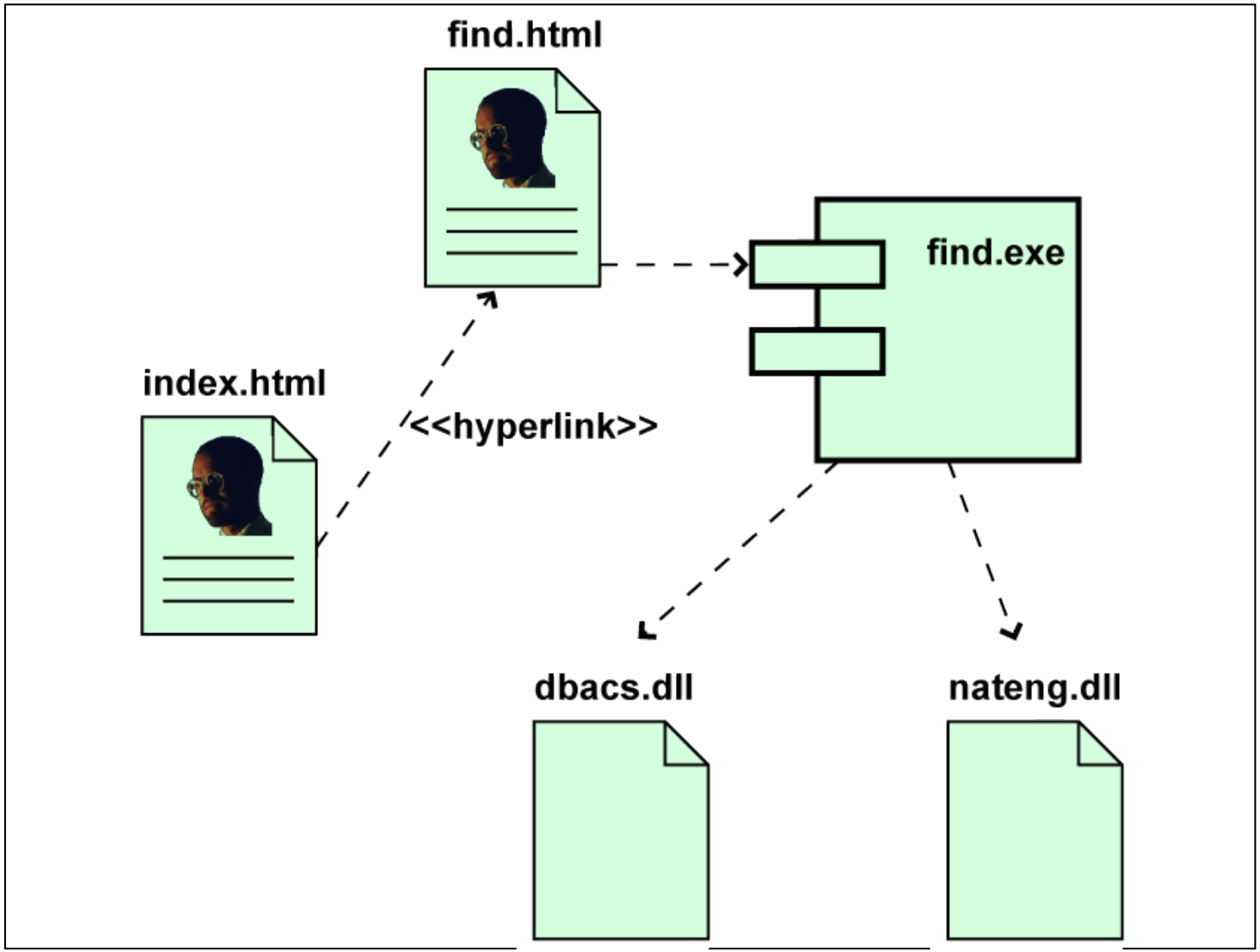
Software Design
Models, Tools & Processes

Lecture 6: Transition Phase

Cecilia Mascolo

UML Component diagram



Component documentation

- Your own classes should be documented the same way library classes are.
- Other people should be able to use your class without reading the implementation.
- Make your class a 'library class'!

Elements of documentation

Documentation for a class should include:

- the class name
- a comment describing the overall purpose and characteristics of the class
- a version number
- the authors' names
- documentation for each constructor and each method

Elements of documentation

The documentation for each constructor and method should include:

- the name of the method
- the return type
- the parameter names and types
- a description of the purpose and function of the method
- a description of each parameter
- a description of the value returned

Javadoc (should know)

- Part of the Java standard
- Each class and method can include special keywords in a comment explaining the interface to that class
- During javadoc compilation, the keyword information gets converted to a consistent reference format using HTML
- The documentation for standard Java libraries is all generated using javadoc

javadoc example

Class comment:

```
/**
 * The Responder class represents a response
 * generator object. It is used to generate an
 * automatic response.
 *
 * @author     Michael Kölling and David J. Barnes
 * @version    1.0  (1.Feb.2002)
 */
```

javadoc example

Method comment:

```
/**
 * Read a line of text from standard input (the text
 * terminal), and return it as a set of words.
 *
 * @param  prompt  A prompt to print to screen.
 * @return  A set of Strings, where each String is
 *          one of the words typed by the user
 */
public HashSet getInput(String prompt)
{
   ...
}
```

What is the goal of testing?

- A) To define the end point of the software development process as a managed objective?
- B) To prove that the programmers have implemented the specification correctly?
- C) To demonstrate that the resulting software product meets defined quality standards?
- D) To ensure that the software product won't fail, with results that might be damaging?

Testing and quality

- Wikipedia
  - "Software testing is the process used to assess the ***quality*** of computer software. It is an empirical technical investigation conducted to provide stakeholders with information about the ***quality*** of the product or service under test, with respect to the context in which it is intended to operate."
- Edsger Dijkstra
  - "Program testing can be used to show the ***presence*** of bugs, but ***never*** to show their absence"

Remember design as learning?

- Design is the process of learning about a problem and describing a solution
  - at first with many gaps …
  - eventually in sufficient detail to build it.
- We describe both the problem and the solution in a series of *design models*.
- Testing those models in various ways helps us gather more knowledge.
- Source code is simply the most detailed model used in software development.

Learning through testing

A bug is a system's way of telling you that you don't know something (P. Armour)

- Testing searches for the ***presence*** of bugs.
- Later: 'debugging' searches for the ***cause*** of bugs, once testing has found that a bug exists.
  - The manifestation of a bug as observable behaviour of the system may well occur at some 'distance' from its cause.

Testing principles

- Look for violations of the interface contract.
  - Aim is to find bugs, ***not*** to prove that unit works as expected from its interface contract
  - Use positive tests (expected to pass) in the hope that they ***won't*** pass
  - Use negative tests (expected to fail) in the hope that they ***don't*** fail
- Try to test *boundaries* of the contract
  - e.g. zero, one, overflow, search empty collection, add to a full collection.

Unit testing priorities

- Concentrate on modules most likely to contain errors:
  - Particularly complex
  - Novel things you have not done before
  - Areas known to be error-prone
- Some habits in unit test ordering
  - Start with small modules
  - Try to get input/output modules working early
    - Allows you to work with real test data
  - Add new ones gradually
  - You probably want to test critical modules early
    - For peace of mind, not because you expect errors

How to do it: testing strategies

- Manual techniques
  - Software inspections and code walkthrough
- Black box testing
  - Based on specified unit interfaces, not internal structure, for test case design
- White box testing
  - Based on knowing the internal structure
- Stress testing
  - At what point will it fail?
- ‘Random’ (unexpected) testing
  - Remember the goal: most errors in least time

Pioneers – Michael Fagan

- Software Inspections
  - 1976, IBM
- Approach to design checking, including planning, control and checkpoints.
- Try to find errors in design and code by systematic *walkthrough*
- Work in teams including designer, coder, tester and moderator.

Software inspections

- A low-tech approach, relatively underused, but more powerful than appreciated.
- Read the source code in execution order, acting out the role of the computer
  - High-level (step) or low-level (step-into) views.
- An expert tries to find common errors
  - Array bound errors
  - Off-by-one errors
  - File I/O (and threaded network I/O)
  - Default values
  - Comparisons
  - Reference versus copy

Inspection by yourself

- Get away from the computer and ‘run’ a program by hand
- Note the current object state on paper
- Try to find opportunities for incorrect behaviour by creating incorrect state.
- Tabulate values of fields, including invalid combinations.
- Identify the state changes that result from each method call.

Black box testing

- Based on interface specifications for whole system or individual modules
- Analyse input ranges to determine test cases
- Boundary values
  - Upper and lower bounds for each value
  - Invalid inputs outside each bound
- Equivalence classes
  - Identify data ranges and combinations that are ‘known’ to be equivalent
  - Ensure each equivalence class is sampled, but not over-represented in test case data

White box testing

- Design test cases by looking at internal structure, including all possible bug sources
  - Test each independent path at least once
  - Prepare test case data to force paths
  - Focus on error-prone situations (e.g. empty list)
  - The goal is to find as many errors as you can
- Control structure tests:
  - conditions – take each possible branch
  - data flow – confirm path through parameters
  - loops – executed zero, one, many times
  - exceptions – ensure that they occur

Stress testing

- The aim of stress testing is to find out *at what point* the system will fail
  - You really *do* want to know what that point is.
  - You *have to keep going* until the system fails.
  - If it hasn't failed, you haven't done stress testing.
- Consider both volume and speed
- Note difference from *performance testing*, which aims to confirm that the system will perform as specified.
  - Used as a contractual demonstration
  - It's not an efficient way of finding errors

Random testing

- There are far more combinations of state and data than can be tested exhaustively
- Systematic test case design helps explore the range of possible system behaviour
  - But remember the goal is to make the system fail, not to identify the many ways it works correctly.
- Experienced testers have an instinct for the kinds of things that make a system fail
  - Usually by thinking about the system in ways the programmer did not expect.
  - Sometimes, just doing things at random can be an effective strategy for this.

Regression testing

- ‘Regression’ is when you go backwards, or things get worse
  - Regression in software usually results from re-introducing faults that were previously fixed.
  - Each bug fix has around 20% probability of reintroducing some other old problem.
  - Refactoring can reintroduce design faults
- So regression testing is designed to ensure that a new version gives the same answers as the old version did

Regression testing

- Use a large database of test cases
- Include all bugs reported by customers:
  - customers are much more upset by failure of an already familiar feature than of a new one
  - reliability of software is relative to a set of inputs, so better test inputs that users actually generate!
- Regression testing is boring and unpopular
  - test automation tools reduce mundane repetition
  - perhaps biggest single advance in tools for software engineering of packaged software

Test automation

- Thorough testing (especially regression testing) is time consuming and repetitive.
- Write special classes to test interfaces of other classes automatically
  - "test rig" or "test harness"
  - "test stubs" substitute for unwritten code, or simulate real-time / complex data
- Use standard tools to exercise external API, commands, or UI (e.g. mouse replay)
  - In commercial contexts, often driven from build and configuration tools.

Unit testing

- Each unit of an application may be tested.
  - Method, class, interface, package
- Can (should) be done *during* development.
  - Finding and fixing early lowers development costs (e.g. programmer time).
  - Build up a test suite of necessary harnesses, stubs and data files
- JUnit is often used to manage and run tests
  - you will use this to check your practical exercises
  - www.junit.org

Other system tests

- Security testing
  - automated probes, or
  - a favour from your Russian friends
- Efficiency testing
  - test expected increase with data size
  - use code profilers to find hot spots
- Usability testing
  - essential to product success
  - will be covered in further detail in Part II

Testing efficiency: optimisation

- Worst error is using wrong algorithm
  - e.g. lab graduate reduced 48 hours to 2 minutes
  - Try different size data sets – does execution time vary as N, 2N, N^2, N^3, N^4, k^N ...?
- If this is the best algorithm, and you know it scales in a way appropriate to your data, but still goes too slow for some reason, ask:
  - How often will this program / feature be run?
  - Hardware gets faster quickly
  - Optimisation may be a waste of ***your*** time

Testing efficiency: optimisation

- When optimisation is required
  - First: check out compiler optimisation flags
  - For *some parts* of extreme applications
    - Use code profiler to find hotspots/bottlenecks
    - Most likely cause: overuse of some library/OS function
  - When pushing hardware envelope
    - Cache or pre-calculate critical data
    - Recode a function in C or assembler
    - Use special fast math tricks & bit-twiddling
    - Unroll loops (but compilers should do this)
- But if this is an interactive system …
  - … how fast will the user be?

User interface efficiency

- Usability testing can measure speed of use
  - How *long* did Fred take to order a book from Amazon?
  - How many *errors* did he make?
- But every observation is different.
  - Fred might be faster (or slower) next time
  - Jane might be consistently faster
- So we compare averages:
  - over a number of trials
  - over a range of people (experimental subjects)
- Results usually have a normal distribution

Summary

- We have described the main principles of testing and many different approaches to testing.