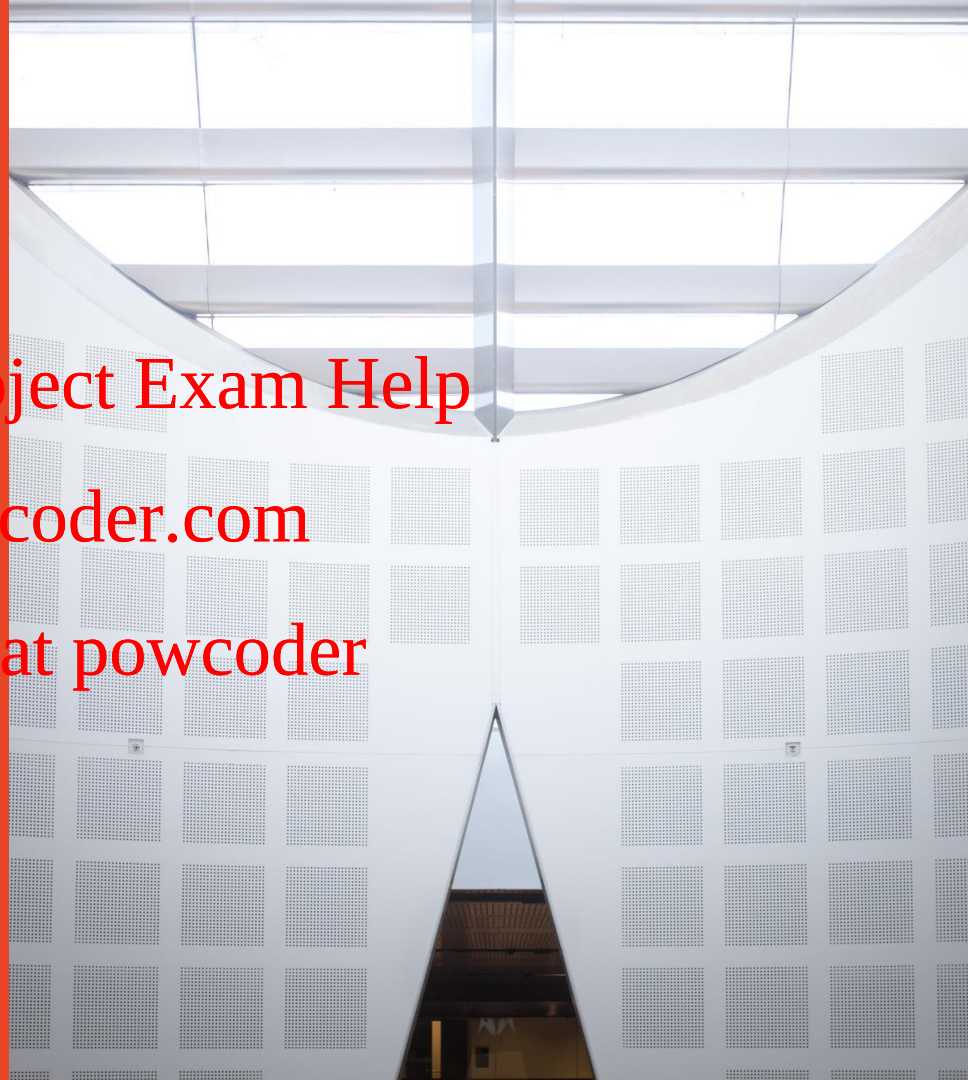
Software Design and Construction 2
SOFT3202 / COMP9202

Advanced Testing Techniques (2)

Dr. Basem Suleiman

School of Information Technologies

THE UNIVERSITY OF SYDNEY



Copyright Warning

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the University of Sydney pursuant to Part VB of the Copyright Act 1968 (**the Act**).

The material in this communication may be subject to copyright under the Act. Any further copying or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice.

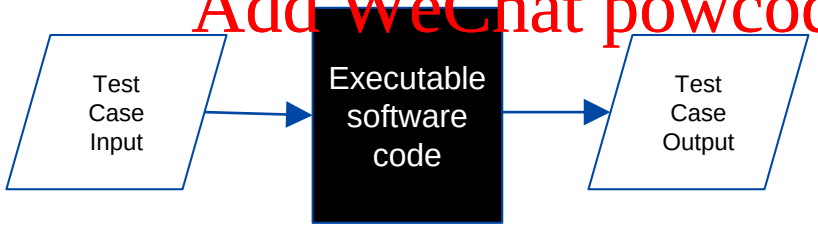
Agenda

- Testing Techniques
  - Black-box, white-box and gray-box testing
  - Fuzzing
- Code/Test Coverage
- System Testing
- User Acceptance Testing

Advanced Testing Types

Black-box Testing

- No programming and software knowledge
- Carried by software testers
- May lead to poor coverage
- Can be applied unit, integration, system and acceptance testing

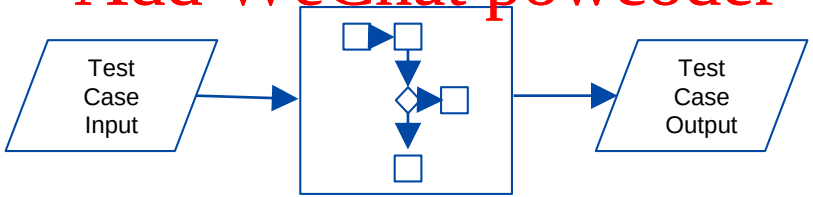


Black Box Testing – Example

- Test planned without knowledge of the code
- Based only on specification or design
- E.g., given a function that computes $sign\ (x+y)$

White-box Testing

- Verifies the internal structures of a source code (*structural testing*)
- Test cases are designed based on code understanding and programming skills (by software developers)
- Unit, integration and system testing
- Can help detecting many errors, but not unimplemented/missing requirements



Grey-box Testing

- Tests are designed based on the internal structures and algorithms
  - Test cases from high-level and detailed application documentation
  - To design more informed test cases (better test coverage)
- Tests are executed from outside
  - Black-box testing

Black, White or Grey?

```
1 class MyGregorianCalendar {
2     ....
3         public static int getNumDaysInMonth(int month, int year){
4         ....
5     }
6 }
```

Black-box Testing – Test Cases

Equivalence Class	Value for month	Value for year
Months with 31 days, non-leap years	7 (July)	1901
Months with 31 days, leap years	7 (July)	1904
Months with 30 days, non-leap years	6 (June)	1901
Months with 30 days, leap year	6 (June)	1904
Months with 28 or 29 days, non-leap year	2 February	1901
Months with 28 or 29 days, leap year	2 February	1904

Equivalence Class	Value for month	Value for year
Leap years divisible by 400	2 (February)	2000
Non-leap years divisible by 100	2 (February)	1900
Non-positive invalid month	0	1291
Positive invalid months	13	1315

Exercise – White-box Testing

```
1  ....
2  class MyGregorianCalendar {
3      public static boolean isLeapYear(int year) {
4          boolean leap;
5          if ((year%4) == 0){
6              leap = true;
7          } else {
8              leap = false;
9          }
10         return leap;
11     }
12     public static int getNumDaysInMonth(int month, int year)
13         throws MonthOutOfBounds, YearOutOfBounds {
14     int numDays;
15     if (year < 1) {
16         throw new YearOutOfBounds(year);
17         }
18     if (month == 1 || month == 3 || month == 5 || month == 7 ||
19         month == 10 || month == 12) {
20          numDays = 32;
21     } else if (month == 4 || month == 6 || month == 9 || month == 11) {
22         numDays = 30;
23     } else if (month == 2) {
24     if (isLeapYear(year)) {
25         numDays = 29;
26     } else {
27         numDays = 28;
28     }
29     } else {
30         throw new MonthOutOfBounds(month);
31         }
32         return numDays;
33     }
34 }
```

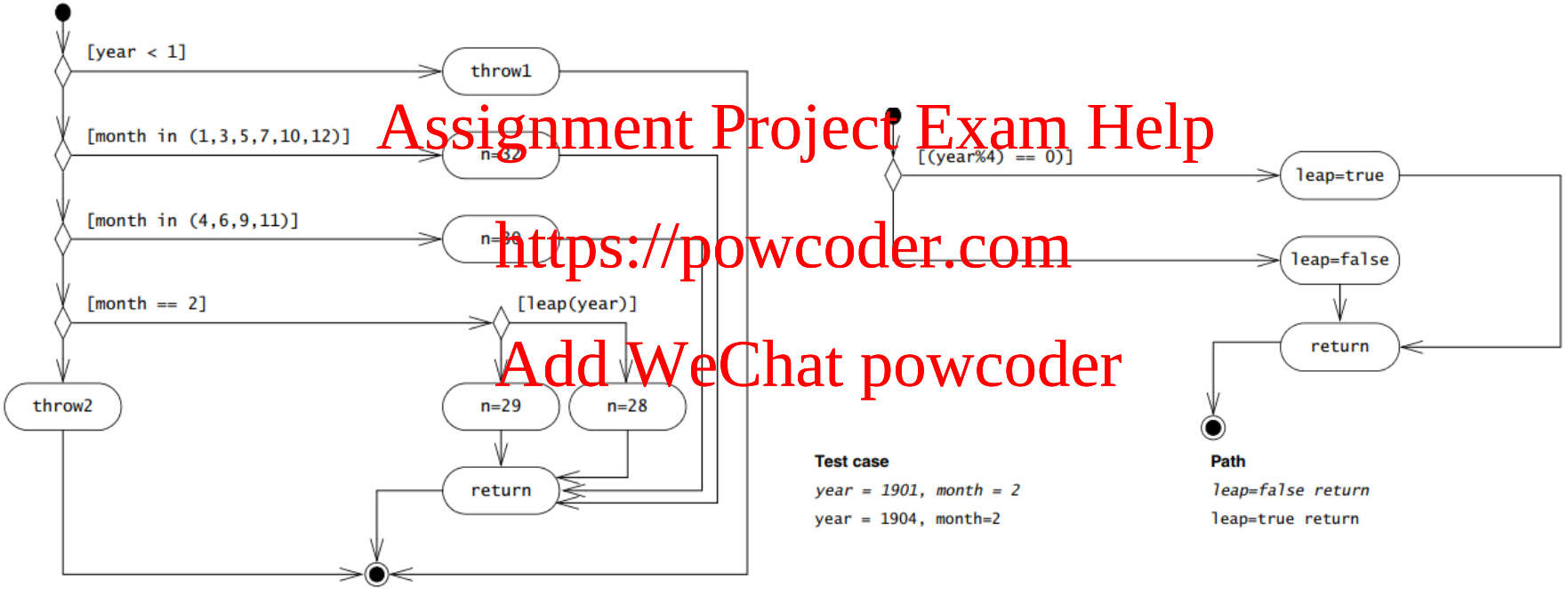
Based on the provided code, design test cases for the method *getNumDaysInMonth()*

Hint: Consider various paths in the code

White-box Testing

- Exercising all possible paths through the code at least once, most faults will trigger failures
- Paths identification requires understanding the source code and the data structure
- Diagrams (e.g., flow graph, activity diagram) may be utilized to visualize all paths
- Test cases that each edge in the activity diagram is traversed at least once
  - Condition – select inputs both the true and false branches

White-box Testing – Visualizing Paths



Test case	Path
year = 1901, month = 2	*leap=false return*
year = 1904, month=2	leap=true return

White-box Testing – Test Cases

Test case	Path
(year = 0, month = 1)	{throw1}
(year = 1901, month = 1)	{n=32 return}
(year = 1901, month = 2) (year = 1904, month = 2)	{n=28 return}
(year = 1901, month = 4)	{n=30 return}
(year = 1901, month = 0)	{throw2}

White-box Testing – Compare

Equivalence Class	Value for month	Value for year
Months with 31 days, non-leap years	7 (July)	1901
Months with 31 days, leap years	7 (July)	1904
Months with 30 days, non-leap years	6 (June)	1901
Months with 30 days, leap year	6 (June)	1904
Months with 28 or 29 days, non-leap year	2 February	1901
Months with 28 or 29 days, leap year	2 February	1904

Equivalence Class	Value for month	Value for year
Leap years divisible by 400	2 (February)	2000
Non-leap years divisible by 100	2 (February)	1900
Non-positive invalid month	0	1291
Positive invalid months	13	1315

Test case	Path
(year = 0, month = 1)	{throw1}
(year = 1901, month = 1)	{n=32 return}
(year = 1901, month = 2) (year = 1904, month = 2)	{n=28 return}
(year = 1901, month = 4)	{n=30 return}
(year = 1901, month = 0)	{throw2}

White-box Testing – Compare

- Both extensively test cases of February calculations
- No test cases for years divisible by 100 – path testing with *isLeapYear()*
  - Path testing exercise a path in the program (e.g. numDays=32)
- Omission and missing requirements cannot be detected in path testing
  - Failure to handle the non-leap year 1900

Code (Test) Coverage

Code (Test) Coverage

- A metric that measures the extent to which a source code has been executed by a set of tests
- White-box testing
- Usually measured as percentage, e.g., 70% coverage
- Different criteria to measure coverage
  - E.g., method, statement, loop

Coverage Criteria

Coverage Criteria	Description
Method	How many of the methods are called, during the tests
Call	How many of the potential method calls are executed, during the tests
Statement	How many statements are exercised, during the tests
Branch (decision)	How many of the possible outcomes of the branches have occurred during the tests
Condition	Has each separate condition within each branch been evaluated to both true and false
Condition/decision coverage	Requires both decision and condition coverage be satisfied
Loop	Each loop executed zero times, once and more than once
Data flow coverage	Each variable definition and its usage executed

Coverage Criteria – Method

- Examine the code. Given the following test cases, calculate the method coverage
  - month = 7, year = 1901
  - month – 0, year = 2002
  - month = 2, year = 1901
  - month = 2, year = 1904
- Discuss

```
....
class MyGregorianCalendar {
    public static boolean isLeapYear(int year) {
        boolean leap;
        if ((year%4) == 0){
            leap = true;
        } else {
            leap = false;
        }
        return leap;
    }
    public static int getNumDaysInMonth(int month, int year)
        throws MonthOutOfBounds, YearOutOfBounds {
    int numDays;
    if (year < 1000)
        throw new YearOutOfBounds(year);
        }
    if (month == 1 || month == 3 || month == 5 || month == 7 ||
        month == 10 || month == 12) {
        numDays = 31;
    } else if (month == 4 || month == 6 || month == 9 || month == 11) {
        numDays = 30;
    } else if (month == 2) {
    if (isLeapYear(year)) {
        numDays = 29;
    } else {
        numDays = 28;
    }
    } else {
        throw new MonthOutOfBounds(month);
        }
        return numDays;
    }
}
```

Coverage Criteria – Statement

- Examine the code snippet. Calculate the statement coverage after executing each of the following test cases:
  - year = 1901
  - year = 1904
- Discuss

```
1  //....
2  class MyGregorianCalendar {
3      public static boolean isLeapYear(int year) {
4          boolean leap;
5          if ((year%4) == 0){
6              leap = true;
7          else
8              leap = false;
9          }
10         return leap;
11     }
12     //....
```

Coverage Criteria – Branch

- Examine the code snippet. Identify test cases that will ensure that each branch is exercised at least once
- Discuss

```
1  //....
2  class MyGregorianCalendar {
3      public static boolean isLeapYear(int year) {
4          boolean leap;
5          if ((year%4) == 0){
6              leap = true;
7          } else {
8              leap = false;
9          }
10         return leap;
11     }
12     //....
```

Coverage Criteria – Condition

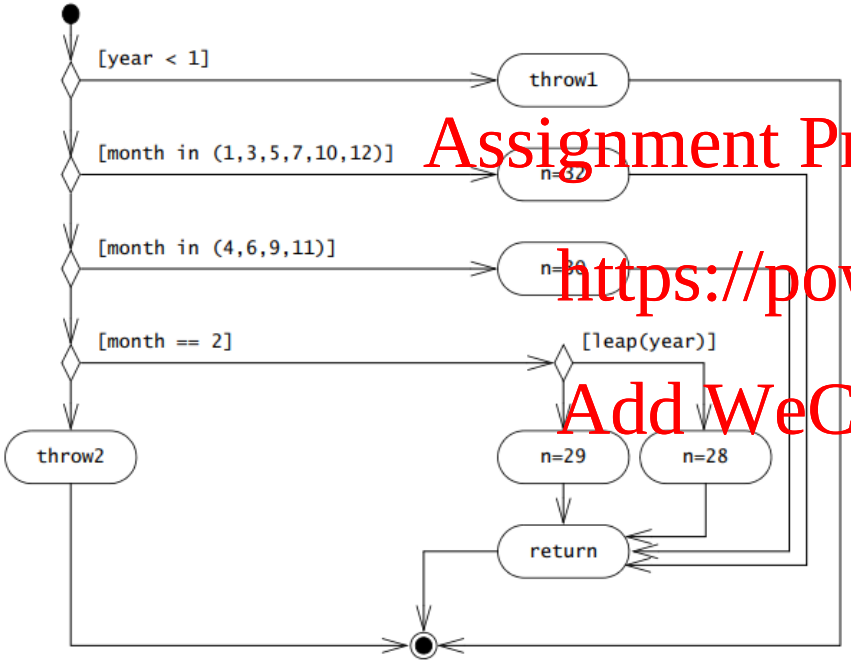
- Examine the code snippet. Calculate the condition coverage after executing each of the following test cases:
  - year = 2000
  - year = 2018
- Discuss

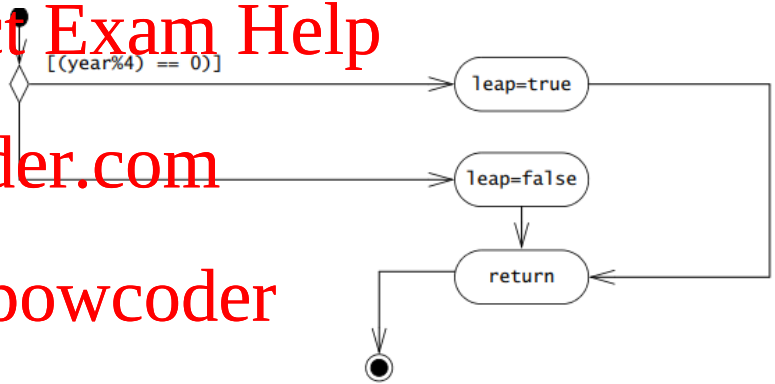
```
1  //....
2  class MyGregorianCalendar {
3      public static boolean isLeapYear(int year) {
4          boolean leap;
5          if ((year%4) == 0){
6              leap = true;
7          } else {
8              leap = false;
9          }
10         return leap;
11     }
12     //....
```

Other Coverage Criteria

- Path coverage
  - Has every possible route through a given part of the code been executed
- Entry/exit coverage
  - Has every possible call and return of the function been executed
- State coverage
  - Has every state in a FSM (finite-state-machine) been reached and explored
- Other…

Visualizing Paths – Recall





Test case

year = 1901, month = 2

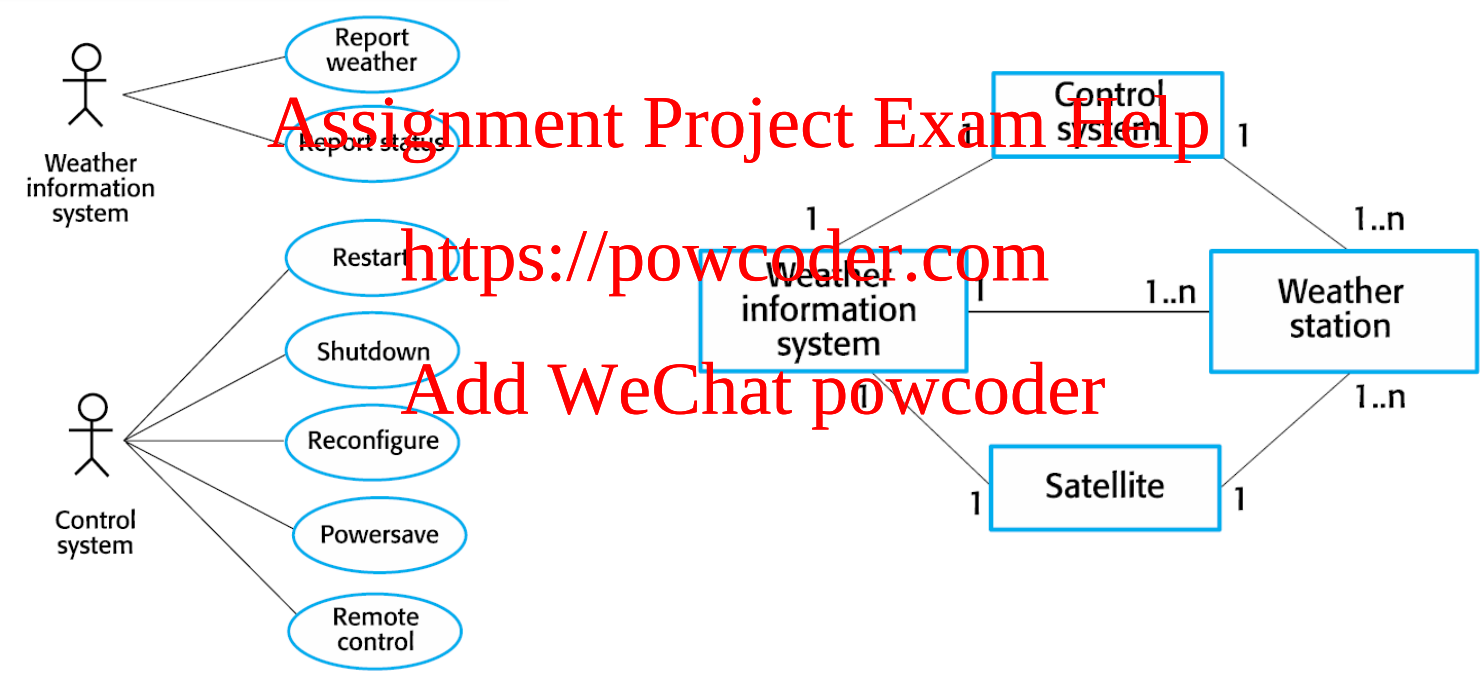
year = 1904, month=2

Path

leap=false return

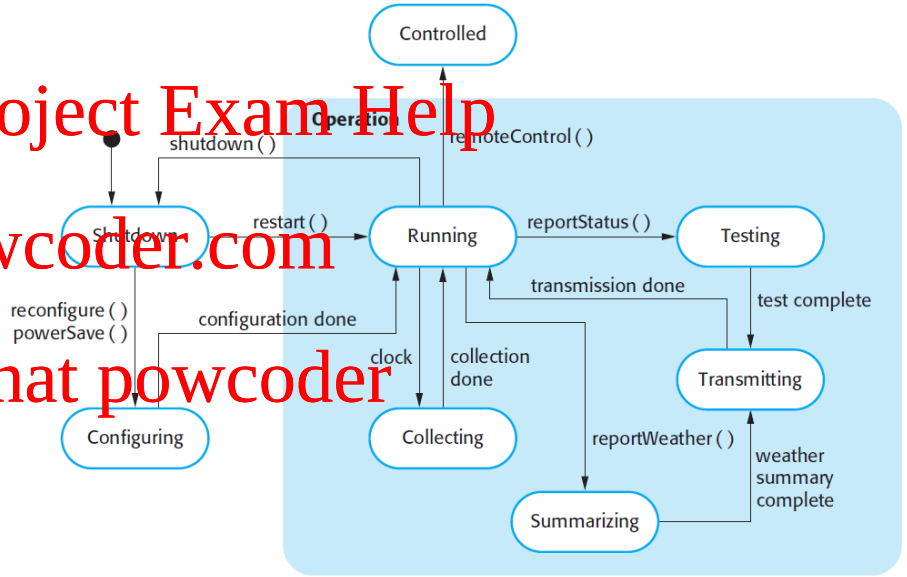
leap=true return

Weather Station Case Study - Revisit



State Coverage

- Each state in a finite-state-machine been exercised reached and explored
- Testing states of *WeatherStation* using state model
  - Identify sequences of state transitions to be tested
  - Define event sequences to force these transitions
- Examples:
  - Shutdown → Running → Shutdown
  - Configuration → Running → Testing → Transmitting → Running
  - Many others…



Which Coverage Criteria?

- Some coverage criteria are implied in other ones
  - Path coverage implies decision, statement and entry/exit coverage
  - Decision coverage implies statement
- Full coverage might be infeasible
  - Module with n decisions
  - Loops can result in infinite number of paths

Coverage Target

- What coverage should one aims for?
- Software criticality determines coverage level
  -
- Extremely high coverage for safety-critical (dependable) software
  - Government/standardization organizations
- E.g., European corporation for space standardization (ESS-E-ST-40C)
  100% statement and decision coverage for 2 out of 4 criticality levels

Coverage – Tests Quality

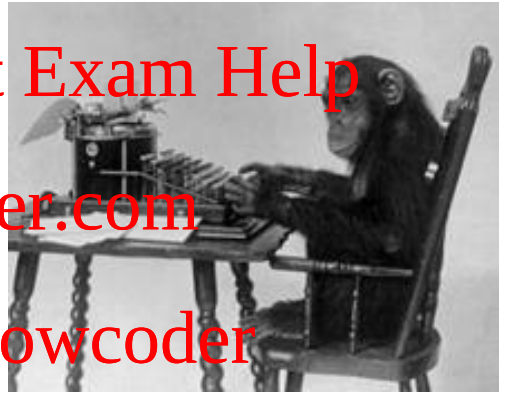
- Coverage percentage might confusingly be linked to a quality target
- Achieving certain coverage target not necessarily useful
  - e.g., software to be deployed when 80% coverage is achieved
  - High coverage numbers can be attained but probably by test smells
- TDD can help but not sufficient to get good test
- Understanding different types of coverage criteria can effectively help in identifying effective test cases

https://martinfowler.com/bliki/TestCoverage.html

Fuzzy Testing (Fuzzing)

Fuzz Testing (Fuzzing)

- Automated software testing techniques that verifies the software behaviour using invalid, unexpected or random data
- Random testing or monkey testing



- *Infinite monkey theorem –*
- *"a monkey hitting keys at random on a typewriter keyboard for an infinite amount of time will almost surely type any given text, such as the complete works of William Shakespeare"*

https://en.wikipedia.org/wiki/Infinite_monkey_theorem

Fuzz Testing

- The Fuzzer would generate particular sequence of inputs that might trigger a crash/fault
- Typically input domain is structured
- Effective fuzzing should generate inputs
  - "Valid enough" to create unexpected behaviours
  - "invalid enough" to expose boundary cases

https://en.wikipedia.org/wiki/Fuzzing

Fuzzing based on Inputs

- Generation-based Fuzzer
  - Inputs are generated from scratch based on a provided model
- Mutation-based Fuzzer
  - Inputs are generated by mutating provided seeds
- Input-structure Fuzzer
  - Generate inputs for software that take structured input
  - Sequence of mouse or keyboard events

https://en.wikipedia.org/wiki/Fuzzing

Fuzzing based Program Structure

- Black-box Fuzzer
  - Unaware of program structure randomly generating inputs
  - Can be run in parallel and scale
  - May discover obvious bugs
- White-box Fuzzer
  - Use program analysis to systematically increase code coverage
  - Might use model-based testing to generate inputs and check outputs against the program specifications

https://en.wikipedia.org/wiki/Fuzzing

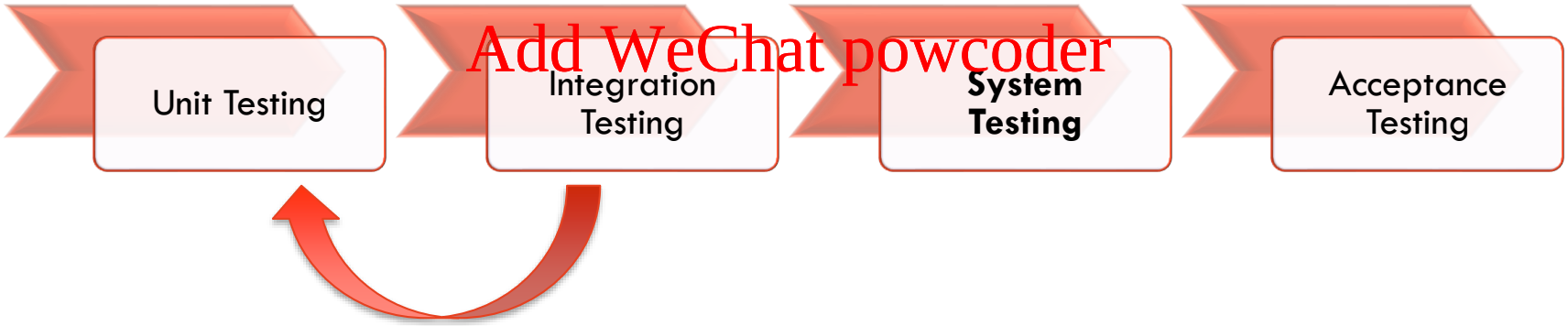
System Testing

System Testing

- Verifying the behavior of the entire system
- Does the system meet the requirements?
  - Functional requirements
  - Non-functional properties (e.g., security, performance)
  - Quality of requirements impact ease of testing
- Also considers external interfaces, operating environment, hardware devices
- Carried out by the development/testing team

Testing Levels – System Testing

- Effective unit and integration testing should have identified many of the software defects
- System testing might uncover undetected defects (defect testing)



System Testing – Functional Testing

- Test system functionality (what the system suppose to do)
- Test cases designed from the system requirements (use cases)
- System as a Black-box

Functional Testing – Example

- Consider the following requirements of a healthcare system

"*If a patient is known to be allergic to any medication, then the prescription of that medication shall result in a warning message being issued to the system user. If a prescriber chooses to ignore an allergy warning, they shall provide a reason why this had been ignored*"

How one can verify that these requirements have been satisfied?

Functional Testing – Example

1. A patient record with no known allergies → prescribed medication for allergies → a warning message is not issued by the system

2. A patient record with a known allergy → prescribe the medication to that the patient is allergic to → a warning is issued by the system

3. A patient record in which allergies to two or more drugs → prescribe these drugs separately → has a correct warning for each drug been issued?

4. Prescribe two drugs that the patient is allergic to → two warnings are correctly issued?

5. Prescribe a drug that issues a warning and overrule that warning → the system must require the user to provide information explaining why the warning was overruled

Scenario based Testing

- Approach for system testing where typical scenarios of system use are used for testing
  - A story describes how the system might be used
  - Test several requirements including functional and non-functional
- Scenarios should be realistic and real system users should reflect the user thinking and system processes
- Scenarios should include deliberate mistakes to verify the system behavior
- Problems should be noted
- Part of the system requirement analysis (use case scenarios)

Scenario based Testing – Example

Consider the usage following scenario for a Mental Health Care-Patient Management System. What functions/non-functional requirements will be tested?

Kate is a nurse who specializes in mental health care. One of her responsibilities is to visit patients at home to check that their treatment is effective and that they are not suffering from medication side effects.

On a day for home visits, Kate logs into the MHC-PMS and uses it to print her schedule of home visits for that day, along with summary information about the patients to be visited. She requests that the records for these patients be downloaded to her laptop. She is prompted for her key phrase to encrypt the records on the laptop.

One of the patients that she visits is Jim, who is being treated with medication for depression. Jim feels that the medication is helping him but believes that it has the side effect of keeping him awake at night. Kate looks up Jim's record and is prompted for her key phrase to decrypt the record. She checks the drug prescribed and queries its side effects. Sleeplessness is a known side effect so she notes the problem in Jim's record and suggests that he visits the clinic to have his medication changed. He agrees so Kate enters a prompt to call him when she gets back to the clinic to make an appointment with a physician. She ends the consultation and the system re-encrypts Jim's record.

After, finishing her consultations, Kate returns to the clinic and uploads the records of patients visited to the database. The system generates a call list for Kate of those patients who she has to contact for follow-up information and make clinic appointments.

System Testing – Non-Functional Properties

- Test how well the system behave regarding non-functional properties
  - Performance, security, usability, reliability, other
- Defined as non-functional requirements from which test cases are designed
  - Quality attributes or "*ilities*"
  - Typically defined precisely and associated with quantifiable measures used (e.g., response time, availability)
- Testbeds and real production environments

Performance Testing

- Verify how the system performs under intended workload
- Typically performance measures
- Speed: amount of time an application needs to perform a task
  - Once a user input search terms, search results should be displayed in less than 400ms
  - 95% of transactions should be executed in less than 2 seonds
- Throughput: the amount of work an application must perform in unit time
  - The application should handle 800 transactions per second
- Scalability:
  - How does an application perform when number of simultaneous users increase
  - How does an application perform as the data it processes increases in size

Types of Performance Testing

- Load Testing
  - Verify the system performance under expected workload (number of users/transactions)
  - Identify performance bottlenecks (hardware and software)
- Stress Testing
  - Verify software behavior at the maximum and beyond the designed load
  - Determine the behavioral limits and test robustness in mission-critical software
- Spike testing
  - Verify the system performance under sudden workload increase/decrease
  - Determine how well the system will handle such sudden changes

https://en.wikipedia.org/wiki/Software_performance_testing#Load_testing

Types of Performance Testing

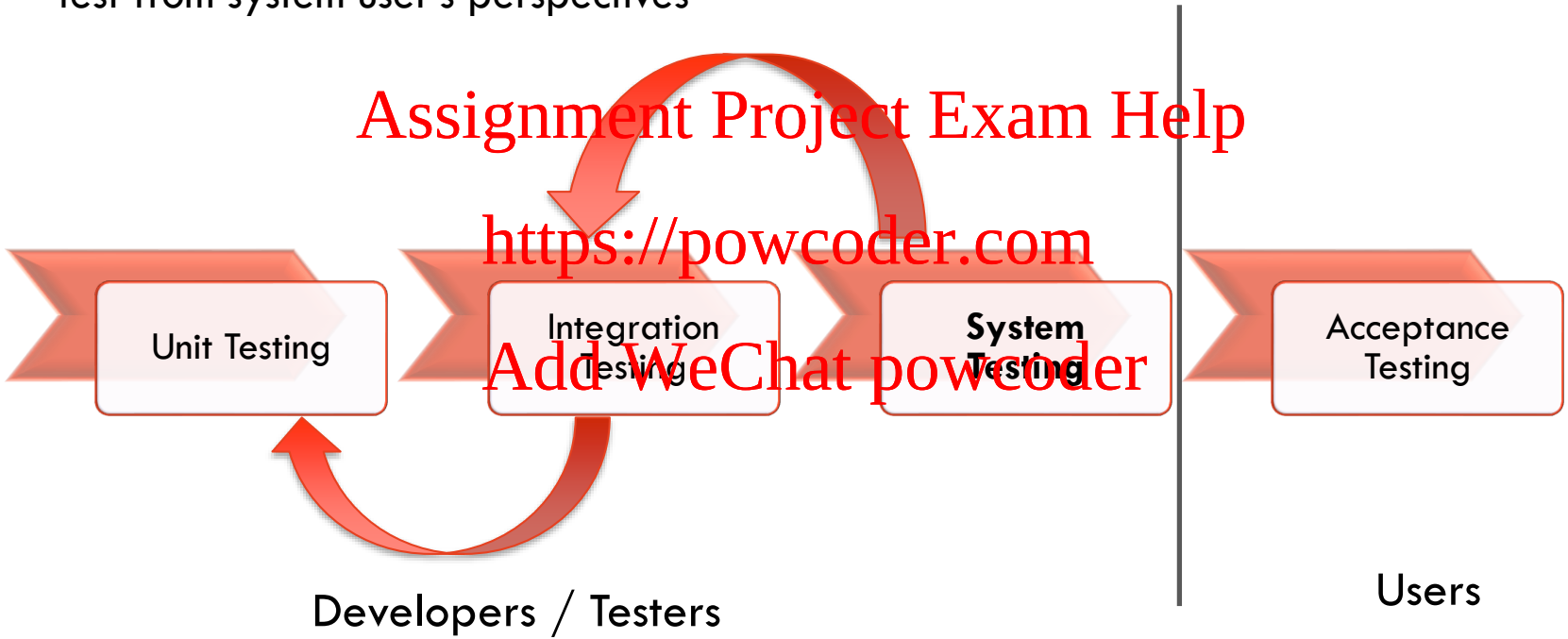
- Soak (endurance) Testing
  - Verify how the system perform under a typical production load over extended period of time
  - E.g., memory leaks could trigger after certain time duration of a typical production load
  - Resources leak, performance degradation over time
- Configuration Testing
  - Verify system performance under different configuration changes to system's components
  - Server configuration, software configuration

Acceptance Testing

THE UNIVERSITY OF SYDNEY

User Acceptance Testing

- Test from system user's perspectives



User (Acceptance) Testing

- Users test the system by providing input and assess the system behavior
- Developers/testing team try to replicated user's environment and interactions
- Users determine if the system behave as expected in their work environment in terms of functional and non-functionality aspects
- Alpha, Beta and User testing

User Testing – Alpha Testing

- Users use the software at the developer's environment where the software used in a controlled setting

- Users identify problems that are not obvious to the developers

- Provide early validation of the system functionality

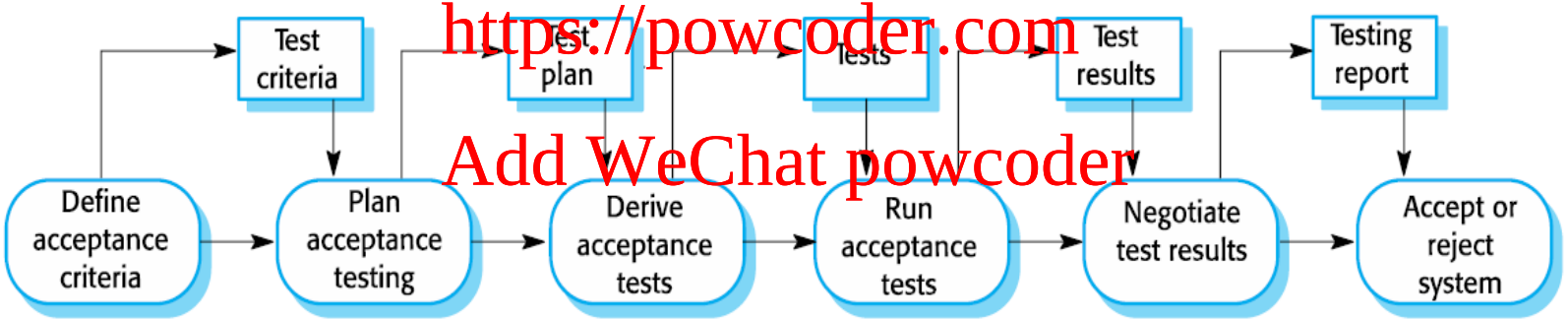
- Agile methods involve users in testing developed functions in every development iteration

User Testing – Beta Testing

- User use a release of the software in their environment to verify system behavior
- Group of lead users/customers
- Also to discover problems between the software and features of the environment

User Testing – Acceptance Testing

- Users test the system and decide whether it is ready for use
- Previously agreed acceptance criteria



User Testing – Acceptance Testing

- Define acceptance criteria (early as part of the system contract
- Plan acceptance testing
  - Resources, test plan, and required coverage
- Derive acceptance tests
  - Design tests that cover both functional and non-functional requirements
- Run tests
  - Ideally in actual environment by end users
- Negotiate test results
  - Based on acceptance criteria, identified issues to be addressed by developers
- Accept/reject
  - Good enough for use or further development is needed

Software Testing Classification

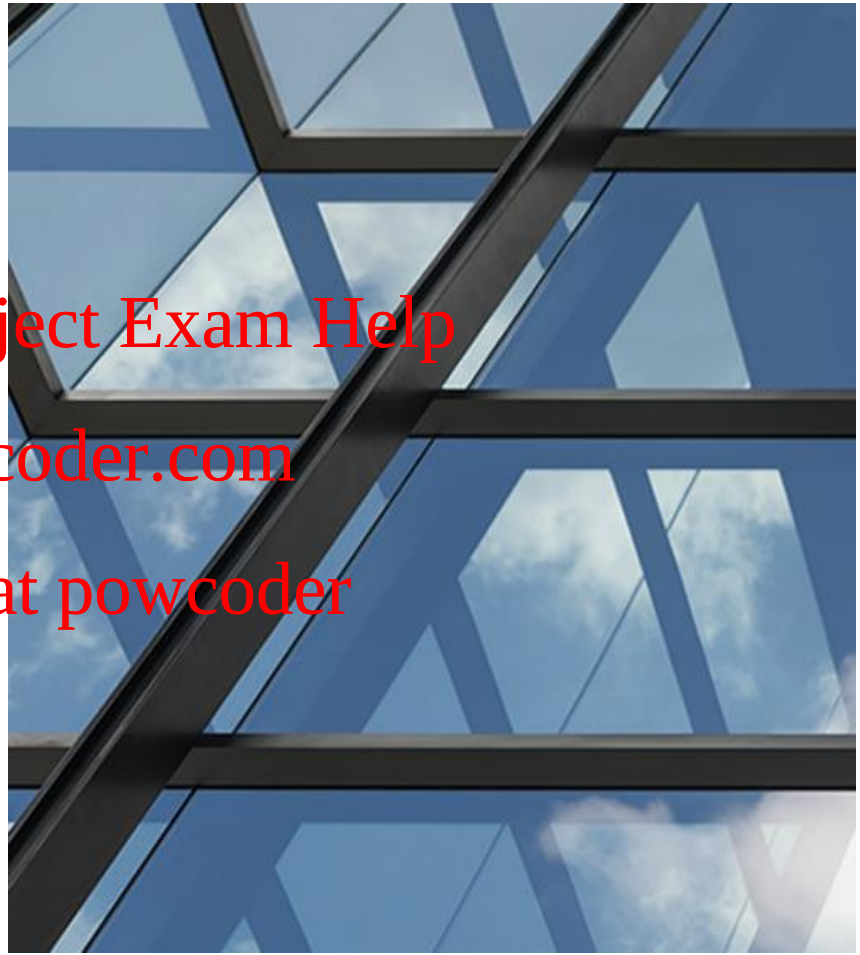
Functional Testing

- *Unit testing*
- *Integration testing*
- *System testing*
- *Regression testing*
- Interface testing
- User Acceptance Testing (UAT) – Alpha and Beta testing
- Configuration, smoke, sanity, end-to-end testing

Non-Functional Testing

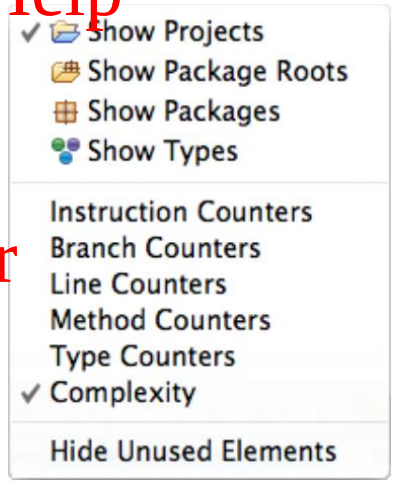
- *Performance testing*
- *Load testing*
- Security testing
- *Stress testing*
- Reliability testing
- Usability testing

Test/Code Coverage



Tools for Code Coverage

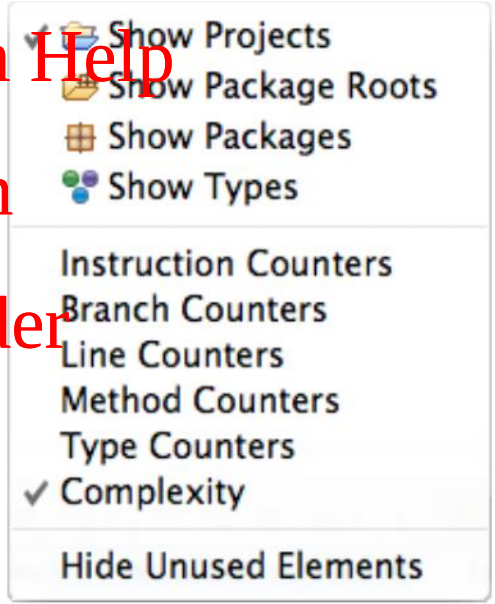
- There are many tools/plug-ins for code coverage in Java
- Example: EclEmma*
- EclEmma is a code coverage plug-in for Eclipse
  - It provides rich features for code coverage analysis in Eclipse IDE
  - EclEmma is based on the JaCoCo code coverage library
    - *Ja*CoCo is a free code coverage library for Java, which has been created by the EclEmma team



https://www.eclemma.org/

EclEmma – Counters

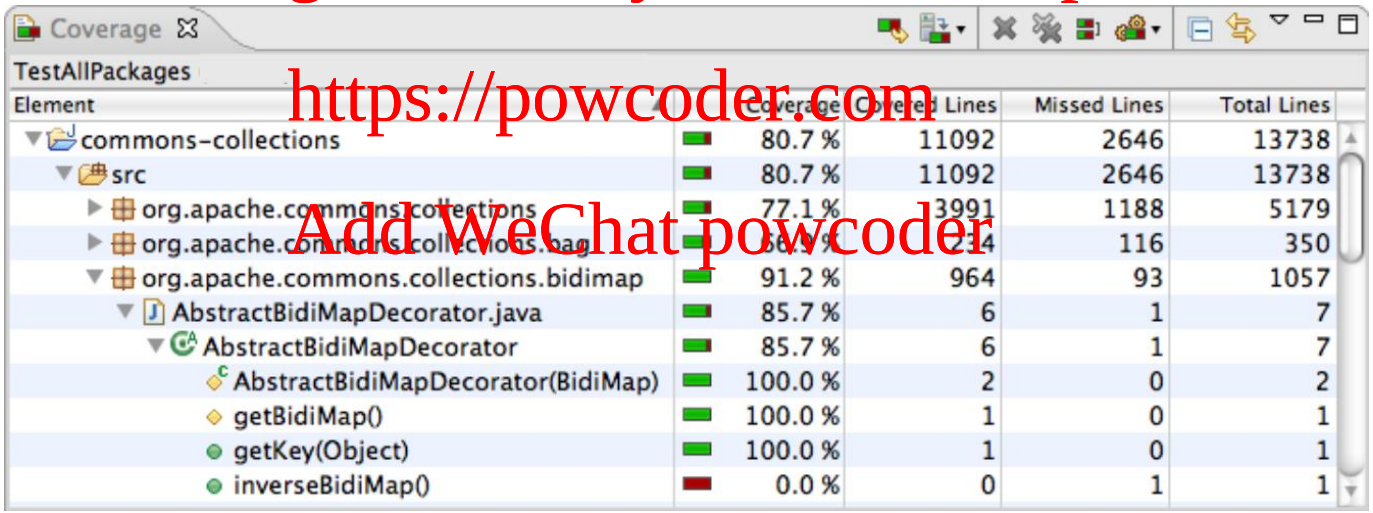
- EclEmma supports different types of counters to be summarized in code coverage overview
  - bytecode instructions, branches, lines, methods, types and cyclomatic complexity
  - Should understand each counter and how it is measured
  - Counters are based on JaCoCo – see JaCoCo documentation for detailed counter definitions



https://www.eclemma.org/

EclEmma Coverage View

The ***Coverage view*** shows all analyzed Java elements within the common Java hierarchy. Individual columns contain the numbers for the active session, always summarizing the child elements of the respective Java element



https://www.eclemma.org/

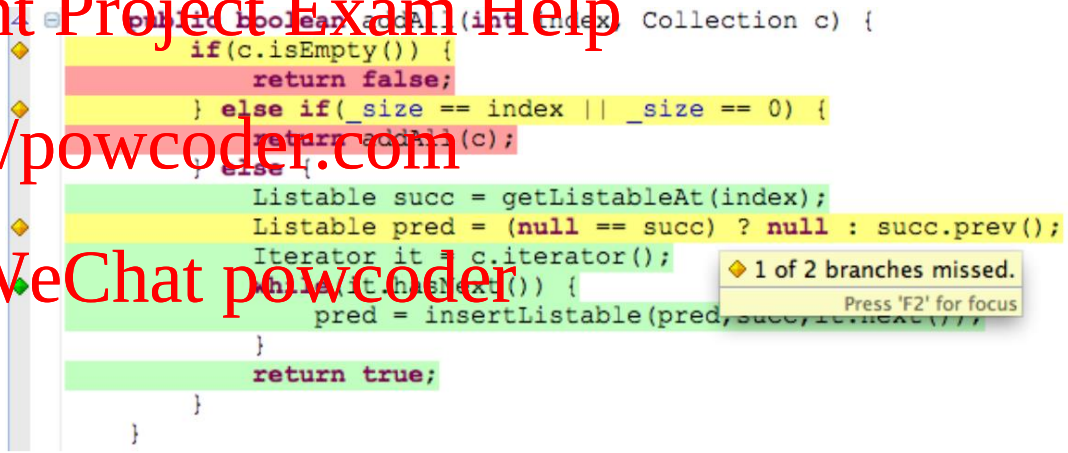
EclEmma Coverage – Source Code Annotations

Source lines color code:

- **green** for fully covered lines,
- **yellow** for partly covered lines (some instructions or branches missed)
- **red** for lines that have not been executed at all

Diamonds color code

- **green** for fully covered branches,
- **yellow** for partly covered branches
- **red** when no branches in the particular line have been executed.



https://www.eclemma.org/

References

- Ian Sommerville. 2016. Software Engineering (10th ed.) Global Edition. Pearson, Essex England
- Bernd Bruegge and Allen H. Dutoit. 2009. Object-Oriented Software Engineering Using Uml, Patterns, and Java (3rd ed.). Pearson.
- Martin Fowler, Test Coverage. https://martinfowler.com/bliki/TestCoverage.html
- Software Performance Testing, https://en.wikipedia.org/wiki/Software_performance_testing
- Software Testing, https://en.wikipedia.org/wiki/Software_testing
- Fuzz Testing (Fuzzing) https://en.wikipedia.org/wiki/Fuzzing

Next Lecture/Tutorial…

W5 Tutorial: Test Techniques 2 + W5 quiz

W5 Lecture: Review and Overview of Design Patterns

Testing Assignment A2 release

THE UNIVERSITY OF SYDNEY