6.170 Quiz Review

Topics:

1. Decoupling
2. Data Abstraction
3. AF & RI
4. Iteration Abstraction & Iterators
5. OMs and Invariants
6. Equality, Copying, Views
7. Dynamic Analysis
8. Design Patterns
9. Subtyping
10. Case Studies

Decoupling

L2, L3, Ch 1, Ch 13:1-3, Ch 2

Decomposition

Division of Labor
Reuse
Modular Analysis
Localized Change

Top Down Design vs. Modularization

Decoupling

L2: Uses, Dependencies, Specifications, MDDs

Uses Diagram: Trees, Layers, Cycles

Reasoning
Reuse
Construction Order

Dependencies & Specifications; MDDs for

- Weakened assumptions
- Evaluating changes
- Communication
- Multiple implementations

Decoupling

L2: MDDs, Techniques

MDDs

- Specification parts
- Implementation parts
- Meets, depends, weak depends relationships

Techniques

- Façade: new implementation part between two sets of parts
- Hiding representation: avoid mentioning how data is represented

- Polymorphism: 'many shaped'
- Callbacks: runtime reference to a procedure

Decoupling

L3: Java Namespace, Access Control

Java Namespace

- Packages → {Interfaces, Classes} → {methods, named fields}

Access Control

- public: accessed from anywhere
- protected: accessed within package or by subclass outside of package
- default: accessed within package
- private: only within the class

Decoupling

L3: Safe Languages, Interfaces

Safe Languages

- One part should only depend on another if it names it
- Strong typing: access of type t in program text is guaranteed at runtime
- Check types at compile time: 'static typing'

Interfaces: more flexible subtyping

- Express pure specification
- Allows several implementation parts to one specification part

Decoupling

L3: Instrumenting a Program

- Abstraction by parameterization
- Decoupling with interfaces
- Interfaces vs. Abstract Classes
- Static Fields

Data Abstraction

L4, L5, Ch 3-5, Ch 9

Specifications

- Pre-condition (requires)
  - Obligation on the client (caller of the method)
  - Omitted: true; requires nothing
- Post-condition (effects)
  - Obligation on implementer
  - Cannot be omitted
- Frame condition (modified)
  - Describes which small state is modified
  - Omitted: modifies nothing

Data Abstraction

L4: Specification

- Operational specification: series of steps the method performs
- Declarative specification: do not give details of intermediate steps (preferable)

▪Exceptions & Preconditions (decisions)

- Preconditions: cost of check, scope of method
- Check via runtime assertions
- If violated, throw unchecked exception (not mentioned in specification)

Data Abstraction

L4: Specifications

▪Shorthands

- Returns: modifies nothing, and returns a value
- Throws: condition and exception both given in throws clause; modifies nothing

▪Specification Ordering: A specification A is at least as strong as a specification B if

- A's precondition is no stronger than B's
- A's postcondition is no weaker than B's, for the states that satisfy B's precondition
- (can always weaken the precondition; can always strengthen the postcondition)

Data Abstraction

L4: Specifications

- Judging specifications
  - Coherent
  - Informative
  - Strong enough
  - Weak enough
- Crucial firewall between implementer and client

Data Abstraction

L5: Abstract Types

- Data abstraction: type is characterized by the operations you can perform on it
- Mutable: can be changed; provide operations which when executed cause results of other operations on the same object to give different results (Vectors)
- Immutable: cannot be changed (Strings)

Data Abstraction

L5: Abstract Types

▪Operations (T = abstract type, t = some other type)

- Constructors: t → T
- Producers: T, t → T
- Mutators: T, t → void
- Observers: T, t → t

▪List example

Data Abstraction
L5: Abstract Types

▪Designing an Abstract Type

- Few, simple operations that can be combined in powerful ways
- Operations should have well-defined purpose, coherent behavior
- Set of operations should be adequate
- Type may be generic (list, set, graph) or domain specific (street map, employee db, phone book), but not both

Data Abstraction
L5: Abstract Types

▪Representation: class that implements an abstract type provides a representation

▪Representation Independence

- Ensuring that use of an abstract type is independent of representation
- Changes in representation should not affect using code

▪Representation Exposure

- Representation is passed to the client
- Client is allowed direct access to representation
- Need careful programming discipline

Data Abstraction
L5: Abstract Types

▪Language Mechanisms

- private fields: prevent access to representation
- interfaces: rep. Independence (List → ArrayList, Linked List)
  - No non-static fields allowed
  - Cannot have constructors

Data Abstraction
L6: Abstraction Functions & Rep Invariants

▪Rep Invariant

- Constraint that characterized whether an instance of an ADT is well-formed (representation point of view)
- RI: Object → Boolean
- Some properties of OM not in RI (eg. sharing/multiplicities)
- Some properties of RI not in OM (eg. primitives)

Data Abstraction
L6: Abstraction Functions & Rep Invariants

▪Inductive Reasoning

- Rep Invariant: makes modular reasoning possible
- Constructor creates an object that satisfies the invariant
- Producer preserves the invariant

- Mutator: if RI holds at beginning, must hold at end
- Observer does not modify, so RI should hold

Data Abstraction

L6: Abstraction Functions & Rep Invariants

▪Abstraction function: interprets representation

- Concrete objects: actual objects of the implementation
- Abstract objects: mathematical objects that correspond to the way the specification of the abstract type describes its values
- Function between concrete and abstract realms is the abstraction function
- May be partial
- Different representations have different abstraction functions

Data Abstraction

L6: Abstraction Functions & Rep Invariants

▪Benevolent side-effects: allow observers to mutate the rep as long as abstract value is preserved

▪RIs:

- Modular reasoning
- Helps catch errors

▪AF: specifies how representation of an ADT is interpreted as an abstract value

Data Abstraction

L7: Iteration Abstraction and Iterators

- Rep Exposure: have remove() throw UnsupportedOperationException
- Refer to Ch. 6 of text.

Object Models & Invariants

L8, Ch 12:1

▪Object model: description of collection of configurations

- Classification of objects
- Relationships between objects
- Subset (implements, extends)
- Relationships & labels
- Multiplicity: how many objects in one class can be related to a given object in another class
- Mutability: how states may change

Object Models & Invariants

▪Multiplicity symbols:

- * (>= 0)
- + (>= 1)
- ? (0 or 1)
- ! (exactly 1)

- Source → Target
    - End of the arrow: how many targets are associated with each source?
    - Beginning of arrow: how many sources can be mapped to the target?
- Instance diagrams

Object Models & Invariants

- Program object models
- Abstract & Concrete viewpoints
    - AF: can show how values of concrete are interpreted as abstract values
    - RI: object model is a type of RI—a constraint that holds during the lifetime of a program
    - Rep Exposure: ADT provides direct access to one of the objects within the rep invariant contour

Equality, Copying, Views

L9, Ch5:5-7

- Object Contract
    - equals()
    - hashCode()
- Equality Properties (Point and ColorPoint)
    - Reflexivity
    - Symmetry
    - Transitivity
- Hashing: if two objects are equal() → must have same hashCode()

Equality, Copying, Views

- Copying
    - Shallow: fields point to the same fields as old object
    - Deep
- Cloneable interface
- Element and Container equality
    - Liskov solution:
        - Equals – behaviorally equivalent
        - Similar – observationally equivalent

Equality, Copying, Views

- Rep exposure: contour includes element class (LinkedList example)
    - Mutating hash keys
- Views
    - Distinct objects that offer different kind of access to the underlying data structure
    - Both view and underlying structure modifiable

Dynamic Analysis

L10, L11, Chapter 10

- Executing program and observe it's behavior
- Dijkstra: "Testing can reveal the presence of errors but never their absence"

- Cannot depend on dynamic analysis alone – need good specifications and design

Dynamic Analysis

L10: Defensive Programming

- Guidelines
    - Inserting redundant checks – runtime assertions
    - As you are writing the code
    - Where?
        - At the start of a procedure (precondition)
        - End of a complicated procedure (postcondition)
        - When an operation may have an external effect

Dynamic Analysis

L10: Defensive Programming

- Catching Common Exceptions
    - NullPointerException
    - ArrayIndexOutOfBoundsException
    - ClassCastException
- Check the Rep Invariant
    - public void repCheck() throws (runtime expn)
- Assertion framework
    - public static void assert(boolean b, String loc)
    - Assert.assert(… , "MyClass.myMethod");

Dynamic Analysis

L10: Defensive Programming

- Assertions in Subclasses
- Responding to Failure
    - Fix: complicated, more bugs, if you know the cause → you could have avoided it anyway?
    - Execute special actions: depends on the system → hard to determine set of actions
    - Abort execution: depends on the program; compiler vs. word processor

Dynamic Analysis

L11: Testing

- Testing Considerations
    - Properties you want to test (problem domain, program knowledge)
    - Modules you want to test (critical, complex, most likely to malfunction)
    - How to generate test cases
    - How to check results
    - When you know you are done

Dynamic Analysis

L11: Regression Tests

- Tests suites that can be re-executed
- Test-first programming: construction of regression tests before application code is written (part of extreme programming)

Dynamic Analysis

L11: Criteria

- S(t, P(t)) = false; t is a failing test case
- C: Suite, Program, Spec → boolean
- C: Suite, Spec → boolean is specification-based criterion; black box
- C: Suite, Program → boolean is a program-based criterion; glass box

Dynamic Analysis

L11: Subdomains

- Subdomains: input space divisions
    - Determine if test suites are good enough
    - Drive testing in to regions where there are most likely bugs
- Revealing subdomain

Dynamic Analysis

L11: Subdomain Criteria

- Statement Coverage: every statement must be executed at least once
- Decision Coverage: every edge in the control flow graph must be executed
- Condition Coverage: boolean expressions to be evaluated to both true & false; MCDC
- Boundary testing: boundary cases for each conditional
- Specification based criteria: only in terms of subdomains
    - Empty set, non-empty & contains element, non-empty & not contains element

Dynamic Analysis

L11: Feasibility & Practicalities

- Criterion is feasible if it is possible to satisfy it.
- Use specification based criteria to guide development of test suite.
- Program based criteria to evaluate it. (Measure code coverage).

Design Patterns

L12, L13, L14, Chapter 15

- So far:
    - Encapsulation (data hiding)
    - Subclassing (inheritance)
    - Iteration
    - Exceptions

- Don't use design patterns prematurely
- Complex, decrease understandability

Design Patterns

L12: Creational Patterns

- Factories
    - Factory method: method that manufactures an object of a particular type
    - Factory object: object that encapsulates factory methods
    - Prototype: object can clone() itself, object is passed in to a method (instead of a factory object)

Design Patterns

L12: Creational Patterns

- Sharing
    - Singleton: only one object of a class exists
    - Interning: reuses object instead of creating new ones; correct for immutable objects only
    - Flyweight: (generalization of interning), can be used if most of the object is immutable
        - Intrinsic vs. extrinsic states
        - Only used if space is a critical bottleneck

Design Patterns

L13: Behavioral Patterns

- Multi-way Communication
    - Observer: maintain a list of observers (that follow a particular interface) to be notified when state changes; needs add and remove observer methods
    - Blackboard: (generalizes Observer pattern); multiple data sources and multiple viewers; asynchronous
        - Repository of messages which is readable and writable by all processes
        - Interoperability; well understood message format
    - Mediator: (intermediate between Observer and Blackboard); decouples information, but not control, synchronous

Design Patterns

L13: Traversing Composites

- Support many different operations
- Perform operations on subparts of a composite

- Interpreter: groups together operations for a particular type of object
- Procedural: groups together all code that implements a particular operation
- Visitor: depth-first traversal over a hierarchical structure; Nodes accept Visitors; Visitors visit Nodes

Design Patterns

L14: Structural Patterns

- Wrappers

	Pattern	Functionality	Interface
▪	Adaptor (interoperability)	Same	Different
▪	Decorator (extends)	Different	Same
▪	Proxy (controls or limits)	Same	Same

Design Patterns

L14: Structural Patterns

- Implementation of Wrappers
  - Subclassing
  - Delegation: stores an object in a field; preferred implementation for wrappers
- Composite
  - Allows client to manipulate a unit or collection of units in the same way

Subtyping

L15, Ch 7

- MDDs
- Substitution principle
  - Signatures
  - Methods
    - requires less/contravariance
    - guarantees more/covariance
  - Properties
- Java Subclasses vs. subtypes
- Interface
  - Guarantee behavior w/o sharing code
  - Multiple inheritance

Case Study: Java Collections API

- Type Hierarchy
  - Interfaces: Collection, Set, SortedSet, List
  - Skeletal implementations: AbstractCollection, AbstractSet, AbstractList, AbstractSequentialList
  - Concrete implementations: TreeSet, HashSet, ArrayList, LinkedList
- Parallel structure
- Interfaces vs. abstract classes

Case Study: Java Collections API

L16, Ch 13, Ch 14

- Optional Methods: throws UnsupportedOperationException

- Polymorphism
- Skeletal implementations ('template methods' and 'hook methods')
- Capacity, allocation, garbage collection
- Copies, Conversions, Wrappers
- Sorted Collections: Comparable vs. Comparator
- Views

Case Study: JUnit

L17

- MDD: fully connected
- Design Patterns
    - Template Method
    - Command
    - Composite
    - Observer
- TestSuite using Java Reflection

Case Study: Tagger

L18

- Design Aspects
    - Actions
    - Cross references
    - Property maps
    - Autonumbering
    - Style sheet view
    - Type-safe enums
- Quality needs
- Pattern density

Conceptual Object Models

L19, Ch 11-12

- Atom:
    - Indivisible
    - Immutable
    - Uninterpreted
- Set: collection of atoms
    - Domains: sets without supersets
    - Relation: relates atoms
    - Transpose: ~relation
    - Transitive closure: +relation
- Reflexive closure: *relation

Conceptual Object Models

- Ternary relations
- Indexed relation
- Examples

- Java Types: Object, Var, Type
- Meta Model: graphical object modeling notation
- Numbering: Tagger

Design Strategy

L20

- Development Process
  - Program analysis (OMs and operations)
  - Design (code OM, MDD, module specs)
  - Implementation
- Testing
  - Regression tests
  - Runtime assertions
  - Rep Invariants

Design Strategy

Design Properties

- Extensibility
  - OM sufficiency
  - Locality and decoupling
- Reliability
  - Careful modeling
  - Review, analysis, testing
- Efficiency
  - OM
  - Avoid bias
  - Optimization
  - Choice of Reps

Design Strategy

OM Transformations

- Introducing a generalization (subsets)
- Inserting a collection
- Reversing a relation
- Moving a relation
- Relation to table
- Adding redundant state
- Factoring out mutable relations
- Interpolating an interface
- Eliminating dynamic sets