Final Exam Review (extended)

15-413: Introduction to Software Engineering

Jonathan Aldrich

Hoare Logic

\{ N \geq 0, M \geq 0 \}
\begin{align*}
p &:= 1 \\
i &:= N \\
\text{while } (i > 0) \\
&\quad p := p \cdot M; \\
&\quad i := i - 1; \\
\{ & p = M^i \}
\end{align*}

- Loop invariant?
  - \( p = M^{N-i} \) \& \( i \geq 0 \)
- Variant function?
  - \( i \)
- Loop verification condition?
  - INV \& \( i > 0 \) =>
    - \( p \cdot M = M^{N-(i-1)} \) \& \( (i-1) \geq 0 \)
    - \( p = M^{N-(i+1)} \) \& \( i > 0 \)
    - \( p = M^{N-i} \) \& \( i > 0 \)

8 December 2005
Translation of Temporal Logic

- \( p = \text{"p holds in the current state"} \)
- \( A f = \text{"for all paths from the current state"} f \)
- \( E f = \text{"there exists some path from the current state such that"} f \)
- \( X f = \text{"in the next state on the path"} f \)
- \( G f = \text{"in all states along the path"} f \)
- \( f U g = \text{"along the path there will eventually be a state where"} g \)
  \quad \text{"and for every state before that"} f \)
- \( F g = \text{"along the path there will eventually be a state where"} g \)

- Examples
  - \( AG p = \text{"for all paths from the current state, in all states along the path, p holds"} \)
  - \( EF p = \text{"there exists a path from the current state such that along the path there will eventually be a state where p holds"} \)

Temporal Logic

- Translate these statements into temporal logic
  - No matter what happens, the dike will never overflow at any time
    - \( AG ~\text{overflow} \)
  - No matter how I use my computer, at any point, there is always something I can do that will eventually shut down
    - \( AG EF \text{ shutdown} \)
  - No matter how I use my computer, at any point, if I press the reset button then the computer will restart immediately afterwards
    - \( AG (\text{reset} \Rightarrow AX \text{ restart}) \)
Behavioral: Visitor

- **Applicability**
  - Structure with many classes
  - Want to perform operations that depend on classes
  - Set of classes is stable
  - Want to define new operations

- **Consequences**
  - Easy to add new operations
  - Groups related behavior in Visitor
  - Adding new elements is hard
  - Visitor can store state
  - Elements must expose interface

Behavioral: Observer

- **Applicability**
  - When an abstraction has two aspects, one dependent on the other, and you want to reuse each
  - When change to one object requires changing others, and you don’t know how many objects need to be changed
  - When an object should be able to notify others without knowing who they are

- **Consequences**
  - Loose coupling between subject and observer, enhancing reuse
  - Support for broadcast communication
  - Notification can lead to further updates, causing a cascade effect
Behavioral: Template Method

- Applicability
  - When an algorithm consists of varying and invariant parts that must be customized
  - When common behavior in subclasses should be factored and localized to avoid code duplication
  - To control subclass extensions to specific operations

- Consequences
  - Code reuse
  - Inverted “Hollywood” control: don’t call us, we’ll call you
  - Ensures the invariant parts of the algorithm are not changed by subclasses

8 December 2005

Creative: Abstract factory

- Applicability
  - System should be independent of product creation
  - Want to configure with multiple families of products
  - Want to ensure that a product family is used together

- Consequences
  - Isolates concrete classes
  - Makes it easy to change product families
  - Helps ensure consistent use of family
  - Hard to support new kinds of products

8 December 2005
Creational: Factory Method

- Applicability
  - A class can’t anticipate the class of objects it must create
  - A class wants its subclasses to specify the objects it creates

- Consequences
  - Provides hooks for subclasses to customize creation behavior
  - Connects parallel class hierarchies

Creational: Singleton

- Applicability
  - There must be exactly one instance of a class
  - When it must be accessible to clients from a well-known place
  - When the sole instance should be extensible by subclassing, with unmodified clients using the subclass

- Consequences
  - Controlled access to sole instance
  - Reduced name space (vs. global variables)
  - Can be refined in subclass or changed to allow multiple instances
  - More flexible than class operations
    - Can change later if you need to

- Implementation
  - Constructor is protected
  - Instance variable is private
  - Public operation returns singleton
  - May lazily create singleton

- Subclassing
  - Instance() method can look up subclass to create in environment

8 December 2005
Structural: Proxy

- **Applicability**
  - Whenever you need a more sophisticated object reference than a simple pointer
    - Local representative for a remote object
    - Create or load expensive object on demand
    - Control access to an object
    - Reference count an object

- **Consequences**
  - Introduces a level of indirection
    - Hides distribution from client
    - Hides optimizations from client
    - Adds housekeeping tasks

---

What is an architecture?

- A software architecture is the **structure** or **structures** of a system, which comprise elements, their externally-visible **properties**, and the **relationships** among them

- But what kinds of structure?
  - **modules**: showing composition/decomposition
  - **runtime**: components at runtime
  - **allocation**: how software is deployed
  - ...

- Each is the basis of an **Architectural View**
Component-and-Connector (C&C) View

- Decomposition of system into **components**…
  - **Components**: principal units of run-time computation and data stores
    - Examples: client, server
  - Typically hierarchical
- And **connectors**…
  - **Connectors**: define an abstraction of the interactions between components
    - Examples: procedure call, pipe, event announce
- Using architectural **styles**…
  - Guide composition of components and connectors
- And **constraints** (or invariants)

---

What Is Institutionalization?

- Institutionalization involves implementing practices that
  - Ensure processes can be communicated about (they are defined, documented, understood)
  - Ensure the processes are effective, repeatable and lasting
  - Provide needed infrastructure support
  - Enable organizational learning to improve the process
Black-box Testing

- Verify each piece of functionality of the system  
  - Black-box: don’t look at the code
- Systematic testing  
  - Test each use case  
  - Test combinations of functionality (bold + italic + font + size)  
    - Generally have to sample  
  - Test incorrect user input  
  - Test each “equivalence class” (similar input/output)  
  - Test uncommon cases  
    - Generating all error messages  
    - Using uncommon functionality  
  - Test borderline cases  
    - Edges of ranges, overflow inputs, array of size 0 or 1

8 December 2005
White-box Testing

- Look at the code (white-box) and try to systematically cause it to fail
- Coverage criteria: a way to be systematic
  - Function coverage
    - Execute each function
  - Statement coverage
    - Most common
  - Edge coverage
    - Take both sides of each branch
  - Path coverage
    - Note: infinite number of paths!
    - Typical compromise: 0-1-many loop iterations
  - Condition coverage
    - Choose a set of predicates
    - Cover each statement with each combination of predicates
  - Exercise data structures
    - Each conceptual state or sequence of states
- Typically cannot reach 100% coverage
  - Especially true of paths, conditions