cis20.2
design and implementation of software applications II
spring 2008
session # IV.2
design patterns and software testing

topics:

- design patterns
- software testing
- test scenarios

design patterns: overview

- one of the "hot topics" in the object-oriented software engineering community
- goal: to create a body of solutions to common problems in the area of software development
- this includes common vocabulary, strategies/algorithms and code for re-use
- origins: **Design Patterns: Elements of Reusable Object-Oriented Software**, by Gamma, Helm, Johnson and Vlissides
  also called the "Gang of Four" or **GoF**
- history:
  - initially used in Smalltalk to help novice programmers, as a "pattern language"
  - later used in C++ as an "idiom"
  - these ideas evolved into "design patterns"

design patterns: definition

- "A pattern is the abstraction from a concrete form which keeps recurring in specific non-arbitrary contexts." [Riehle and Zullinghoven, 1996]
- in software terms:
  "A pattern is a named nugget of instructive information that captures the essential structure and insight of a successful family of proven solutions to a recurring problem that arises within a certain context and system of forces." [Appleton, 2000]
- usually involve a modular architecture which is comprised of parts which together make a whole; the patterns come in when constructing the modules
- a pattern is a three-part rule containing: *context*, *problem* and *solution*
  or a pattern is a "thing" that happens in the world, the rule which tells how to create the thing and when to create it [Gabriel]

types of patterns

- generative patterns are used to create something
- non-generative patterns are used to describe something that recurs, but don't tell how to create it
- generative patterns show how to create something and illustrate characteristics of good (best) practice
- everything isn't a pattern!
- a pattern must have the three parts (context, problem, solution) and *it must recur!*
- good patterns:
  - solve a problem
  - demonstrate a proven concept
  - provide a non-obvious solution
  - describe a relationship between modules and system structures
  - contain a significant human component

- a pattern is not a "lesson learned"
- a pattern is a "best practice"
- we focus here on *software design patterns*, though many other types of patterns exist (like organizational patterns, analysis patterns, etc)

elements of a pattern

- "Alexandrian form"

```
IF   you find yourself in CONTEXT
     for example EXAMPLES,
     with PROBLEM,
     entailing FORCES
THEN for some REASONS,
     apply DESIGN FORM AND/OR RULE
     to construct SOLUTION
     leading to NEW CONTEXT and OTHER PATTERNS
```

- contain the following essential elements:
  - **name** — meaningful name for the pattern; can include a classification
  - **problem** — statement of the problem and its intent (goals and objectives it wishes to obtain)
  - **context** — preconditions under which problem and solution recur; i.e., applicability of the pattern

  - **forces** — description of relevant forces and constraints
  - **solution** — static relationships and dynamic rules describing how to realize the desired outcome
  - **examples** — sample applications of the pattern
  - **resulting context** — state of system after pattern has been applied
  - **rationale** — justifying explanation of steps/rules in the pattern
  - **related patterns** — relationships between this pattern and others in the same pattern language/system
  - **known uses** — known occurrences of the pattern and its aplicaiton within existing systems; may overlap with examples, but may be more complex since "examples" should be simple

forces

- generalize the kinds of criteria that software engineers use to justify designs and implementations
- e.g., in algorithms, the main force to be resolved is efficiency (time complexity)
- but patterns deal with the larger, harder-to-measure, and conflicting sets of goals and constraints encountered in the development of every artifact created
- examples:
  - Correctness
    - Completeness and correctness of solution
    - Static type safety, dynamic type safety
    - Multithreaded safety, liveness
    - Fault tolerance, transactionality
    - Security, robustness
  - Resources
    - Efficiency: performance, time complexity, number of messages sent, bandwidth requirements

    - Space utilization: number of memory cells, objects, threads, processes, communication channels, processors, ...
    - Incrementalness (on-demand-ness)
    - Policy dynamics: Fairness, equilibrium, stability
  - Structure
    - Modularity, encapsulation, coupling, independence
    - Extensibility: subclassibility, tunability, evolvability, maintainability
    - Reusability, openness, composibility, portability, embeddability
    - Context dependence
    - Interoperability
    - ... other "ilities" and "quality factors"
  - Construction
    - Understandability, minimality, simplicity, elegance.
    - Error-proneness of implementation
    - Coexistence with other software
    - Maintainability
    - Impact on/of development process
    - Impact on/of development team structure and dynamics

    - Impact on/of user participation
    - Impact on/of productivity, scheduling, cost
  - Usage
    - Ethics of use
    - Human factors: learnability, undoability, ...
    - Adaptability to a changing world
    - Aesthetics
    - Medical and environmental impact
    - Social, economic and political impact
    - ... other impact on human existence"

qualities of patterns

- encapsulation and abstraction
  - should encapsulate a well-defined problem
  - should abstract domain knowledge and experience
- openness and variability
  - should be open for extensions, in a wide variety of applications
- generativity and composability
  - applying one pattern should generate the context for another...
- equilibrium
  - should achieve a balance between forces and constraints

frameworks

- closely related to patterns
- design patterns can be used by frameworks but are more abstract than frameworks
- design patterns are smaller architecture elements than frameworks
- design patterns are more general than frameworks
- frameworks use "inverted flow of control" between its clients and itself
  "don't call us, we'll call you"...
  "leave the driving to us"...

patterns: cataloging and writing

- pattern catalog: collection of related patterns
- pattern system: set of related patterns with an underlying structure connecting the patterns together
- a pattern system is more structured than a pattern catalog, which is more like a list
- "pattern mining": looking for patterns in an existing system
- writing patterns is HARD
- patterns are not a silver bullet!

software testing

- "the process of executing a program with the intent of finding errors" [Myers 1979]
- physical systems tend to fail in a few, small (fixed) set of ways; software systems tend to fail in many (strange) ways
- physical systems tend to fail due to manufacturer errors; software systems tend to fail due to design errors
- physical systems tend to fail with age, usage; software systems can fail at any time...
- even small modules can be computationally complex
- exhaustive testing is not tractable: a program that adds two 32-bit integers would take hundreds of years to test exhaustively (2^{64} distinct test cases)
- fixing bugs may introduce new (often more subtle) bugs

why test software?

- to improve quality
  - bugs can be costly ($ and lives... remember examples of ariane and therac)
  - quality implies conforming to design requirements
- for verification and validation
  - funcionality (exterior quality)
  - engineering (interior quality)
  - adaptability (future quality)
- for reliability estimation

classifications

- by purpose:
  - correctness testing
  - performance testing
  - reliability testing
  - security testing
- by life-cycle phase:
  - requirements phase testing
  - design phase testing
  - program phase testing
  - evaluation test results
  - installation phase testing
  - acceptance testing
  - maintenance testing

- by scope
  - unit testing
  - component testing
  - integration testing
  - system testing

correctness testing

- minimum requirement of software testing
- need to be able to tell correct from incorrect behavior
- "white-box" and "black-box" methods
- black-box testing
  - also called "data driven" testing
  - test data are derived from functional requirements
  - testing involves providing inputs to a module and testing the resulting outputs; hence the name "black box"
  - only testing the functionality
- white-box testing
  - also called "glass box" testing
  - structure and flow of module being tested is visible
  - test cases are derived from program structure
  - some degree of exhaustion is desirable, e.g., executing every line of code at least once

- other methods: control flow testing, mutation testing, random testing
- control flow testing
  - also called/includes loop testing and data-flow testing
  - program flow is mapped in a flowchart
  - code is tested according to this flow
  - can be used to eliminate redundant or unused code
- mutation testing
  - original program code is perturbed and result is many new programs
  - all are tested—the most effective test cases/data are chosen based on which eliminate the most mutant programs
  - but this is (even more) exhaustive and intractable
- random testing
  - test caes are chosen randomly
  - cost effective, but won't necessarily hit all the important cases
- combinations of above yield best results in terms of completeness/effectiveness of testing, tractability and cost

performance testing

- e.g., make sure that software doesn't take infinite time to run or require infinite resources
- performance evaluation considers:
  - resource usage
    e.g., network bandwidth requirements, CPU cycles, disk space, disk access operations, memory usage
  - throughput
  - stimulus-response timing
  - queue lengths
- benchmarking frequently used here

reliability testing

- determination of failure-free system operation
- dependable software does not fail in unexpected or catastrophic ways
- "robustness" means the degree to which software can function when it receives unexpected inputs or within stressful conditions
- "stress testing" or "load testing" pushes the software/system beyond the typical limits to see if/when it will fail

security testing

- refers to testing for flaws that can be exploited by security holes
- particularly relevant to software that runs on the internet
- simulated security attacks help test this category

testing automation

- software testing tools help cut costs of manual testing
- typically involve the use of test scripts
- which are also costly to create
- so are used in cases where they are less costly to create and run than manual testing

when to stop?

- never! (hehe)
- there are always two more bugs—the one you know about and the one you don't...
- trade-offs between budget, time, quality
- choices must be made...
- alternatives?
  - buggy software/systems?
  - some kind(s) of testing is necessary
  - "proofs" using formal methods
  - do you think the use of design patterns may help reduce testing?

software testing: conclusions

- software testing is an art
- testing is more than just debugging
- testing is expensive
- complete testing is infeasible
- testing may not be the most effective way to improve software quality

test scenarios

- a *scenario* is a "hypothetical story used to help a person think through a complex problem or system"
- the idea of "scenario planning" gained popularity in the 2nd half of the 1900's
- can be useful for illustrating a point, as well as testing a system
- ideal scenario has five characteristics:
  - a story that is
  - motivating
  - credible
  - complex
  - easy to evaluate
- storytelling is an art, but a good scenario is a good story!
- risks and problems:
  - other approaches are better for testing early code
  - not designed for coverage of an entire system

  - often heavily documented and used repeatedly, but often expose design errors rather than implementation (coding) errors
- scenario testing vs requirements analysis
  - requirements analysis is used to create agreement about a system to be built; scenario testing is used to predict problems with a system
  - scenario testing only needs to point out problems, not solve them, reach conclusions or make recommendations about what to do about them
  - scenario testing does not make design trade-offs, but can expose consequences of trade-offs
  - scenario testing does not need to be exhaustive, only useful

creating test scenarios

- write life histories for objects in the system
- list possible users; analyze their interests and objectives
- list "system events" and "special events"
- list benefits and create end-to-end tasks to check them
- interview users about famous challenges and failures of the old system
- work alongside users to see how they work and what they do
- read aobut what systems like this are supposed to do
- study complaints about he predecessor to this system and/or its competitors
- create a mock business; treat it as real and process its data

(from Kaner article)