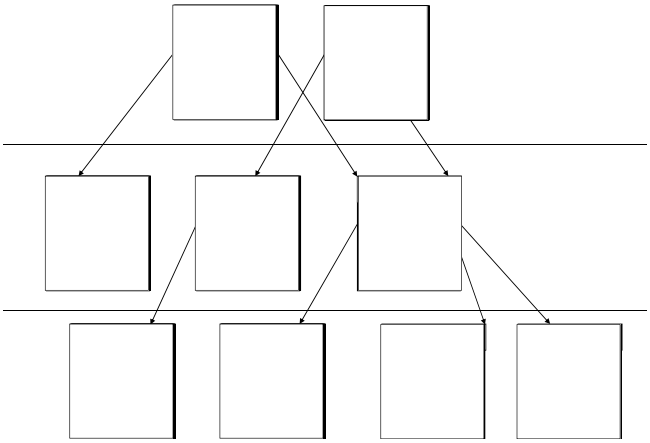
# Decomposition: Modules

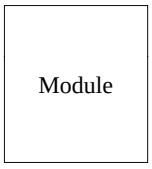
- What's wrong with this?

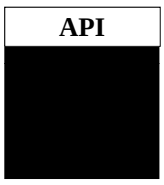




# CS3215: Software Engineering Project

**CS3215, LN set #4: Specifying APIs (Assignment #2)**




Public: interface

Hidden, Private: implementation

- We want to let others use our module without knowing all the module details - how?
- We want to change a module without affecting other modules – how?
- API: a description of what you can do with a module

# API Design

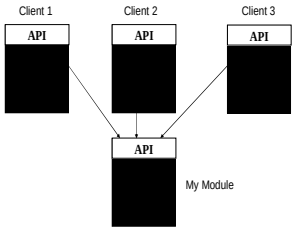
1. Decide about which operations should make it into the API
   - How will others use my module?
2. Document – communicate APIs
   - Must be understandable but also precise

- APIs are used many times by the clients
  - Good API will save time of many
  - Bad API will hinder productivity of many
  - Incorrect APIs may lead to project disasters
- Quality APIs are an essential part of architecture



# How to identify API operations?

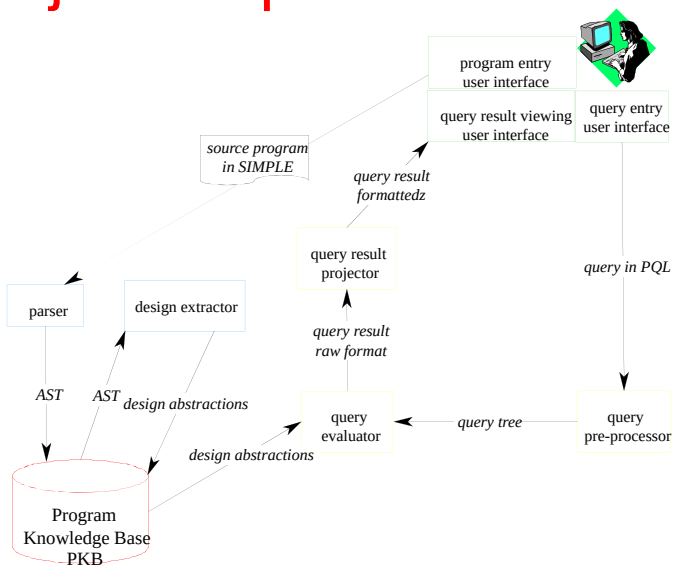
- Multiple clients using My Module:




  - I need know how different Clients use My Module

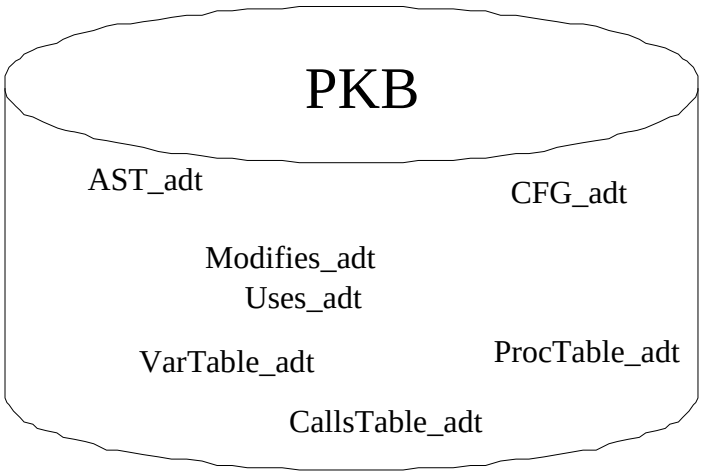
1. Work in pairs, with another student playing role of a Client
2. Ask third student to review API documentation

# Major components of an SPA






# What is stored in the PKB?



# Major ADTs in the PKB

- AST_adt - an abstract syntax tree
- CFG_adt - a control flow graph
- Var_adt - stores program variable info (a symbol table)
- ProcTable_adt - stores program procedure info (symbol table)
- CallsTable_adt - indicates procedure call relationship
- Modifies_adt - indicates which variables are modified in a given statement or procedure
- Uses_adt - indicates which variables are modified in a given statement or procedure



# Understanding associations among ADTs

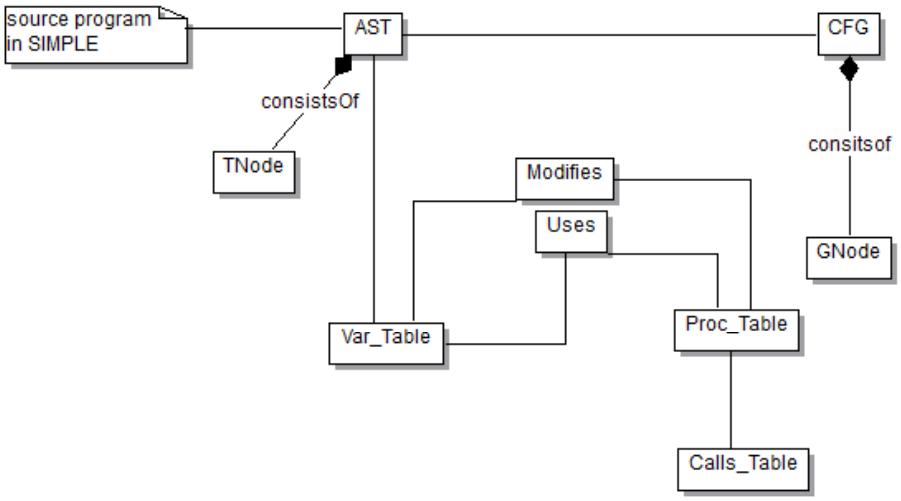


Table 1. Documenting mappings among ADTs in the PKB

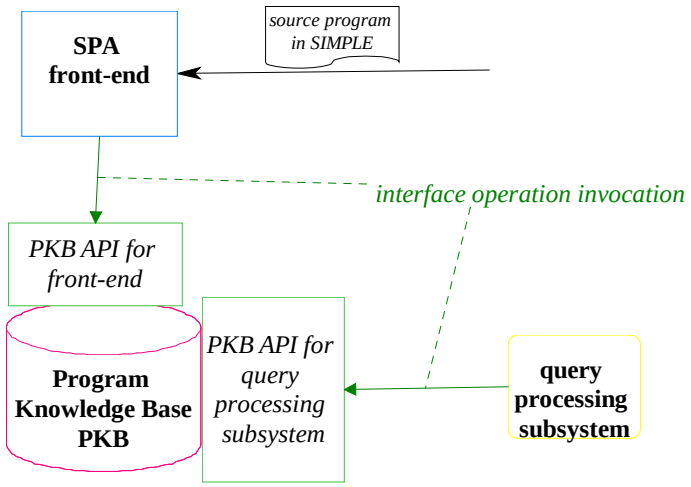
# Associations among ADTs

*Document associations among ADTs in the following way:*

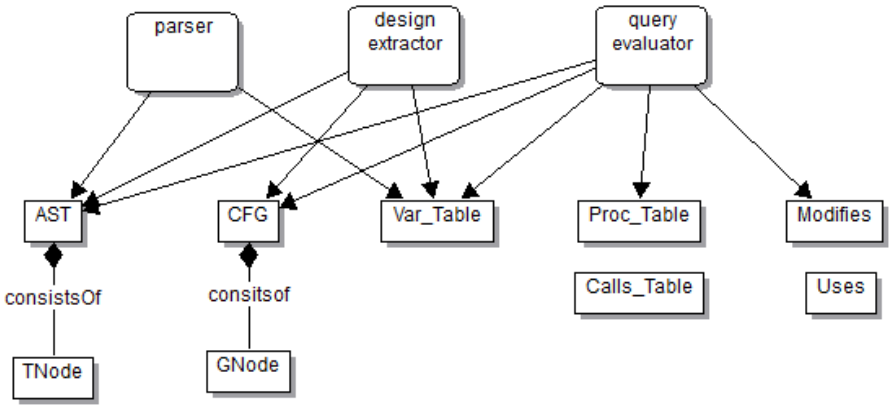
| Association | The meaning and role of association |
|---|---|
| mapping among source program and AST | for each statement number we shall be able to identify AST node that contains that statement and vice versa |
| mapping among AST and CFG | for each AST node we shall be able to identify a corresponding CFG node and vice versa |
| mappings from Proc_Table to Modifies and from Modifies to Var_Table | for each procedure we shall be able to identify to Modifies vector that indicates variables (mapping to Var_Table) modified in this procedure |
| etc. | |



# Who and how will use PKB?



# Identify APIs for different clients



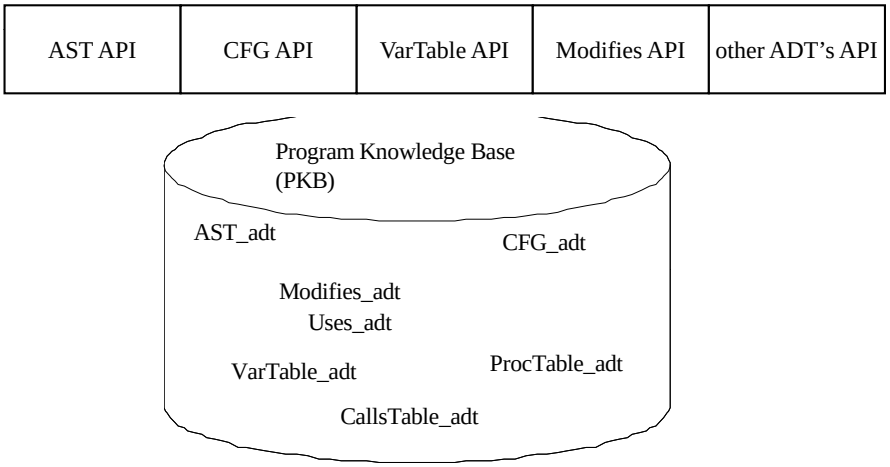



# Discover parser's API to AST

- Check a dialog in Handbook, Section 9.6:
- **Parser**: I need create AST. Can you provide me with simple but flexible means to do that? I definitely do not want to be concerned with how you implement AST.
- **AST**: That's great. I still consider a number of options of how to implement AST. So if you base your decisions on my implementation, you would have to wait. Also, if I decide to change my implementation later on, which is likely to happen we are bound to run into constant problems and lots of re-work. So I will give you AST API - a set of interface operations to AST. You will be able to work with AST using AST API.
- **Parser**: At the moment, you have not implemented AST and I have not implemented a parser. Shall we then just forget about implementation and come up with abstract AST API that logically makes sense, based on common sense understanding of essential properties of AST?

# Consolidated PKB API

- PKB API: a union of ADT APIs:






# How to describe API operations?

# Agree on API specification format

ADT name {

*Overview*: explain the rationale and responsibility of an ADT

*Public interface*: interface operations documented as follows:

*Operation header*:

returned-value operation-name (list of parameters)

– give names to parameters, specify types only if necessary

**Parameters* *(optional):*

*Description*: describe what the operation does

describe both normal and abnormal behavior

}



# API for Variable Table

- Variable Table stores program variables:

| index | variable name | descriptors |
|---|---|---|
| 1 | x | |
| 2 | y | |
| 3 | z | |

- Who and How we uses Variable Table?
  - Parser must insert variables to the Table
  - Others must get variable name stored at given index

# API for Variable Table

**VarTable{**

***Overview***: VarTable keeps the program variables

***Public Interface:***

**INDEX** insertVar (**VAR** v);

***Description***: If 'v' is not in the VarTable, inserts 'v' into the VarTable and returns its index.

**Else**, returns its index and the table remains unchanged.

**STRING** getVarName (**INDEX** ind);

***Description***: Returns the name of a variable at VarTable [ind]

If 'ind' is out of range, Throws: InvalidReferenceException

**INDEX** getVarIndex (**VAR** v);

***Description***: If 'v' is in VarTable, returns its index; otherwise, returns -1 (special value)

**INTEGER** getSize();

***Description***: Returns the total number of variables in VarTable



# Adding more rigor to API specs

**Bank Account {**

***Overview***: ….

***Public Interface:***

**void deposit (amount) ;**

***Requires***: amount > 0 ;

***Description***: balance' () = balance () + amount.

**void withdraw (Money amount) ;**

***Requires***: amount > 0 and

amount <= balance () + overdraftLimit () ;

***Description***: balance' () = balance () – amount.

**Amount balance () ;**

***Description***: returns the balance on the Account.

**before** operation

**after** operation

- ***Requires*** clause describes pre-conditions
- We use elements of formal notation to make description shorter, clearer

# Tips for designing APIs

Which operations do I need in interfaces of ADTs: AST, CFG, Modifies etc. ?

- work in pairs on specific interfaces:
  - parser to AST
  - design extractor to AST
  - design extractor to CFG
  - query evaluator to CFG
- independent review of interface documentation by a student not involved in designing a particular interface



# Tips for specifying APIs

- Adopt standard way of documenting APIs
  - naming conventions
  - unify descriptions across project
- Keep description possibly simple, concise, but precise, unambiguous, complete
  - API must be understandable for others!
- Implementation- and language-independent
- Use symbolic names for types: STRING, INT
- Use elements of formal notations in the API specification, only when it really helps

# Completeness and the right level of abstraction

*completeness of APIs*:

- complete set of APIs: can ADT's clients accomplish their tasks by consulting API of the ADT?

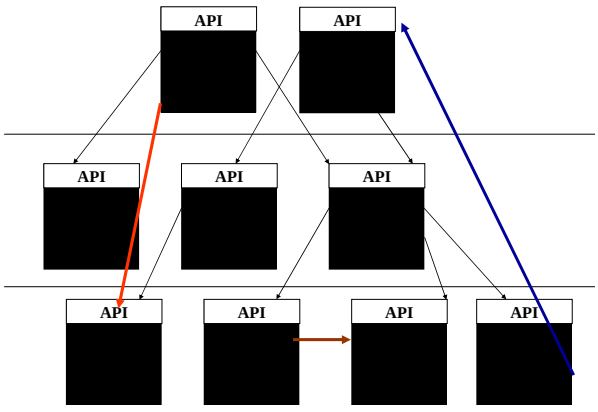
*the right level of abstraction of APIs*:

- at this stage, APIs should reflect only essential properties of ADTs
  - you will be able to add extra operations (e.g., to address optimizations, efficiency concerns) when you consider implementation of ADTs
- APIs should not make assumptions regarding implementation



# APIs and complexity

- Well **chosen**, clearly **documented**, and **stable** APIs are keys to complexity reduction




- At times you may need to compromise rules of good design

# Incremental development of APIs

- Submit initial APIs in assignment 2
- Discuss with your supervisor
- Submit complete PKB APIs (assignment 3)

**--- The End ---**