Handbook

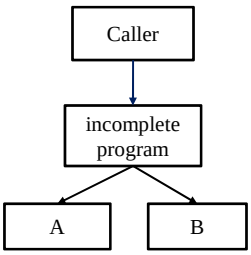
Running test cases

- We must have a program that compiles



Testing incomplete program

- To test, we must compile and run a program



- If A and B are not available, write *test stubs* to simulate the behavior of A and B
- If Caller is not available, write *test driver*

Validation/system testing

- Testing a complete system
- In SPA, different types of queries form system tests
  - Inputs: source in SIMPLE, query
  - Output: query results
- Auto-Tester automates system testing
  - Design tests once - execute many times
  - Quick validation after changes
- Accumulate test cases and execute them often throughout a project

--- The End ---