ADT Design

Recognition
• C++ special member functions (6 of them):
  if one needs to be hand-crafted, likely all of them do

Comprehension
• Best practices for ADT design
• Entity vs. Value-based ADTs
• Rules for compiler-default special member functions (default constructor, destructor, copy constructor, copy assignment)

Application
• Operator overloading
• Const function arguments and member functions
• ADT design (entity vs. value-based design, immutable ADTs, hidden implementation)
• User-defined constructors, destructor, copy constructor, copy assignment
Recognition
- The benefits of modular design
- The benefits of separate compilation

Comprehension
- Principle of Separation of Concerns
- Principle of Information Hiding

Application
- Construct a program’s module dependency diagram
- Specify C++ header files
- Use interfaces/modules and forward declarations to reduce modular dependencies
- Create basic Makefiles that derive header dependencies
Interface Specifications

Recognition
• Specification as a contract.
• Specification as documentation of correct usage.
• The specificand set of a specification

Comprehension
• Specification considerations: specification restrictiveness / generality, comparing specifications, preconditions vs. exceptions

Application
• Specifying the interface of a C++ method or class.
• Specifying the interface of a derived class.
• Determining whether a C++ program satisfies a specification.
• Implementing a C++ program that satisfies a specification.
• Determining whether one specification is stronger than another.
Exceptions

Recognition
• Where and when to use (or not use) exceptions
• Motivation for RAII idiom

Comprehension
• How exception handling works in C++, including rules for find a matching handler and how stack unwinding works
• When to use exceptions vs. assertions, and why
• When to use unique_ptr vs. shared_ptr vs. weak_ptr

Application
• Throw and catch exception objects
• Nest exception classes within ADT classes
• How to use smart pointers
• How to use RAII to ensure resource initialization and cleanup
• noexcept
Rep Invariant / Abstraction Function

Recognition
• Motivation and uses of representation invariant
• Know where to perform runtime checking of a representation invariant
• Motivation and uses of abstraction functions

Comprehension
• Argue the pros and cons of including runtime checking of representation invariants in software that is released

Application
• Specify the representation invariant of an ADT implementation
• Use inductive reasoning to argue informally that an operation is correct
• Specify the abstraction function of an ADT implementation
UML

Recognition
• Recognize class diagrams: class, attribute, association, association name, rolename, association class.
• Recognize rules for designating data as objects or attributes.

Comprehension
• Distinguish between aggregation and composition.
• Understand program behaviour described as a sequence diagram.

Application
• Model a program's set of classes as a class diagram.
• Use multiplicities to constrain allowable instances of a class diagram.
• Express a program execution state as an object model.
Bad Smells / Refactoring

Recognition
• What is a “bad smell”, what is refactoring

Comprehension
• Considerations of whether to refactor (e.g., rep invariant, effort, XP rule of three)
• Argue different design alternatives
  – extract/merge methods or classes
  – primitive data members vs object members
  – multiple parameters vs. parameter objects
  – conditional code in methods vs. polymorphism
  – inheritance vs. composition/delegation
Design Principles

Recognition
• The goals underlying each design principle.
• That the principles are guidelines, not rules or laws.

Comprehension
• Detect that a design violates a principle.
• Argue (with justification) that one design is better than another.

Application
• Design an abstract base class.
• Determine whether one class is substitutable for another.
• Convert an inheritance relation into a composition relation.
• Determine whether a class encompasses more than one responsibility.
Design Patterns

Recognition
• Know what problem a pattern solves.
• Know a pattern’s UML model.

Comprehension
• Select appropriate design pattern to solve particular design problem.

Application
• Apply design patterns to modify given UML model.
• Implement (code fragments) of design patterns
  - Singleton
  - Template Method
  - Strategy
  - Adaptor
  - Observer
  - MVC
  - Composite
  - Iterator
  - Decorator
  - Factory Method
Templates, Generic Programming

Recognition
• Know the purpose and motivation of templates, functors, functor adapters, lambdas
• Know what “Duck Typing” is, how it applies to generics

Comprehension
• Select appropriate STL container type based on how container is used in program

Application
• Use basic STL containers vector, deque, list, set, map
• Write small program using STL algorithms, iterators
• Specialize STL algorithms with functions, functors, lambdas
• Specialize functions/functors with predefined functor adapters
• Write small function or class templates