SZÉCHENYI EGYETEM
UNIVERSITY OF GYŐR

# System and Software Testing (GKNM_INTA057) Rendszer és szoftvertesztelés (GKN/LM_INTM057)

Csapó Ádám - csapo.adam@sze.hu
Varjasi Norbert – norbert.varjasi@sze.hu

Széchenyi István University
Department of Information Technology

# Agenda

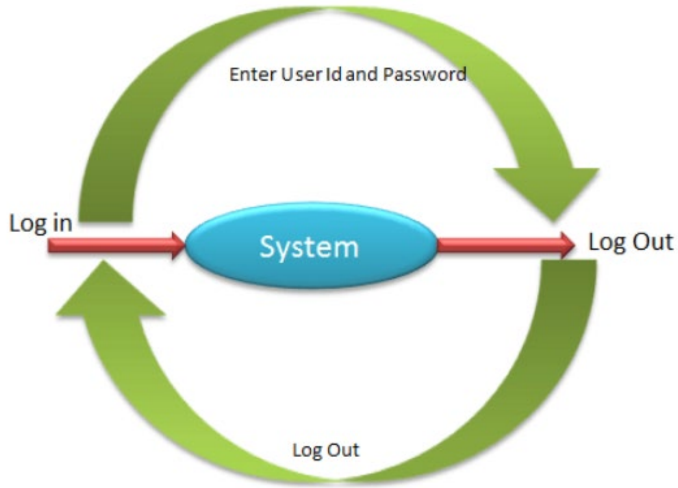
- Earlier we spoke about different kinds of dynamic testing
- Specification-based testing
  - Equivalence partitions, boundary testing, cause-effect analysis
- Today’s topics:
  - Model-based and structure-based techniques

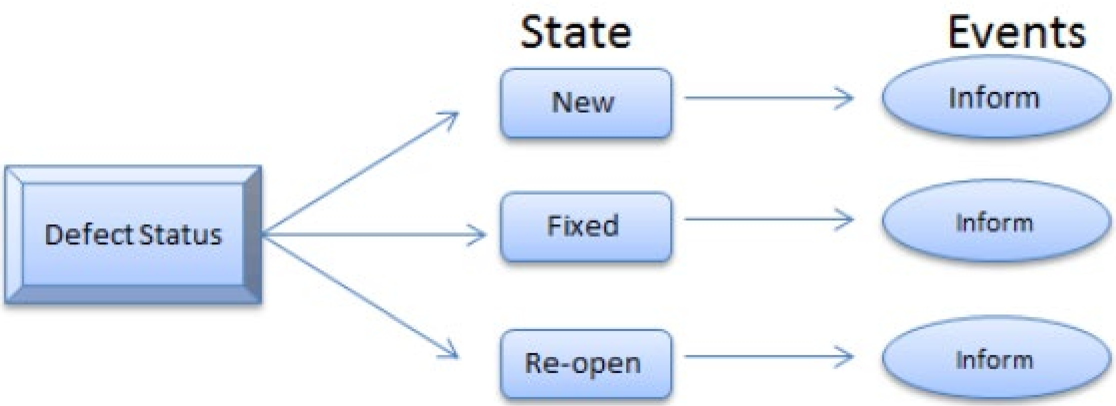
# Model-based techniques

- Model-based testing is a sw testing technique where run time behavior of software is checked against predictions made by a model
- A model can be any description of a system's behavior – whether hand-created or automatically generated
  - Key concepts can include input sequences, actions, conditions, output and flow of data from input to output
- Typical models include: data flow, control flow, dependency graphs, decision tables, state-transition machines

# Rudimentary examples

- Examples of a finite state machine and state chart (state charts are an extension of finite state machines with events associated with states):



# Model-based testing in Javascript

- A good example is the xstate framework
- Xstatejs.org

## @xstate/test

The @xstate/test package contains utilities for facilitating model-based testing for any software.

- Talk: Write Fewer Tests! From Automation to Autogeneration at React Rally 2019 (🎥 Video )

### Quick Start

1. Install `xstate` and `@xstate/test` :

```sh
npm install xstate @xstate/test
```

2. Create the machine that will be used to model the system under test (SUT):

```js
import { createMachine } from 'xstate';

const toggleMachine = createMachine({
  id: 'toggle',
  initial: 'inactive',
  states: {
    inactive: {
      on: {
        TOGGLE: 'active'
      }
    },
    active: {
      on: {
        TOGGLE: 'inactive'
      }
    }
  }
});
```

# Structure-based techniques

- By principle, structure-based techniques are white-box, since they consider the structure of the code
- Goal is to get as high a code coverage as possible
- A key tool: control flow graph (CFG)
  - Nodes: statements in the program
  - Directed edges: jumps between statements
- Cyclomatic complexity is a number that tells you how many independent paths there are within the CFG
  - This is useful in formulating an upper limit for tests (and also for understanding how complex the code is)

# Structure-based techniques

- Cyclomatic complexity:
  - By ‘independent paths’, we mean a set of paths in which each path contains at least one edge that is not contained in any of the others
- Note that not all paths will be guaranteed to be reachable, so CC is a theoretical upper limit
- Goal is to cover all paths, but complete coverage still won’t guarantee that all bugs are found.

# Structure-based techniques

- Types of coverage (besides path coverage):
  - Statement coverage
    - Number of statements executed at least 1x divided by all statements
    - Does not guarantee full coverage due to complex conditions and potential branches in 1 statement

# Structure-based techniques

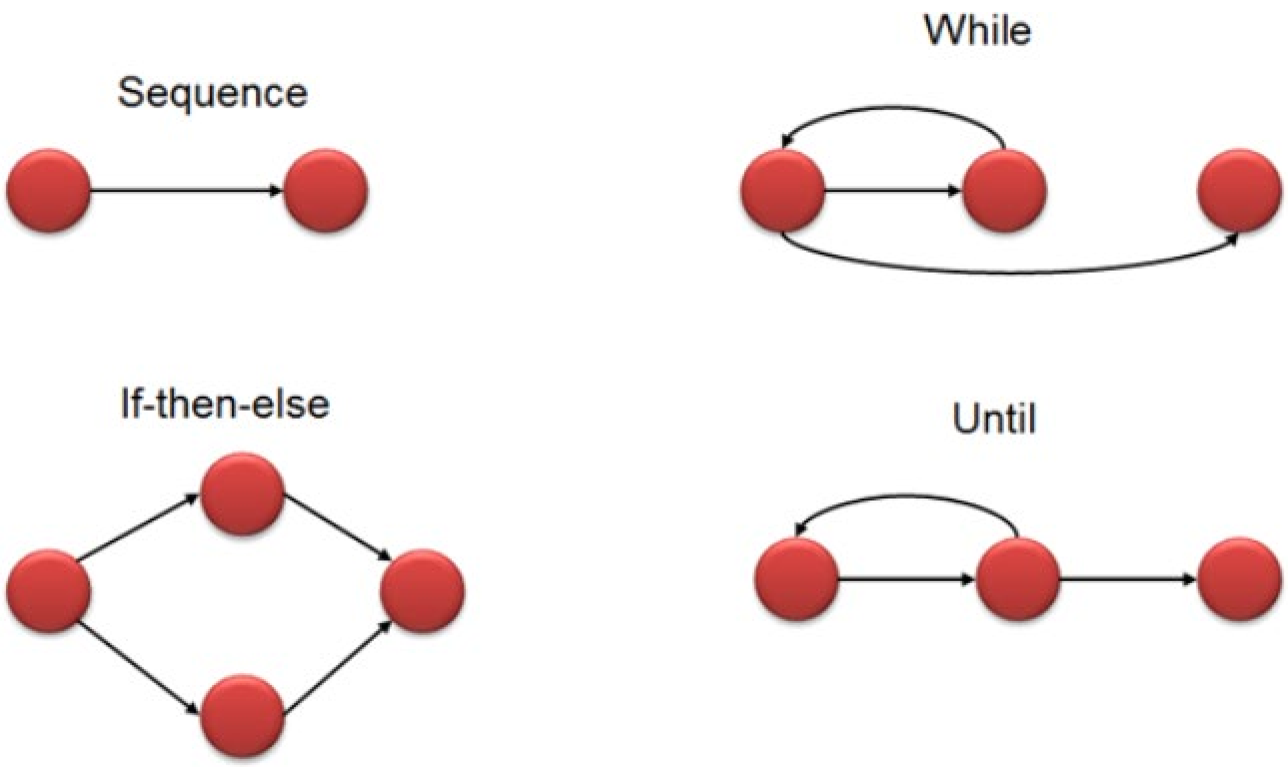
- Types of coverage (besides path coverage):
  - Branch coverage
    - Number of branches covered divided by total number of branches
    - Still not complete, since branches can follow one another: not all permutations are guaranteed to be covered

# Structure-based techniques

- Types of coverage (besides path coverage):
  - Path coverage
    - Number of paths covered divided by total number of paths
    - Implies statement and branch coverage
    - Very strict requirement. Ability to achieve 100% is not guaranteed

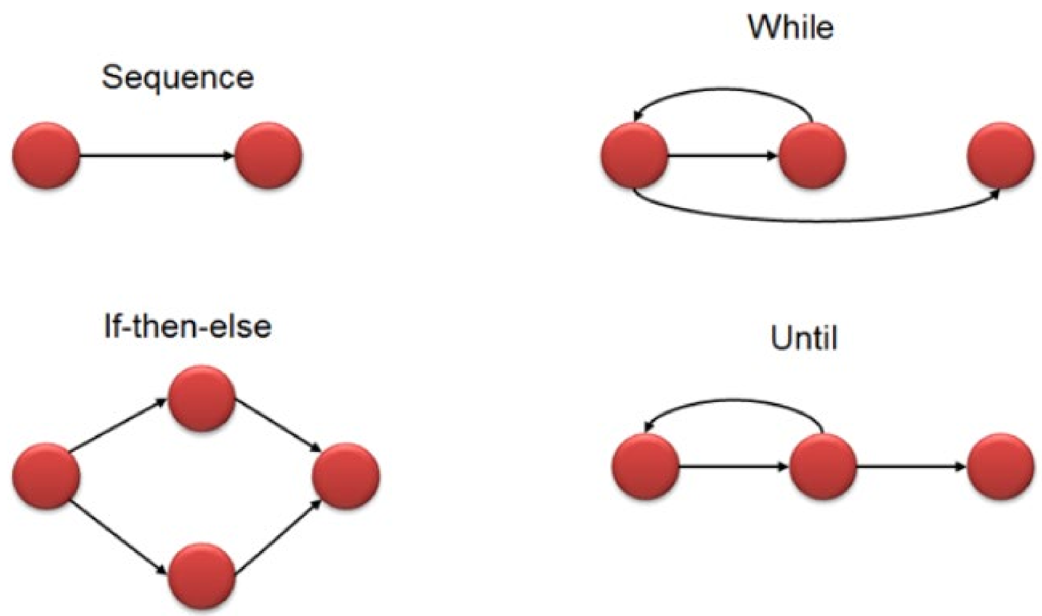
# A principles-first look at cyclomatic complexity

- What kinds of branches can we imagine?



# A principles-first look at cyclomatic complexity

- What kinds of branches can we imagine?
- Mathematically: If we have E edges and N nodes, CC will be:
  - CC = E – N + 2



# A principles-first look at cyclomatic complexity

- Note that long sequences can be reduced. If we have a sequence like:
  - Node1 -> Node2 -> Node3

… and we reduce it to:

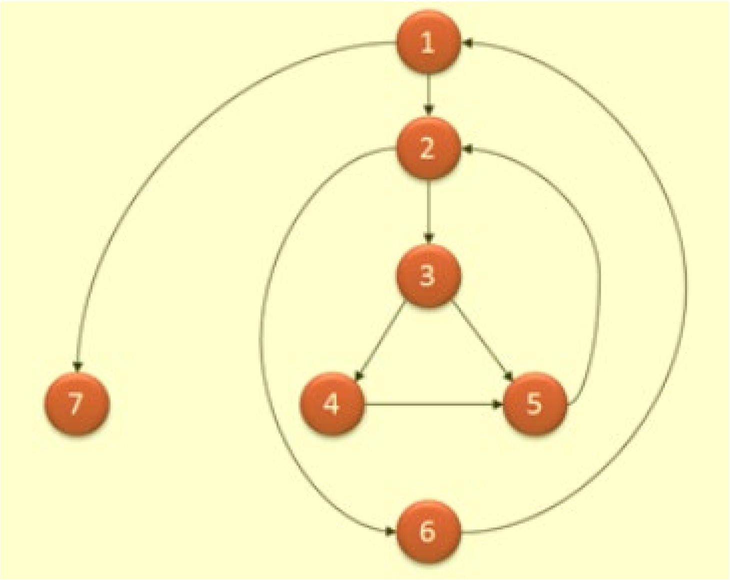
  - Node1 -> Node3

The cyclomatic complexity is the same.

# A principles-first look at cyclomatic complexity

- How does the graph on the right arise from this code?

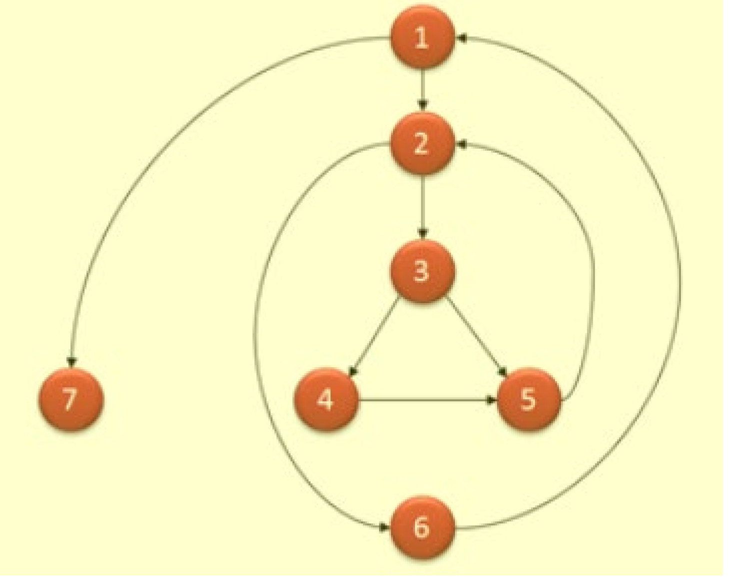
```
i = 0;

n=4; //N-Number of nodes present in the graph

while (i<n-1) do

j = i + 1;

while (j<n) do

if A[i]<A[j] then

swap(A[i], A[j]);

end do;

i=i+1;

end do;
```



# A principles-first look at cyclomatic complexity

- How does the graph on the right arise from this code?
  - Lines that are crossed out are sequences (can be removed)
  - End do; statements return to while statement for a (potential) final check

```
    i = 0;
    n=4; //N-Number of nodes present in the graph
1   while (i<n-1) do
    j = i + 1;
2   while (j<n) do
3   if A[i]<A[j] then
4   swap(A[i], A[j]);
5   end do;
    i=i+1;
6   end do;
```



# Cyclomatic complexity rules of thumb

| Complexity Number | Meaning |
|---|---|
| 1-10 | Structured and well written code<br>High Testability<br>Cost and Effort is less |
| 10-20 | Complex Code<br>Medium Testability<br>Cost and effort is Medium |
| 20-40 | Very complex Code<br>Low Testability<br>Cost and Effort are high |

# Cyclomatic complexity can be checked using many tools, incl. eslint

.eslintrc.js

```
module.exports = {
    root: true,
    parser: '@typescript-eslint/parser',
    plugins: [
      '@typescript-eslint',
    ],
    rules: {
      "complexity": [2,3]
    },
    extends: [
      'eslint:recommended',
      'plugin:@typescript-eslint/recommended',
    ],
};
```

package.json

```
{
  "name": "typescripttest",
  "version": "1.0.0",
  "description": "I can write a description",
  "main": "index.ts",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "lint": "npx eslint . --ext .js,.jsx,.ts,.tsx",
    "complex": "npx eslint -f complexity index.ts || true",
    "build": "tsc",
    "start": "node dist/index.js"
  },
  "author": "Adam Csapo",
  "license": "ISC",
  "devDependencies": {
    "@typescript-eslint/eslint-plugin": "^5.17.0",
    "@typescript-eslint/parser": "^5.17.0",
    "eslint-formatter-complexity": "latest",
    "eslint": "^8.12.0",
    "typescript": "^4.6.3"
  }
}
```

```
c:\Users\Adamc\Pcloud\Teaching-2022\SystemAndSwTesting\01-StaticAnalysis\TypescriptExample>npm run lint

> typescripttest@1.0.0 lint c:\Users\Adamc\Pcloud\Teaching-2022\SystemAndSwTesting\01-StaticAnalysis\TypescriptExample
> npx eslint . --ext .js,.jsx,.ts,.tsx


c:\Users\Adamc\Pcloud\Teaching-2022\SystemAndSwTesting\01-StaticAnalysis\TypescriptExample\index.ts
  11:7   warning  'someFunction' is assigned a value but never used       @typescript-eslint/no-unused-vars
  15:7   warning  'complexFunction' is assigned a value but never used    @typescript-eslint/no-unused-vars
  15:25  error    Arrow function has a complexity of 5. Maximum allowed is 3  complexity

✖ 3 problems (1 error, 2 warnings)
```

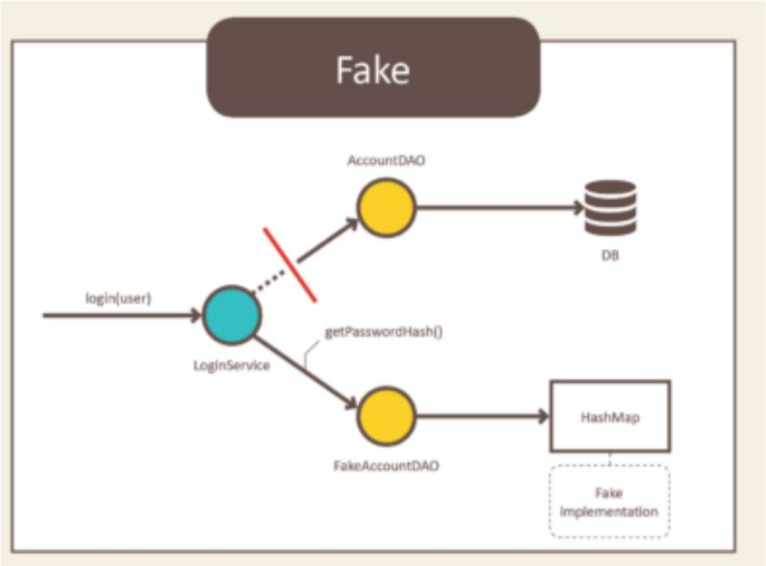
# Simulation using test doubles

- In general, it is improbable that we can (or even want to) test a complex system in its full integration
  - We want to have guarantees per function, per module, per subsystem etc. so that we aren't confronted with complete chaos all at once.

- Three common types of doubles: fakes, stubs, mocks

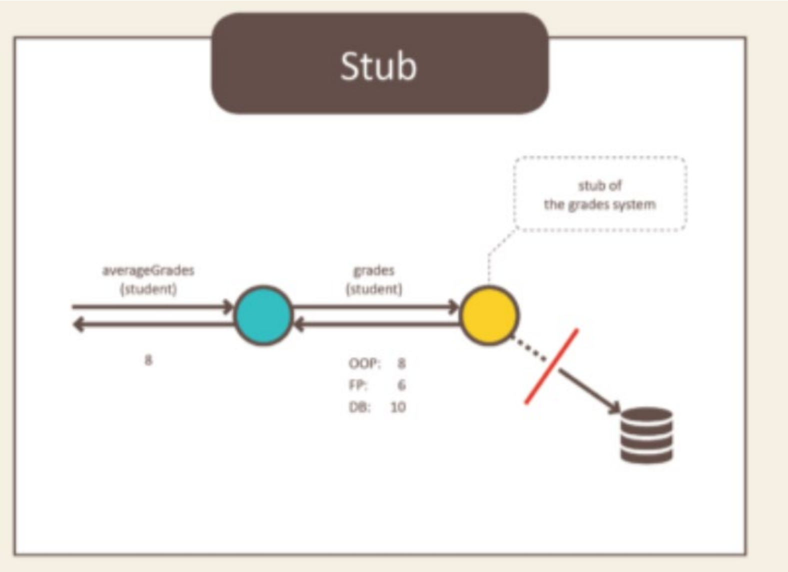
# Simulation using test doubles

- Fake: the module that is substituted has a dynamic implementation, but it is different from the final implementation
  - For example, instead of turning to a real database, it uses an in-memory data structure to read / write data



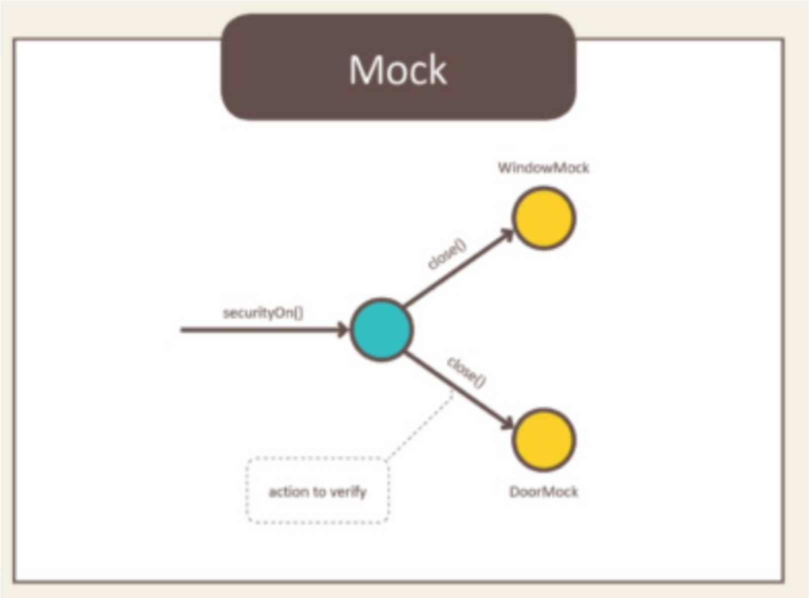
# Simulation using test doubles

- Stub: the module that is substituted has a static implementation that returns “burnt-in” data
  - Imagine that you have a file with fixed values, the module just reads that file and returns the same values every time



# Simulation using test doubles

- Mock: the module that is substituted contains no functionality but generates some artifact (in the simplest case, log messages) so that its operation can be simulated and checked later



# Other types of dynamic testing

- Increasingly, we are seeing software that are data-driven and environment-dependent
- Think of a Machine Learning model – how do you test to make sure it is correct?
  - Too much uncertainty in input data (user gestures, speech etc.), possible non-determinism in functionality
- Important to adopt a hypothesis-testing based scientific approach: a priori criteria, sanity checks etc.