Software Design

Software design and complexity:

- scale
- environment (I/O, Network)
- infrastructure (libraries, frameworks)
- evolution (design for change)
- correctness (testing, analysis tools, automation)

software qualities:

- sufficiency/func correctness
- robustness
- flexibility
- reusability
- efficiency
- scalability
- security

A simple process:

- Discuss the software that needs to be written
- Write some code
- Test the code to identify the defects
- Debug to find causes of defects
- Fix the defects
- If not done, return to first step

design tips:

- Think before coding
- Consider quality attributes (maintainability,extensibility, performance)
- Consider alternatives and make conscious design decisions

Preview: The design process:

- ObjectOriented Analysis
  - Understand the problem
  - Identify the key concepts and their relationships
  - Build a (visual) vocabulary
  - Create a domain model (aka conceptual model)
- ObjectOriented Design
  - Identify software classes and their relationships with class diagrams
  - Assign responsibilities (attributes, methods)
  - Explore behavior with interaction diagrams
  - Explore design alternatives
  - Create an object model (aka design model and design class diagram) and interaction models
- Implementation
  - Map designs to code, implementing classes and methods

Objects:

- A package of state (data) and behavior (actions)
- Can interact with objects by sending messages
  - perform an action (e.g., move)
  - request some information (e.g., getSize)
- Possible messages described through an interface

---

sources:

- `http://www.cs.cmu.edu/~charlie/courses/15-214/2015-fall/index.html#schedule`

Design for Change

Design principle for change: information hiding:

- expose little implementation as possible
- allows you to change hidden details later

Subtype Polymorphism:

- There may be multiple implementations of an interface
- Multiple implementations coexist in the same program
- May not even be distinguishable
- Every object has its own data and behavior

Interface:

- can implement start (defining required methods and classes)
- create class to implement interface

What to test:

- Functional correctness of a method (e.g.,computations, contracts)
- Functional correctness of a class (e.g., class invariants)
- Behavior of a class in a subsystem/multiple subsystems/the entire system
- Behavior when interacting with the world
  - Interacting with files, networks, sensors,
  - Erroneous states
  - Nondeterminism, Parallelism
  - Interaction with users
- Other qualities (performance, robustness, usability, security,)

Unit Tests (Junit good for JAVA):

- Unit tests for small units: functions, classes, subsystems
  - Smallest testable part of a system
  - Test parts before assembling them
  - Intended to catch local bugs
- Typically written by developers
- Many small, fastrunning, independent tests
- Little dependencies on other system parts or environment
- Insufficient but a good starting point
- extra benefits:
  - Documentation (executable specification)
  - Design mechanism (design for testability)

Test cases strategies:

- use specs
- representative cases
- invalid cases
- boundary cond
- think like attacker
- dificult cases

static methods:

- Static methods belong to a class
- global
- Direct dispatch, no subtype polymorphism
- Avoid unless really only a single implementation exists (e.g., Math.min)

Best practices:

- control access
  - fields not accessible from client code
  - methods only accessible in exposed interface
- contracts - agreement between provider and user
  - interface specification
  - functionality and correctness expectations
  - Performance expectations
- Visibility Modifiers
  - design principle for change: information hiding
  - expose little implementation as possible
  - allows you to change hidden details later

Notes:

- try to avoid setters
- Organize program functionality around kinds of abstract objects
  - For each object kind, offer a specific set of operations on the objects
  - Objects are otherwise opaque: Details of representation are hidden
  - Messages to the receiving object
- Distinguish interface from class
  - Interface: expectations
  - Class: delivery on expectations (the implementation)
  - Anonymous class: special Java construct to create objects without explicit classes: Point x = new Point() /* implementation */ ;
- Explicitly represent the taxonomy of object types
  - This is the type hierarchy (!= inheritance, more on that later): A CartesianPoint is a Point
- Design Patterns!!
  - Design Patterns by Gamma,Helm,Johnson,Vlissides

sources:

- `http://www.cs.cmu.edu/~charlie/courses/15-214/2015-fall/index.html#schedule`

Design for Reuse

Delegation:

- Delegation is simply when one object relies on another object for some subset of its functonality
  - Sorter is delegating functonality to some Comparator implementation
- Judicious delegation enables code reuse
  - Sorter can be reused with arbitrary sort orders
  - Comparators can be reused with arbitrary client code that needs to compare integers

Delegation and design:

- Small interfaces
- Classes to encapsulate algorithms
  - ex: the Comparator, the Strategy pattern

Inheritance:

- Typical roles:
  - An interface defines expectations/commitment for clients
  - An abstract class is a convenient hybrid between an interface and a full implementation
  - A subclass overrides a method definition to specialize its implementation

Benefits of Inheritance:

- Reuse of code
- Modeling flexibility
- A Java aside:
  - Each class can directly extend only one parent class
  - A class can implement multiple interfaces

Power of object oriented interfaces:

- Subtype polymorphism
  - Different kinds of objects can be treated uniformly by client code
- e.g., a list of all accounts
  - Each object behaves according to its type
- If you add new kind of account, client code does not change

Inheritance and subtyping:

- Inheritance is for code reuse
  - Write code once and only once
  - Superclass features implicitly available in subclass
- Subtyping is for polymorphism
  - Accessing objects the same way, but getting different behavior
  - Subtype is substitutable for supertype

Java details: final:

- A final field: prevents reassignment to the field after initialization
- A final method: prevents overriding the method
- A final class: prevents extending the class
  - e.g., public final class CheckingAccountImpl

Type Casting:

- Sometimes you wan a different type than what you have
  - ex: float pi=3.14; int indianapi=(int) pi;
- Useful if you have more specific subtype
- Advice: avoid downcasting types

  - Never downcast within superclass to a subclass

Behavioral Subtyping:

- Compiler enforced rules in Java:
  - Subtype (subclass, subinterface, object implementing interface) can add but not remove methods
  - Overriding method must return same type or subtype
  - Overriding method must accept same parameter types
  - Overriding method may not throw additional exceptions
  - Concrete class must implement all undefined interface methods and abstract methods
- A subclass must fulfill all contracts its superclass does:
  - same or strong invariants
  - same or stronger post
  - conds for all methods
  - same or weaker pre
  - conds for all methods

Parametric polymorphism via java generics:

- Parametric polumorphism is the ability to define a type generically to allow static type
- checking with out fully specifying types
- The java.util.Stack instead
  - A stack of some type T:

```
public class Stack<T> {
   public void push(T obj){...}
   public T pop() { ...}
}
```

- Improves typechecking, simplifies client code

Notes:

- Template method design pattern
- Decorator design pattern

Design for Robustness

Exceptions:

- Notify caller of exceptional circumstance (usually operational failure)
- Semantics
  - An exception propagates up the function
  - call stack until main() is reached (terminates program) or until the exception is caught
- Sources of exceptions:
  - Program throwing an exception
  - Exception thrown by the Java Virtual Machine

Java: Finally:

The finally block always runs after try/catch

Design choice: checked and unchecked:

- Unchecked exception: any subclass of RuntimeException
  - Error which is highly unlikely and/or unrecoverable
- check exception: any subclass of Exception that is not a subclass of RuntimeException
  - Error that every caller should be aware of and explicitly decide to handle or pass on
- Return values(- 1, false, null): If failure is common and expected possibility

Creating and throwing your own exceptions:

- Methods must declare any checked exceptions they might throw
- If your class extends java.lang.Throwable you can throw it:
  -
```
if (some__){
  throw new customException("Blah blah");
 }
```

Benefits of exceptions:

- High level summary of error and stack trace
  - Compare: core dumped in C
- Can't forget to handle common failure modes
  - Compare: using a flag or special return value
- Can optionally recover from failure
  - Compare: calling System.exit()
- Improve code structure
  -
  - Separate routine operations from error
  - handling
- Allow consisten clean
- up in both normal and exceptional operation

Guide for exceptions:

- Catch and handle all checked exceptions
  - Unless there is no good way
- Use runtime exceptions for programming error
- Other
  - Don't catch an exception without (at least somewhat) handling the error
  - When you throw an exception describe the error
  - if you re
  - throw an exception, always include the original exception as the clause

Modular Protection:

- Errors and bugs unavoidable but exceptions should not leak across modules (methods, classes) if possible
- Good modules handles exceptional conditions locally

  - Local input validation and local exception handling where possible
  - Explicit interfaces with clear pre/post conditions
- Explicitly documented and checked exceptions where exceptional conditions may propagate between modules
  - Information hiding -encapsulation of critical code (likely bugs, likely exceptions)

Problems when testing (sub -)systems:

- User interfaces and user interactions
  - Users click buttons, interpret output
  - Waiting/timing issues
- Test data vs. real data
- Testing against big infrastructure (database, web services,..)
- Testing with side effects (e.g. printing and mailing documents)
- Nondeterministic behavior
- Concurrency

Testing strategies in environments:

- Separate business logic and data representation from GUI for testing
- Test algorithms locally without large environment using stubs
- Advantage of stubs
  - Create deterministic response
  - Can reliably simulate spurious states (network error)
  - Can speed up test execution
  - Can simulate functionality not yet implemented
- Automate

Scaffolding:

- Catch bugs early: Before client code or service available
- Limit scope of debugging: Localize errors

- Improve coverage
  - System level tests may only cover 70
  - Simulate unusual error conditions
- Validate internal interface/API designs
  - Simulate clients in advance of their developement
  - Simulate services in advance of their development
- Capture developer intent (in absence of specification documentation)
  - A test suite formally captures elements of design intent
  - Developer documentation
- Improve low
- level design
  - Early attention to ability to test