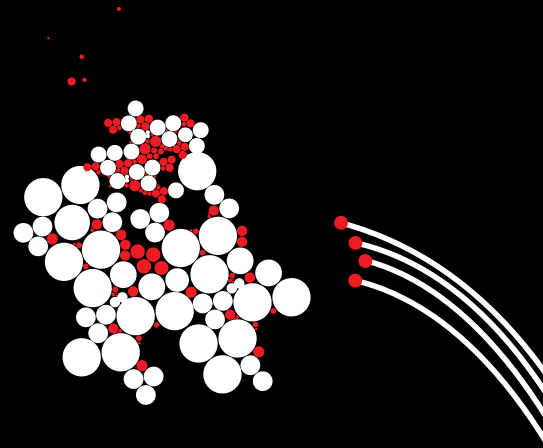
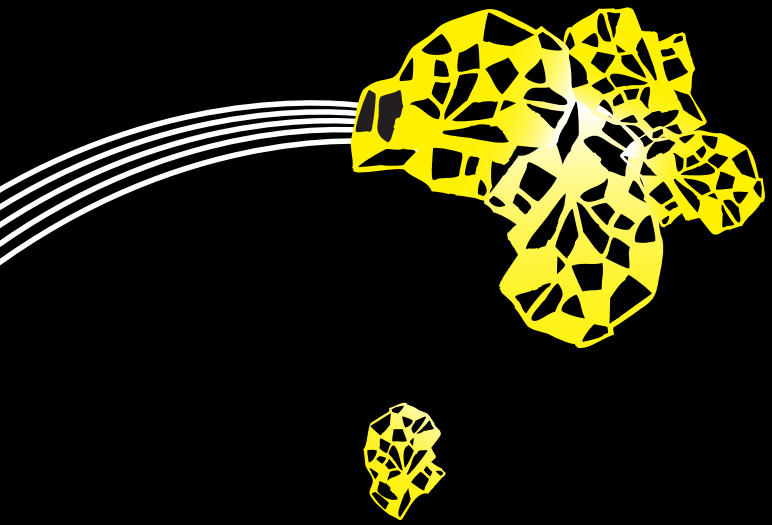
UNIVERSITY OF TWENTE.

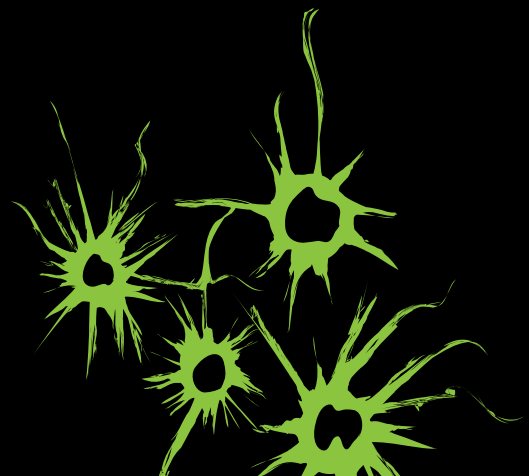


Testing in software engineering

Software Testing and Risk Assessment

Lecture 4a





Today

1. Background of teacher and students
2. Course overview
3. Summary of testing basics
4. Regression testing and Test driven development
5. System testing with Behaviour-Driven Development
6. Software Testing project
7. Classical black-box and white-box testing
8. Integration testing

Petra van den Bos

- Assistant professor in the Formal Methods and Tools group (FMT)
- Research: software quality, model-based testing
- Email: `p.vandenbos@utwente.nl`
- Teaching in STAR: Software Testing

What is your background in testing?

Wooclap quiz

Software Testing lecture overview

- L4a: Testing in software engineering
- L4b: Model-based testing with Labeled Transition Systems
- L5: Conformance relation ioco and test generation algorithms
- L6: Symbolic Transition Systems
- L7: Model-based testing with Axini Modelling Platform (tool)

Software Testing lecture overview

- L4a: Testing in software engineering
- L4b: Model-based testing with Labeled Transition Systems
- L5: Conformance relation ioco and test generation algorithms
- L6: Symbolic Transition Systems
- L7: Model-based testing with Axini Modelling Platform (tool)

Week	When	What	Who	Topic	
6	Wed Feb 5 13:45-15:30	Lecture	MLZ	What is Risk, FMEA	Risk Assessment
	Thu Feb 6 13:45-15:30	Tutorial	RS		
7	Tue Feb 11 10:30-12:30	Lecture	MLZ	Fault trees	
	Wed Feb 12 13:45-15:30	Tutorial	RS		
8	**BREAK**				
9	Tue Feb 25 13:45-15:30	Lecture	MLZ	Dynamic FTs	
	Wed Feb 26 13:45-15:30	Tutorial	RS		
10	Wed Mar 5 8:45-10:30	Lecture	PvdB	Testing in Software Engineering	Testing
	Fri Mar 7 8:45-10:30	Lecture	PvdB	Model-based testing with Labeled Transition Systems	
11	Tue Mar 11 10:45-12:30	Tutorial	RS	Testing in SE, MBT with LTS	
	Thu Feb 13 8:45-10:30	Lecture	PvdB	Test generation and ioco	
12	Mon Mar 17 13:45-15:30	Tutorial	RS	Test generation and ioco	
	Wed Mar 19 8:45-10:30	Lecture	PvdB	Symbolic Transition Systems	
13	Mon Mar 24 13:45-15:30	Lecture	Axini	Axini modeling platform	
	Wed Mar 26 8:45-10:30	Tutorial	RS	Symbolic Transition Systems	
14	Tue Apr 1 13:45-15:30	Lecture	Axini	Axini modeling platform	
	Tue Apr 3 13:45-15:30	Tutorial	RS/Axini	Testing project finalization	
16	Apr 15 13:45-16:30			**EXAM**	

Deadlines

- Homework 4: Wednesday March 12 at 23:59
- Homework 5: Tuesday March 18 at 23:59
- Homework 6: Wednesday March 26 at 23:59
- SmartDoor assignment: Monday March 31 at 23:59
- Software Testing Project: Friday April 11 at 23:59

Testing basics

Why testing?

Why testing?

- It is impossible to write code without any bugs
- Bugs cause problems for users (costs, time loss, accidents,. . .)

What is a test?

A test consists of:

- Setup (if needed)
- Operations/actions to execute program/system
- A check whether expected result has been obtained
  - Gives Pass or Fail in test execution

Tests on different levels of granularity

- Unit tests
- Integration tests (integration of components)
- System/User/End-to-end tests
- User acceptance tests (in production environment)

Tests on different levels of granularity

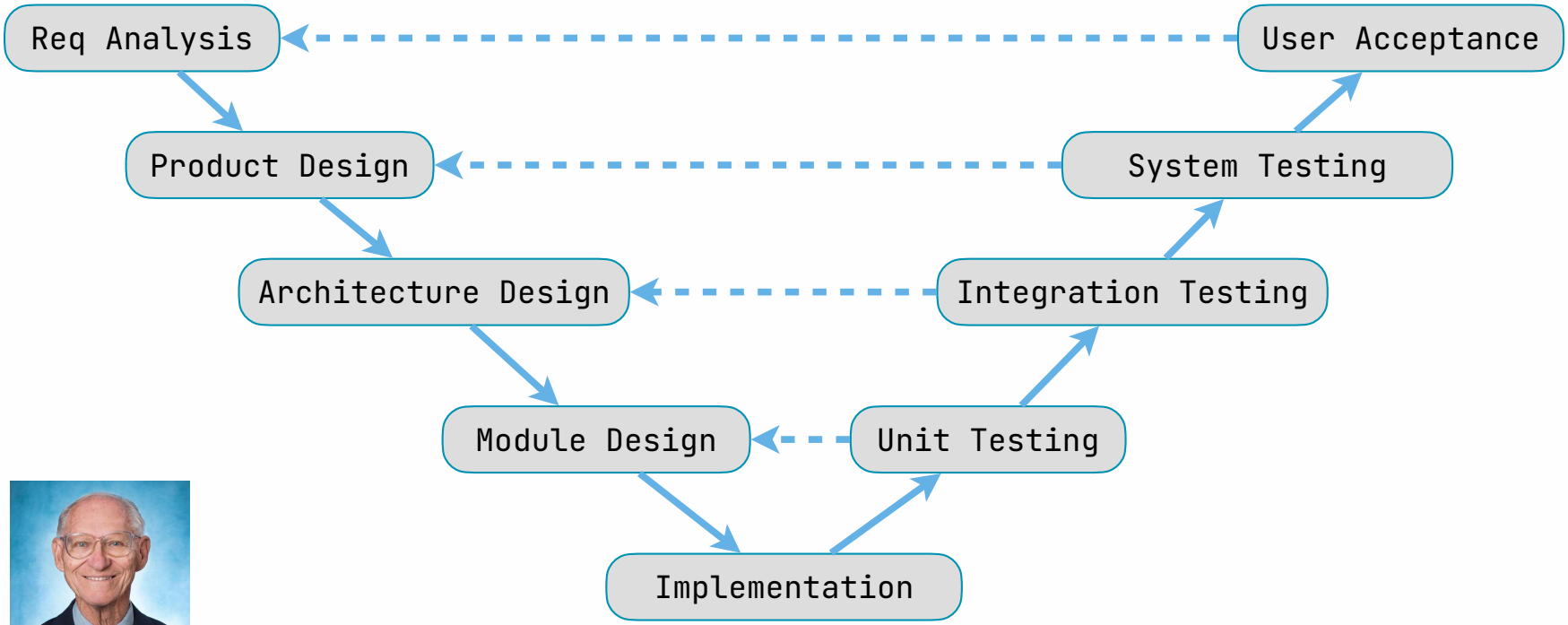
- Unit tests
- Integration tests (integration of components)
- System/User/End-to-end tests
- User acceptance tests (in production environment)

Operations on different levels:

- unit: single method call
- integration: calls to methods of components' interfaces
- system: inputs to and outputs from the user interface

V-model

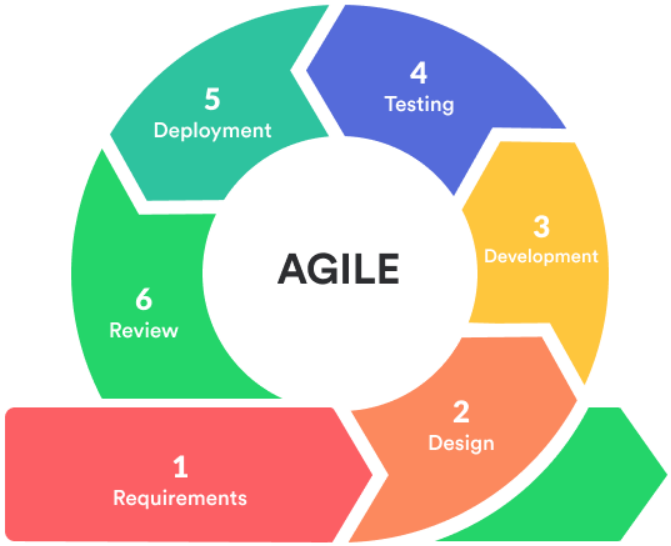
V-Model



B.W.Boehm, *Guidelines for verifying and validating software requirements and design specification*, EuroIFIP, 1979.

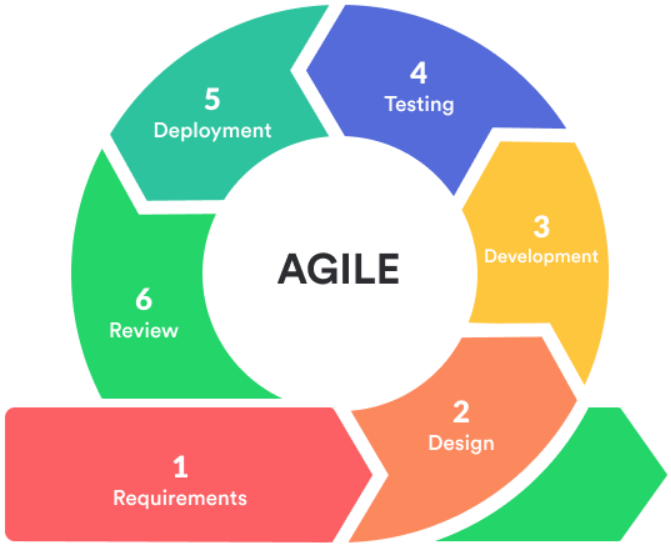
*Above slide occurs in L1T2 of M2, TCS

Agile



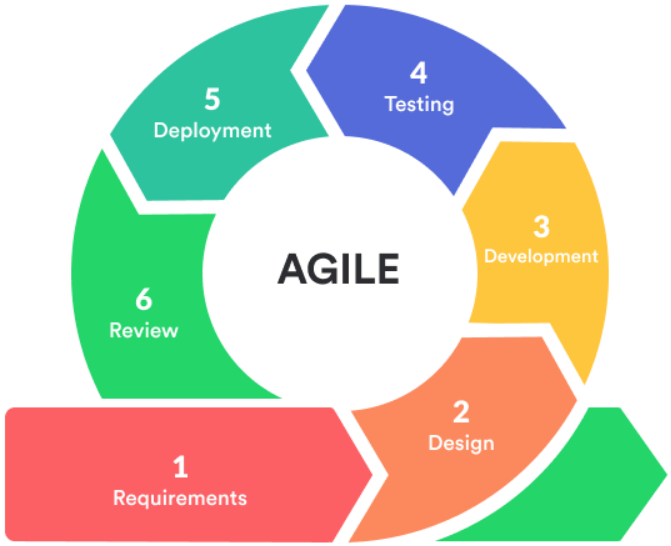
- Some aspects of agile w.r.t. testing:
  - No Big Upfront steps: only code and tests

Agile



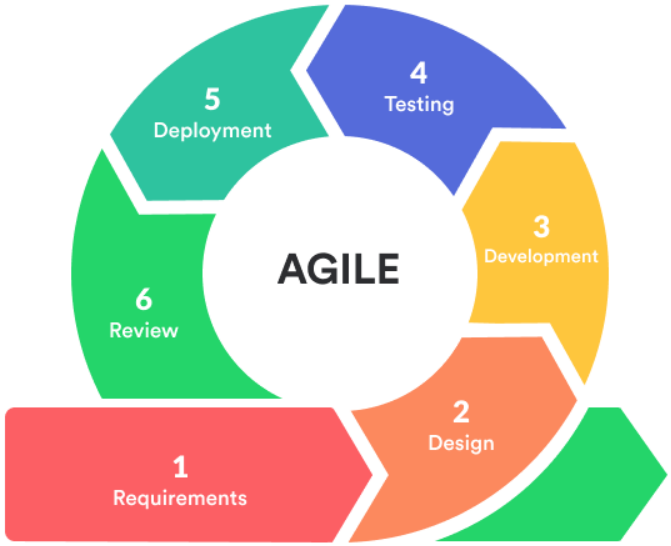
- Some aspects of agile w.r.t. testing:
  - No Big Upfront steps: only code and tests
  - Focus on quality, measured through testing

Agile



- Some aspects of agile w.r.t. testing:
  - No Big Upfront steps: only code and tests
  - Focus on quality, measured through testing
  - Test first (Test-Driven Development)

Agile



- Some aspects of agile w.r.t. testing:
  - No Big Upfront steps: only code and tests
  - Focus on quality, measured through testing
  - Test first (Test-Driven Development)
  - Specify requirements through scenarios

Types of testing

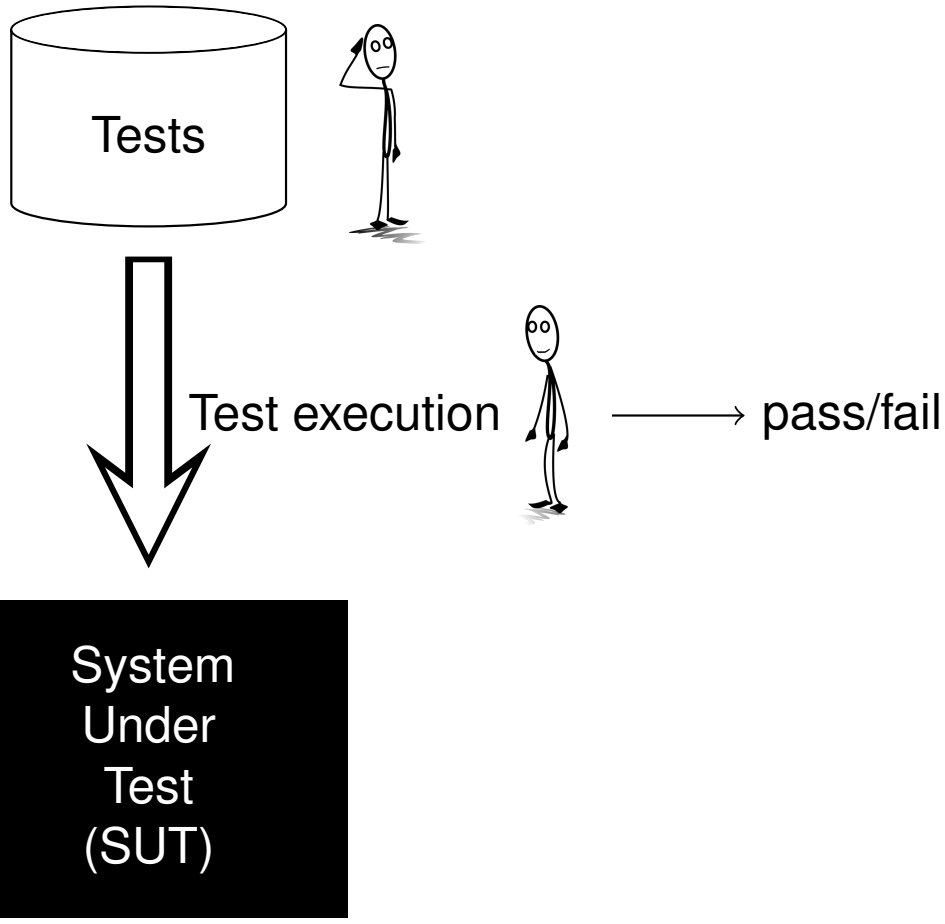
Test:

- **functionality**
- security
- performance
- usability
- reliability
- ...

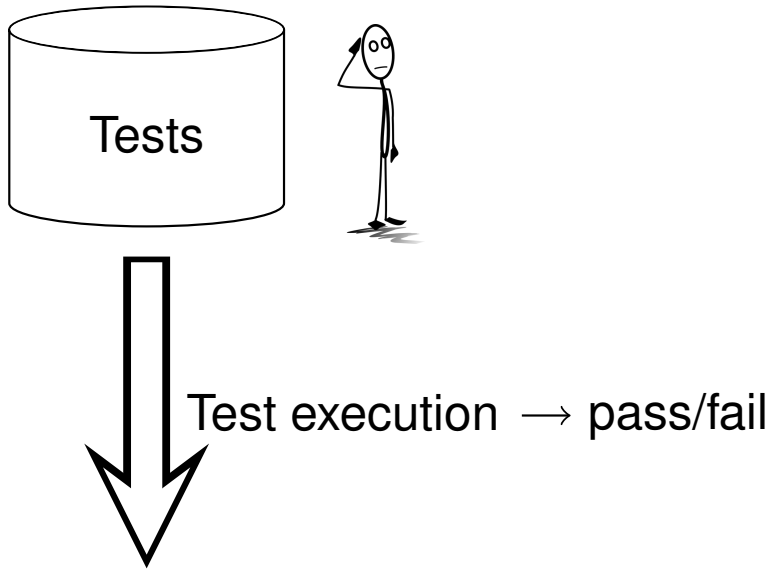
Levels of automation in testing

- Manual testing
- Automated testing
- Model-based testing

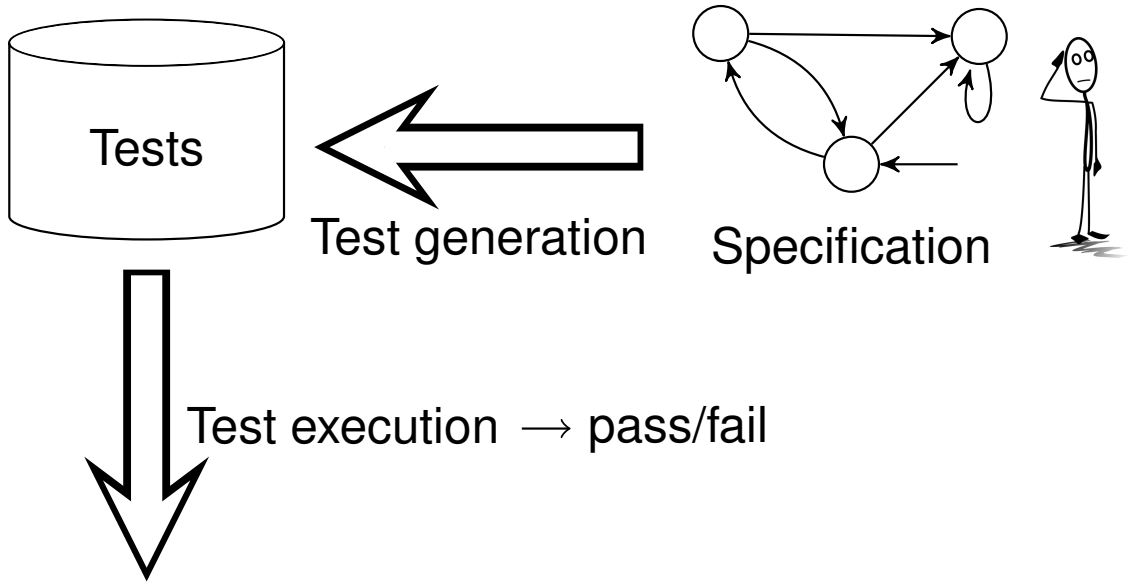
Manual testing

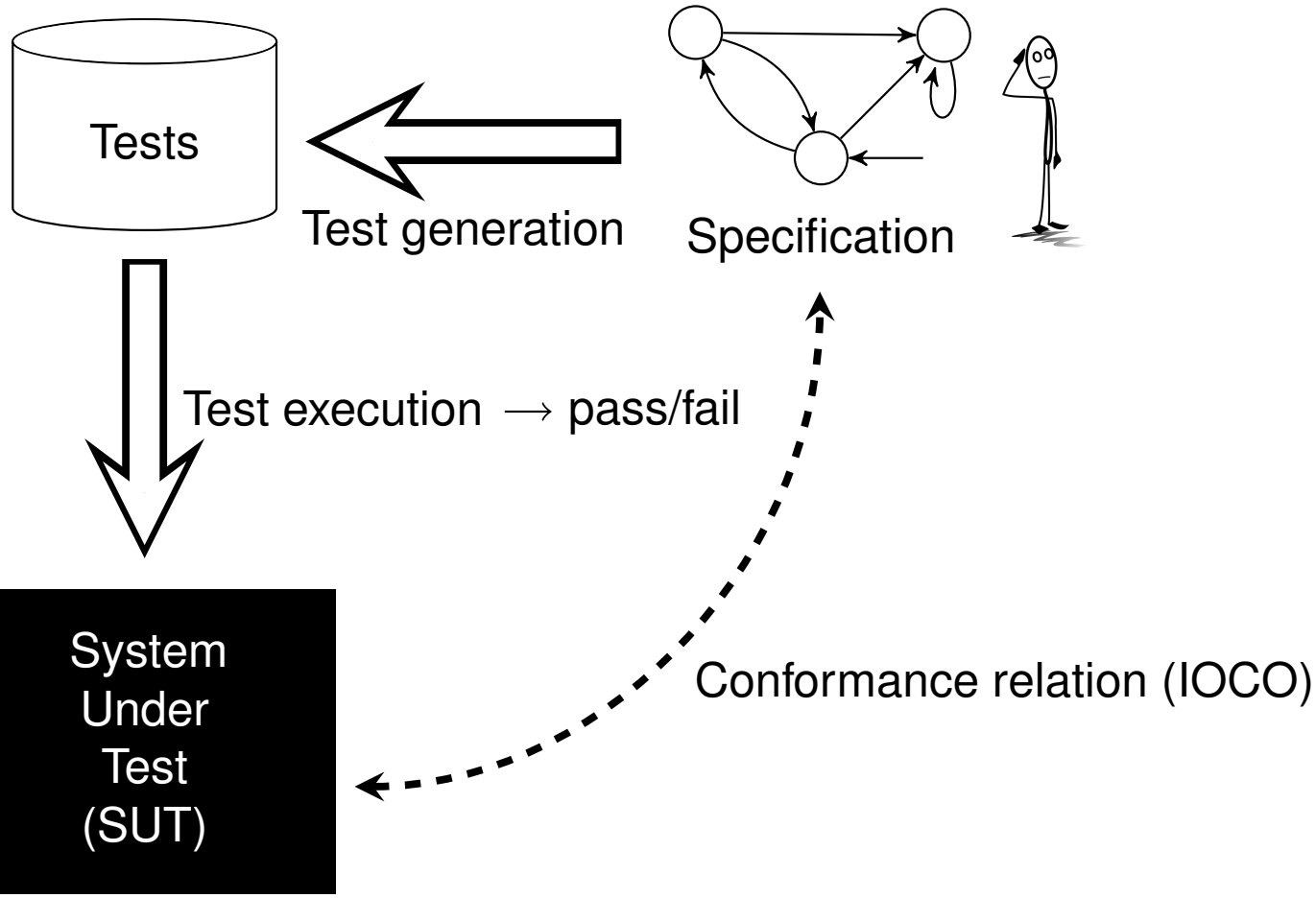


Automated testing



Model-Based Testing




Tests
Test generation
Specification
Test execution → pass/fail
System Under Test (SUT)
Conformance relation (IOCO)

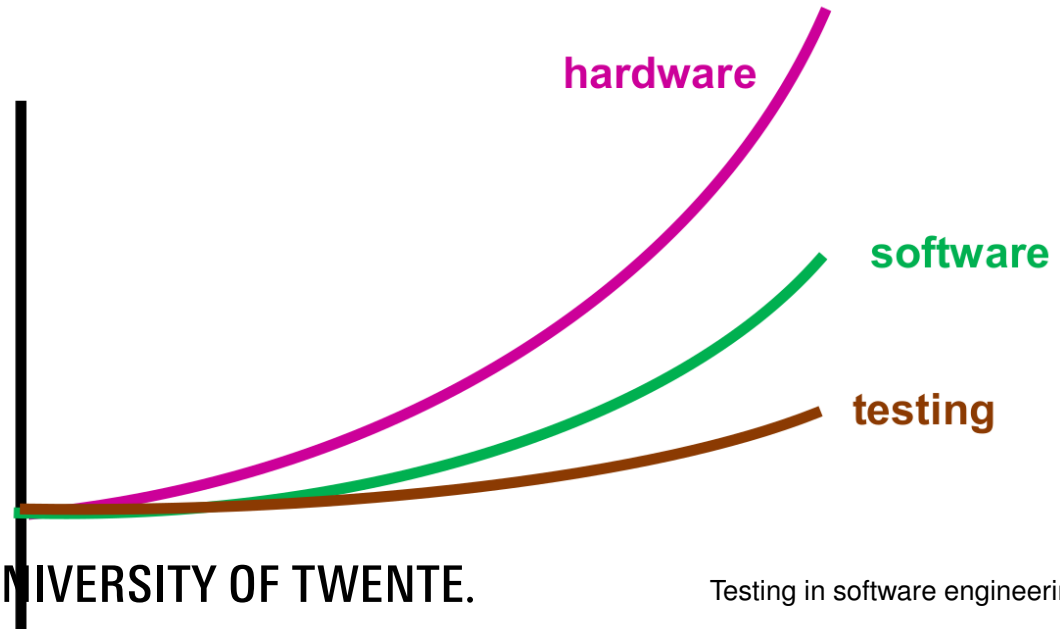
Levels of automation in testing

- Manual testing
- Automated testing:
  - Recorded tests
  - Scripted testing
  - Keyword-driven testing
  - Testing as part of Behaviour-Driven Development
- Model-based testing

Levels of automation in testing

- Manual testing
- Automated testing:
  - Recorded tests
  - Scripted testing
  - Keyword-driven testing
  - Testing as part of Behaviour-Driven Development
- Model-based testing

Efficiency and progress in development:



Quote by Dijkstra

“Testing can only detect the presence of errors, not their absence”

Regression testing and Test driven development

Regression testing

- Regression testing = test after every code change
  - Early detected bugs are easier to fix
- Regression = unexpected behaviour after change in code

Regression testing

- Regression testing = test after every code change
  - Early detected bugs are easier to fix
- Regression = unexpected behaviour after change in code
- Advantages:
  - Make behaviour of code observable
  - Monitor functioning of your software system
  - Detect introduced errors early
  - Manual testing is more effort

Regression testing

- Regression testing = test after every code change
  - Early detected bugs are easier to fix
- Regression = unexpected behaviour after change in code
- Advantages:
  - Make behaviour of code observable
  - Monitor functioning of your software system
  - Detect introduced errors early
  - Manual testing is more effort
- Q: What is the function of regression testing in development?
  - After implementing a new feature:

Regression testing

- Regression testing = test after every code change
  - Early detected bugs are easier to fix
- Regression = unexpected behaviour after change in code
- Advantages:
  - Make behaviour of code observable
  - Monitor functioning of your software system
  - Detect introduced errors early
  - Manual testing is more effort
- Q: What is the function of regression testing in development?
  - After implementing a new feature:
    - Check that other behaviour stays the same
  - After refactoring:

Regression testing

- Regression testing = test after every code change
  - Early detected bugs are easier to fix
- Regression = unexpected behaviour after change in code
- Advantages:
  - Make behaviour of code observable
  - Monitor functioning of your software system
  - Detect introduced errors early
  - Manual testing is more effort
- Q: What is the function of regression testing in development?
  - After implementing a new feature:
    - Check that other behaviour stays the same
  - After refactoring:
    - Check that nothing has changed

How to do regression testing: test automation

- Use a framework e.g. JUnit
  - Tests are methods annotated with `@Test`
  - Use `assert` methods, e.g. `assertEquals` to check result of test

How to do regression testing: test automation

- Use a framework e.g. JUnit
  - Tests are methods annotated with `@Test`
  - Use `assert` methods, e.g. `assertEquals` to check result of test
- Integrate testing in continuous integration pipeline:
  - All tests executed automatically after any push to git repo
  - Don't merge code with failing tests
  - Prevent bugs bugging other developers

Test-driven development (TDD)

Test before writing code

Test-driven development (TDD)

Test before writing code

Procedure:

1. Write test for new functionality
2. Run test
3. Check result: test should fail
   - Pass? Go back to step 1
4. Implement to obtain functionality
5. Run all tests: check if no regression
6. Check result:
   - Pass? Done!
   - Fail? Go back to step 4 (or adapt test and go to step 5)

Test-driven development (TDD)

Test before writing code

Procedure:

1. Write test for new functionality
2. Run test
3. Check result: test should fail
   - Pass? Go back to step 1
4. Implement to obtain functionality
5. Run all tests: check if no regression
6. Check result:
   - Pass? Done!
   - Fail? Go back to step 4 (or adapt test and go to step 5)

Q: Which steps done as tester/developer?

Three laws of Test Driven development

1. No code before failing test failing unit test is written
2. No more tests than sufficient to fail to fail
3. No more code than sufficient to pass tests

Some challenges in TDD[1]

- TDD is a discipline
- TDD needs some design details (for test interface)
- Expected result of test might be unknown

[1] Paper “A Literature Review on the Challenges of Applying Test-Driven Development in Software Engineering” by Daniel Staegemann et al.; published in *Complex Systems Informatics and Modeling Quarterly* journal, 2022.

Some advantages of TDD

- TDD is a code **design** strategy
  - What before How

Some advantages of TDD

- TDD is a code **design** strategy
  - What before How
  - Actively look for issues instead of confirming (happy flow) expectations

Some advantages of TDD

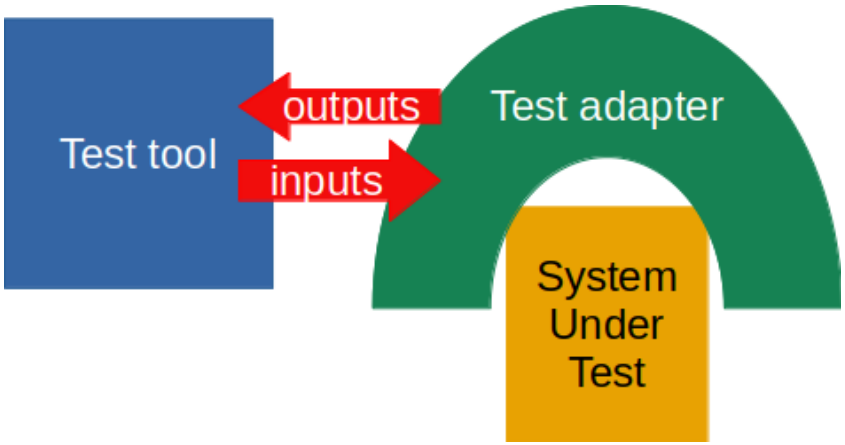
- TDD is a code **design** strategy
  - What before How
  - Actively look for issues instead of confirming (happy flow) expectations
  - Increased quality by small incremental changes

Some advantages of TDD

- TDD is a code **design** strategy
  - What before How
  - Actively look for issues instead of confirming (happy flow) expectations
  - Increased quality by small incremental changes
- TDD traditionally applied for unit tests
  - Also possible for integration and system tests

System testing with Behaviour-Driven Development

System testing



What does a system test test?

Functionality testing:

- Test functionality/behaviour/feature at system level
- Test interaction between user and system
- Black-box: system inputs and outputs

What does a system test test?

Functionality testing:

- Test functionality/behaviour/feature at system level
- Test interaction between user and system
- Black-box: system inputs and outputs

How to formulate functionality:

- Feature/Requirement description
- User story
- Behaviour-Driven Development scenario
- Model
- Property
- ...

User story

- Format:
  As a ⟨type of user⟩
  I want ⟨some goal⟩
  so that ⟨reason⟩ (optional)
- Example:
  As a student,
  I want to search for the contact details of a UTwente employee[2]

[2] `https://people.utwente.nl/`

What is Behaviour-Driven Development?

- BDD: not Binary Decision Diagram (risk assessment)

What is Behaviour-Driven Development?

- BDD: not Binary Decision Diagram (risk assessment)
- Extends agile
- Facilitates:
  - collaboration between different roles
  - shared understanding on what's to be implemented

Example BDD scenario

Scenario: Login to see email
Given a user has an email account
When the user logs in with valid credentials
Then the system displays the 20 most recent emails of the user

BDD scenarios

- A BDD scenario is an instance of required software behaviour
  - specification by example
  - a behaviour specifies user interaction with the system

BDD scenarios

- A BDD scenario is an instance of required software behaviour
  - specification by example
  - a behaviour specifies user interaction with the system
- In structured natural language, e.g.
  - Given-When-Then style of Gherkin notation

BDD scenarios

- A BDD scenario is an instance of required software behaviour
  - specification by example
  - a behaviour specifies user interaction with the system
- In structured natural language, e.g.
  - Given-When-Then style of Gherkin notation
- Use BDD to specify and test essential use cases of your system

Steps in Behaviour-Driven Development

1. Start from agile approach:
   - choose feature/user story/requirement from sprint

Steps in Behaviour-Driven Development

1. Start from agile approach:
   - choose feature/user story/requirement from sprint
2. Discovery: Brainstorm and create concrete examples
   - Ensures communication and shared understanding

Steps in Behaviour-Driven Development

1. Start from agile approach:
   - choose feature/user story/requirement from sprint
2. Discovery: Brainstorm and create concrete examples
   - Ensures communication and shared understanding
3. Formulation: Write scenarios in Given-Then-When-Style
   - Possibly multiple scenarios in a feature

Steps in Behaviour-Driven Development

1. Start from agile approach:
   - choose feature/user story/requirement from sprint
2. Discovery: Brainstorm and create concrete examples
   - Ensures communication and shared understanding
3. Formulation: Write scenarios in Given-Then-When-Style
   - Possibly multiple scenarios in a feature
4. Automate: create tests from scenarios
   - Living documentation

Q: BDD = TDD?

Shared understanding in BDD

- Collaboration between ‘three amigos’:
  1. Product owner:
  2. Developer:
  3. Tester:

Shared understanding in BDD

- Collaboration between ‘three amigos’:
  1. Product owner:
  2. Developer:
  3. Tester:

  Q: goal in BDD process? contribution to BDD process?

Shared understanding in BDD

- Collaboration between ‘three amigos’:
  1. Product owner:
     - goal: achieve business goal
     - contribution: sees what the product should be for user
  2. Developer:
  3. Tester:

Q: goal in BDD process? contribution to BDD process?

Shared understanding in BDD

- Collaboration between ‘three amigos’:
  1. Product owner:
     - goal: achieve business goal
     - contribution: sees what the product should be for user
  2. Developer:
     - goal: achieve implementation of product
     - contribution: sees technical possibilities and obstacles
  3. Tester:

Q: goal in BDD process? contribution to BDD process?

Shared understanding in BDD

- Collaboration between ‘three amigos’:
  1. Product owner:
     - goal: achieve business goal
     - contribution: sees what the product should be for user
  2. Developer:
     - goal: achieve implementation of product
     - contribution: sees technical possibilities and obstacles
  3. Tester:
     - goal: achieve testability of product
     - contribution: sees edge cases

Q: goal in BDD process? contribution to BDD process?

Behaviour Driven Development scenario

- Given-Then-When style format:

  Given ⟨ context ⟩
  When ⟨ action(s) ⟩
  Then ⟨ outcome ⟩

- More specific format:

  Given ⟨ precondition ⟩
  When ⟨ action(s) ⟩
  Then ⟨ postcondition or resulting action ⟩

Example BDD scenario

Scenario: Login to see email
Given a user has an email account
When the user logs in with valid credentials
Then the system displays the 20 most recent emails of the user

Example BDD scenario with parameters[3]

Scenario Outline: Basic movement
Given a player has loaded the level “move”
When the player presses ⟨key⟩
Then the player moves ⟨direction⟩

Scenarios:

key	direction
w	forward
a	left
s	backward
d	right

[3] from master thesis Michael Mulder

From BDD scenario to test case

- Use tool to generate bindings for scenario steps
- i.e. step definitions

From BDD scenario to test case

- Use tool to generate bindings for scenario steps
- i.e. step definitions
- Example:
  - Step: *Given* a player has loaded the level “move”
  - Would generate a method header `givenLoadedLevel(String levelName)` as binding

From BDD scenario to test case

- Use tool to generate bindings for scenario steps
- i.e. step definitions
- Example:
  - Step: *Given* a player has loaded the level “move”
  - Would generate a method header `givenLoadedLevel(String levelName)` as binding
- Implement bindings
- Use tool to generate tests
- Test execution shows: scenario steps and calls to bindings

BDD research[4]

- Challenges in BDD for testing
  - How to bring system to Given step
  - Implementing bindings depends on ambiguous natural language formulation
  - How to check then step (observe or inspect system state or variables)

[4] with PhD student Tannaz Zameni

BDD research[4]

- Challenges in BDD for testing
  - How to bring system to Given step
  - Implementing bindings depends on ambiguous natural language formulation
  - How to check then step (observe or inspect system state or variables)
- From BDD to Model-based testing:
  - Guidelines for BDD scenario format
  - Translate BDD scenarios to MBT models
  - Compose models
  - Use model-based test generation algorithms

[4] with PhD student Tannaz Zameni

Looking for a topic for your internship/thesis?

- Translate BDD scenarios (in natural language!) to MBT models
- Use LLMs

Looking for a topic for your internship/thesis?

- Translate BDD scenarios (in natural language!) to MBT models
- Use LLMs
- Interested?
  - Talk to me in break
  - Send me an email

Software Testing project

Project organization

- In groups of 4 or 5. No groups ≤ 3.
- You may change your current team, only this week!
- Project deadline: Friday April 11 at 23:59

SUT: NewsFeed

- NewsFeed application from Axini

SUT: NewsFeed

- NewsFeed application from Axini
- Included:
  - A jar to run the application

SUT: NewsFeed

- NewsFeed application from Axini
- Included:
  - A jar to run the application
    - Several implementations from manufacturers:
    - Choose ‘Inixa’ for finding bugs
    - Choose ’Axini’ to debug your tests without bugs from application

SUT: NewsFeed

- NewsFeed application from Axini
- Included:
  - A jar to run the application
    - Several implementations from manufacturers:
    - Choose ‘Inixa’ for finding bugs
    - Choose ’Axini’ to debug your tests without bugs from application
  - Specification on expected functionality
    - Subscribe to news topics, receive news headlines

SUT: NewsFeed

- NewsFeed application from Axini
- Included:
  - A jar to run the application
    - Several implementations from manufacturers:
    - Choose ‘Inixa’ for finding bugs
    - Choose ’Axini’ to debug your tests without bugs from application
  - Specification on expected functionality
    - Subscribe to news topics, receive news headlines
  - README on interface with JSON messages

SUT: NewsFeed

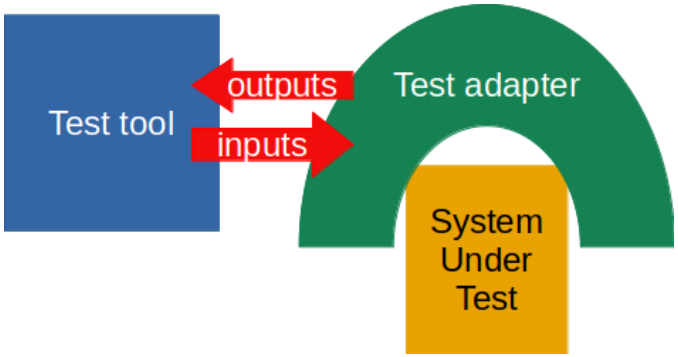
- NewsFeed application from Axini
- Included:
  - A jar to run the application
    - Several implementations from manufacturers:
    - Choose ‘Inixa’ for finding bugs
    - Choose ’Axini’ to debug your tests without bugs from application
  - Specification on expected functionality
    - Subscribe to news topics, receive news headlines
  - README on interface with JSON messages
- Find bugs w.r.t. specification

SUT: NewsFeed

- NewsFeed application from Axini
- Included:
  - A jar to run the application
    - Several implementations from manufacturers:
    - Choose ‘Inixa’ for finding bugs
    - Choose ’Axini’ to debug your tests without bugs from application
  - Specification on expected functionality
    - Subscribe to news topics, receive news headlines
  - README on interface with JSON messages
- Find bugs w.r.t. specification
- Report bugs in interface to `p.vandenbos@utwente.nl`

Project tasks

- Set up testing environment
- Use BDD tool with BDD scenarios
- Use tool for automated tests
- Generate tests from a model using AMP (Axini)
- Find bugs
- Compare tools and test results



Project deliverables

- Report: describe test environment, explain tests, models, and test results, give comparison

Project deliverables

- Report: describe test environment, explain tests, models, and test results, give comparison
- Software artefact: Tests, models, scripts, test results ...

Project deliverables

- Report: describe test environment, explain tests, models, and test results, give comparison
- Software artefact: Tests, models, scripts, test results ...
- Plagiarism:
  - Write your own text
  - Provide references for re-used code, libraries, tools in your report
  - Do not use text or code from Generative AI tools

On Canvas

- Project description with details of deliverables and tasks
- NewsFeed application including its specification and README

Grading

- Software testing project counts for 25% of STAR grade

Grading

- Software testing project counts for 25% of STAR grade
- Smartdoor assignment counts for 20% of the software testing project

Classical black- and white-box testing

Classical test design techniques

- Black box:
  - Error guessing
  - Equivalence partitioning
  - Boundary value analysis
  - Random testing

Classical test design techniques

- Black box:
  - Error guessing
  - Equivalence partitioning
  - Boundary value analysis
  - Random testing
- White-box:
  - Line coverage
  - Statement coverage
  - Branch coverage (a.k.a. decision coverage)
  - ...
  - Path coverage

Error guessing

- Just ‘guess’ where the errors are . . .
- Rely on intuition and experience of tester

Error guessing

- Just ‘guess’ where the errors are . . .
- Rely on intuition and experience of tester
- Strategy:
  - Make a list of possible errors or error-prone situations (e.g. edge cases)
  - Write test cases based on this list

Error guessing

- Just ‘guess’ where the errors are . . .
- Rely on intuition and experience of tester
- Strategy:
  - Make a list of possible errors or error-prone situations (e.g. edge cases)
  - Write test cases based on this list
- Or apply risk analysis!
- Try to identify critical parts of program:
  - Parts with unclear requirements
  - Developed by unexperienced programmer
  - Complex code according to software metrics

Error guessing

- Just ‘guess’ where the errors are . . .
- Rely on intuition and experience of tester
- Strategy:
  - Make a list of possible errors or error-prone situations (e.g. edge cases)
  - Write test cases based on this list
- Or apply risk analysis!
- Try to identify critical parts of program:
  - Parts with unclear requirements
  - Developed by unexperienced programmer
  - Complex code according to software metrics
- Write more and more thorough tests for high-risk code

Error guessing

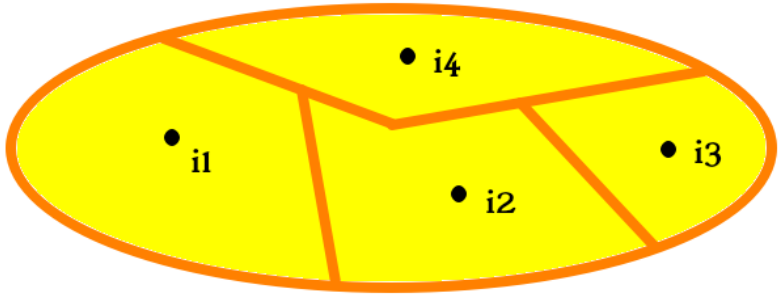
- Just ‘guess’ where the errors are . . .
- Rely on intuition and experience of tester
- Strategy:
  - Make a list of possible errors or error-prone situations (e.g. edge cases)
  - Write test cases based on this list
- Or apply risk analysis!
- Try to identify critical parts of program:
  - Parts with unclear requirements
  - Developed by unexperienced programmer
  - Complex code according to software metrics
- Write more and more thorough tests for high-risk code
  - Or first refactor/re-develop this code

Equivalence partitioning

- Divide all possible inputs into a finite number of equivalence classes (partitions)
  - e.g. for a function `compute(int x)`

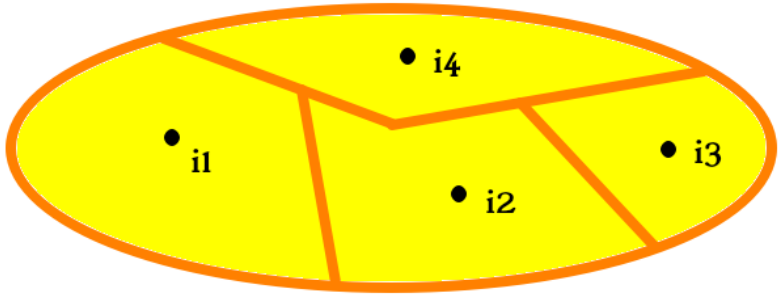
Equivalence partitioning

- Divide all possible inputs into a finite number of equivalence classes (partitions)
  - e.g. for a function `compute(int x)`
- Pick an (arbitrary) input (representative) for each equivalence class



Equivalence partitioning

- Divide all possible inputs into a finite number of equivalence classes (partitions)
  - e.g. for a function `compute(int x)`
- Pick an (arbitrary) input (representative) for each equivalence class



- Choose classes such that you may reasonably assume that:
  - The function behaves analogously for inputs in the same classo
  - One test with one input from each class is sufficient
  - If the representative causes a fault then other would too

Equivalence partitioning example

- Test a program that:
  - computes the sum of the first `N` integers
  - If N is negative, it takes the absolute value `|N|`
  - as long as this sum is less than `max`
  - Otherwise an error should be reported.

Equivalence partitioning example

- Test a program that:
  - computes the sum of the first `N` integers
  - If N is negative, it takes the absolute value `|N|`
  - as long as this sum is less than `max`
  - Otherwise an error should be reported.
- Formally:
  - For any `N`, `max` $\in \mathbb{Z}$:

$$\texttt{sum(N, max)} = \begin{cases} \Sigma_{k=0}^{|\texttt{N}|} k & \text{if } (\Sigma_{k=0}^{|\texttt{N}|} k) <= \texttt{max} \\ \texttt{error} & \text{otherwise} \end{cases}$$

Equivalence partitioning example

- For any $\texttt{N}, \texttt{max} \in \mathbb{Z}$: $\texttt{sum(N, max)} = \begin{cases} \Sigma_{k=0}^{|\texttt{N}|} k & \text{if } (\Sigma_{k=0}^{|\texttt{N}|} k) <= \texttt{max} \\ \texttt{error} & \text{otherwise} \end{cases}$
- Some equivalence classes for the input parameters:

Equivalence partitioning example

- For any $\texttt{N}, \texttt{max} \in \mathbb{Z}$: $\texttt{sum(N, max)} = \begin{cases} \Sigma_{k=0}^{|\texttt{N}|} k & \text{if } (\Sigma_{k=0}^{|\texttt{N}|} k) <= \texttt{max} \\ \texttt{error} & \text{otherwise} \end{cases}$
- Some equivalence classes for the input parameters:
  (a) Provide values for both
  (b) Too few or too many values
  (c) Integers
  (d) A non-integer
  (e) Positive values
  (f) Negative values
- Some equivalence classes for the function output:

Equivalence partitioning example

- For any $\texttt{N}, \texttt{max} \in \mathbb{Z}$: $\texttt{sum(N, max)} = \begin{cases} \Sigma_{k=0}^{|\texttt{N}|} k & \text{if } (\Sigma_{k=0}^{|\texttt{N}|} k) <= \texttt{max} \\ \texttt{error} & \text{otherwise} \end{cases}$
- Some equivalence classes for the input parameters:
  (a) Provide values for both
  (b) Too few or too many values
  (c) Integers
  (d) A non-integer
  (e) Positive values
  (f) Negative values
- Some equivalence classes for the function output:
  (g) Values for each of the cases

- Some test cases:

Equivalence partitioning example

- For any $\texttt{N}, \texttt{max} \in \mathbb{Z}$: $\texttt{sum(N, max)} = \begin{cases} \Sigma_{k=0}^{|\texttt{N}|} k & \text{if } (\Sigma_{k=0}^{|\texttt{N}|} k) <= \texttt{max} \\ \texttt{error} & \text{otherwise} \end{cases}$
- Some equivalence classes for the input parameters:
  (a) Provide values for both
  (b) Too few or too many values
  (c) Integers
  (d) A non-integer
  (e) Positive values
  (f) Negative values
- Some equivalence classes for the function output:
  (g) Values for each of the cases
- Some test cases:

max	N	output
100	10	55
100	-10	55
100	-100	error
10	10	error
10	-	error
”abc”	10	error

Boundary value analysis

- Test inputs on, or directly above or below class boundaries
- Also consider output boundaries
- Combine with arbitrary inputs from equivalence partitioning

Boundary value analysis example

- For any $\texttt{N}, \texttt{max} \in \mathbb{Z}$: $\texttt{sum(N, max)} = \begin{cases} \Sigma_{k=0}^{|\texttt{N}|} k & \text{if } (\Sigma_{k=0}^{|\texttt{N}|} k) <= \texttt{max} \\ \texttt{error} & \text{otherwise} \end{cases}$
- Some edge cases:

Boundary value analysis example

- For any $\texttt{N}, \texttt{max} \in \mathbb{Z}$: $\texttt{sum(N, max)} = \begin{cases} \Sigma_{k=0}^{|\texttt{N}|} k & \text{if } (\Sigma_{k=0}^{|\texttt{N}|} k) <= \texttt{max} \\ \texttt{error} & \text{otherwise} \end{cases}$
- Some edge cases:
  - Empty sum for `N = 0`
  - Absolute value for `N = 0`, or `N = -1`
  - A value for `max` such that `result` $=$ `max`, or `result` $=$ `max + 1`
  - Integer overflow boundary

Random testing

- Try out many random values: use random generators
  - Easy for numbers
  - More challenging for:
    - strings
    - lists
    - trees
    - complicated data structures
    - large data files, e.g. XML files
    - constrained data
    - ...

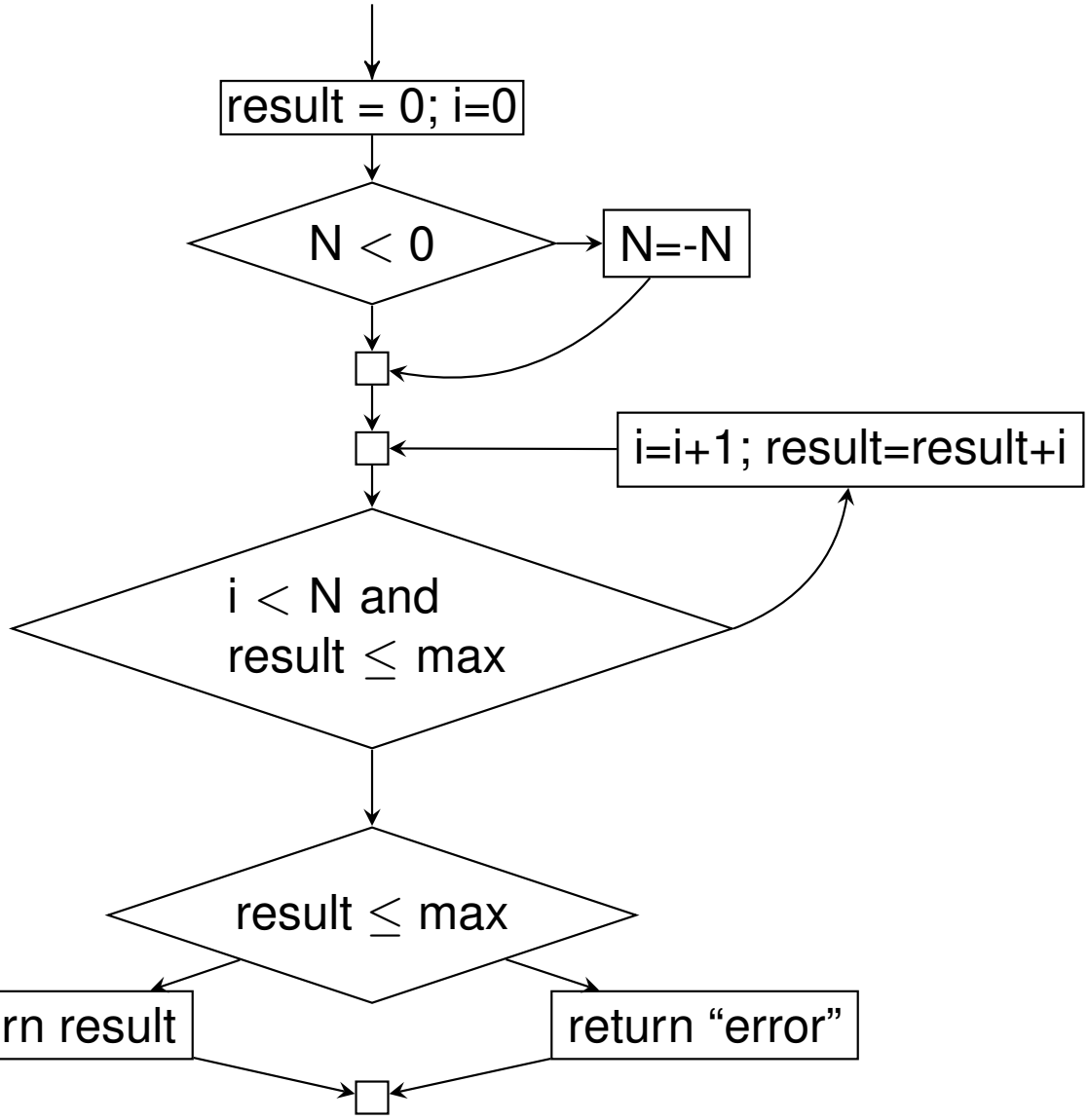
White-box testing

- Testing based on internals: structure of code
- Dependant on programming language
- Measure coverage of code by test cases

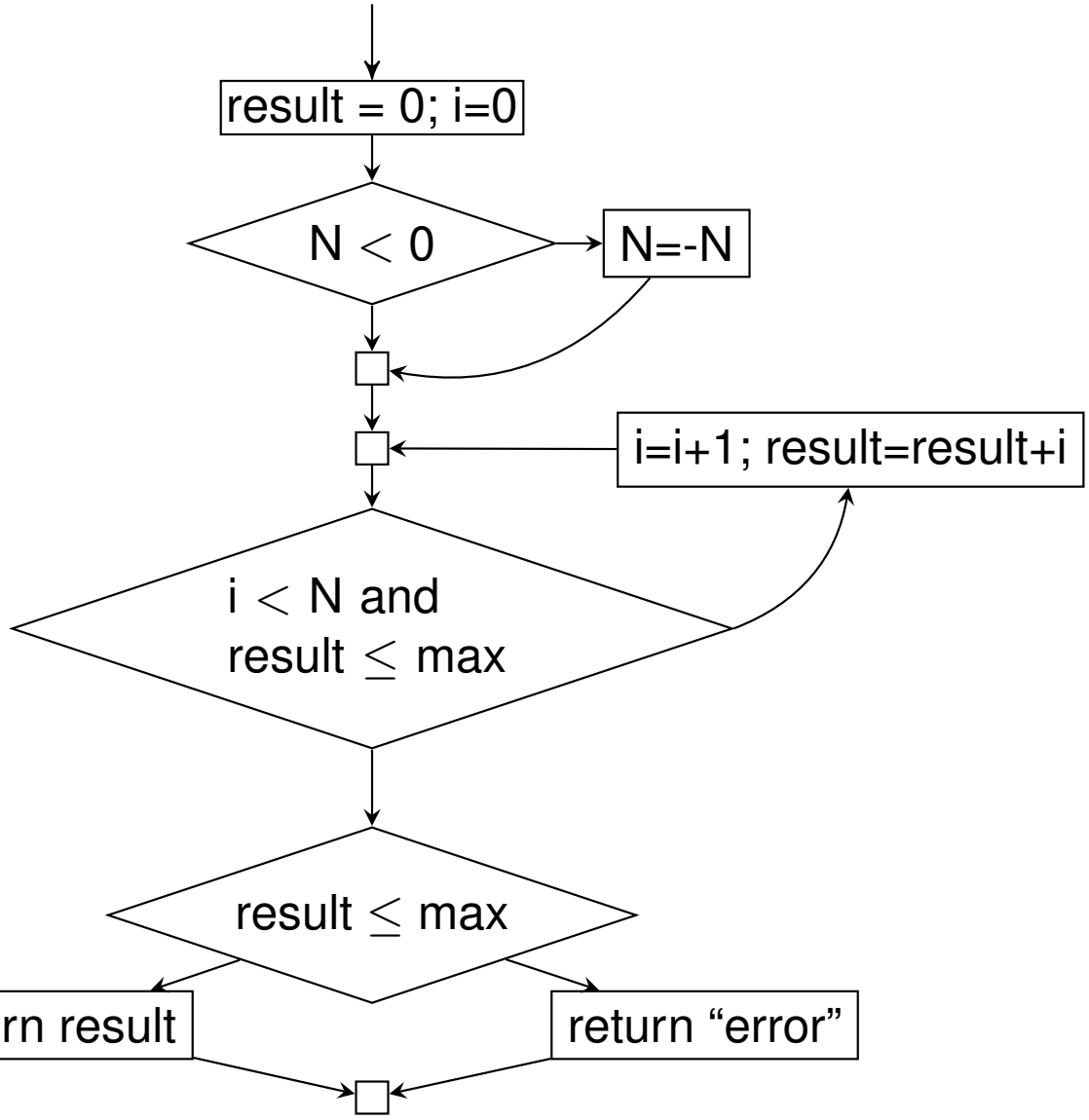
Example program (python)

```
def sum(N, maxnr):
    result = 0;
    i=0;
    if N < 0:
      N = –N;
    while i < N and result <= maxnr:
      i = i+1;
      result = result + i;
    if result <= maxnr:
      return result
    else:
      return ”error”
```

Flow diagram of example program

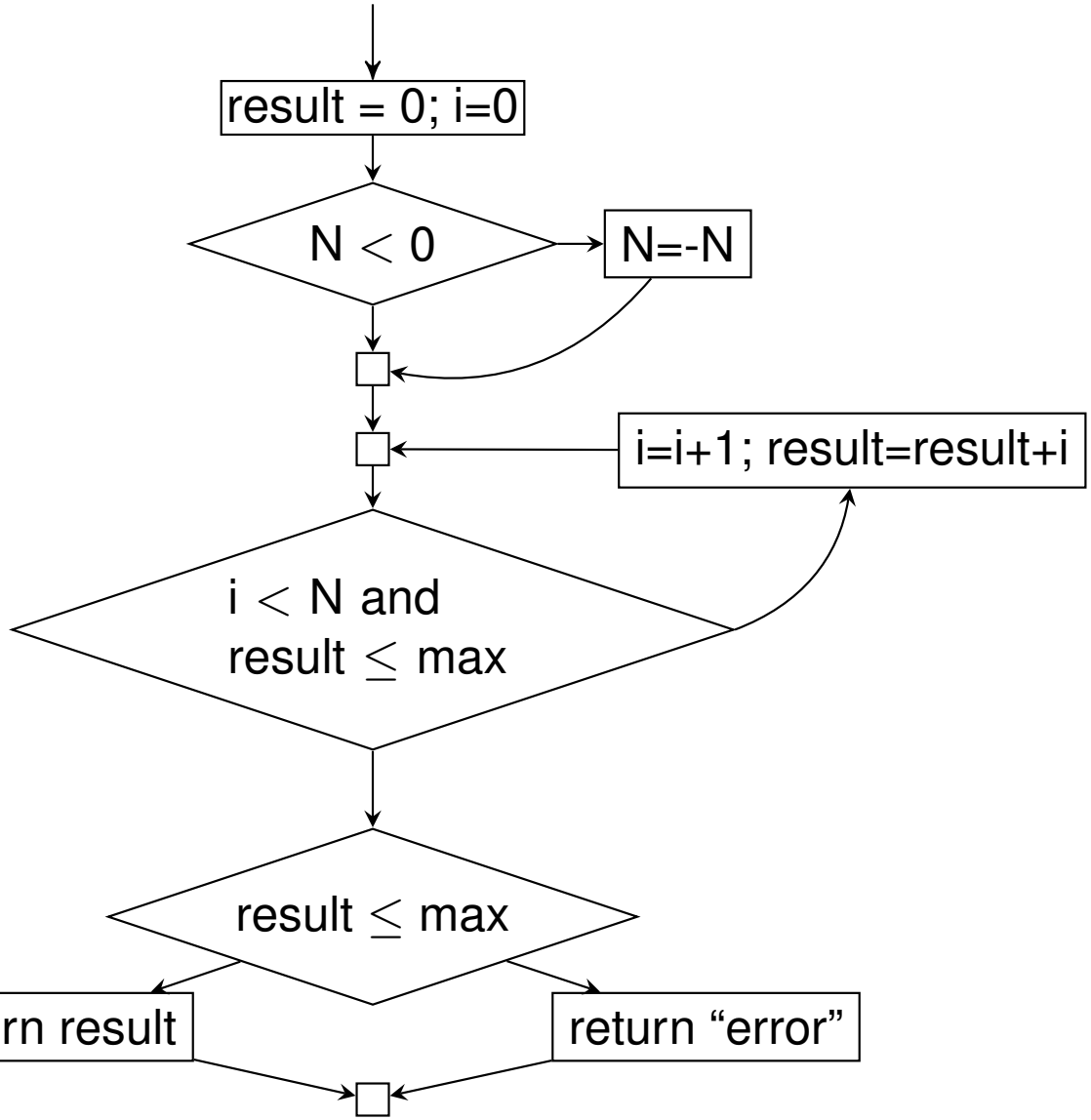


Flow diagram of example program



Statement coverage:

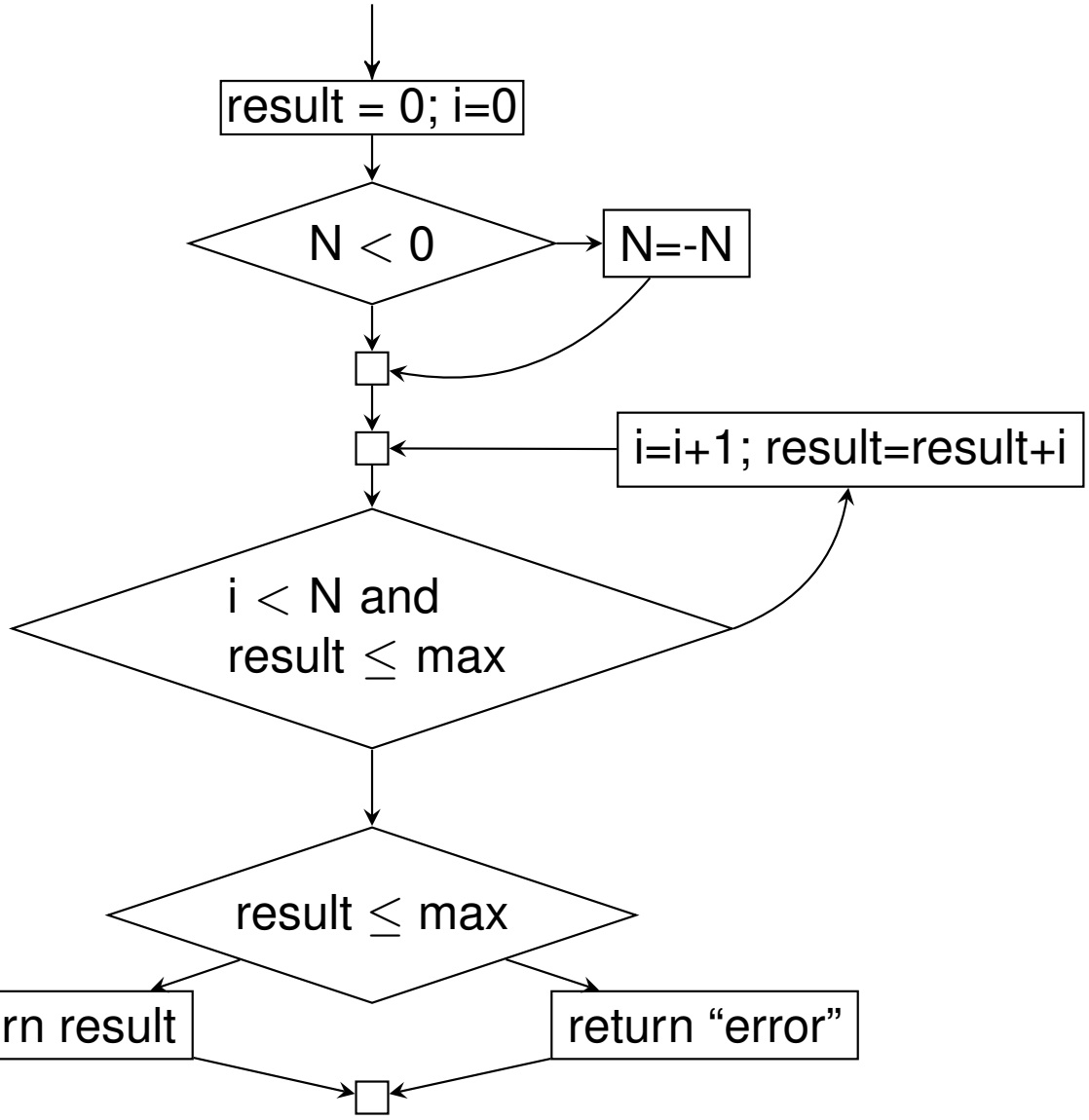
Flow diagram of example program



Statement coverage:

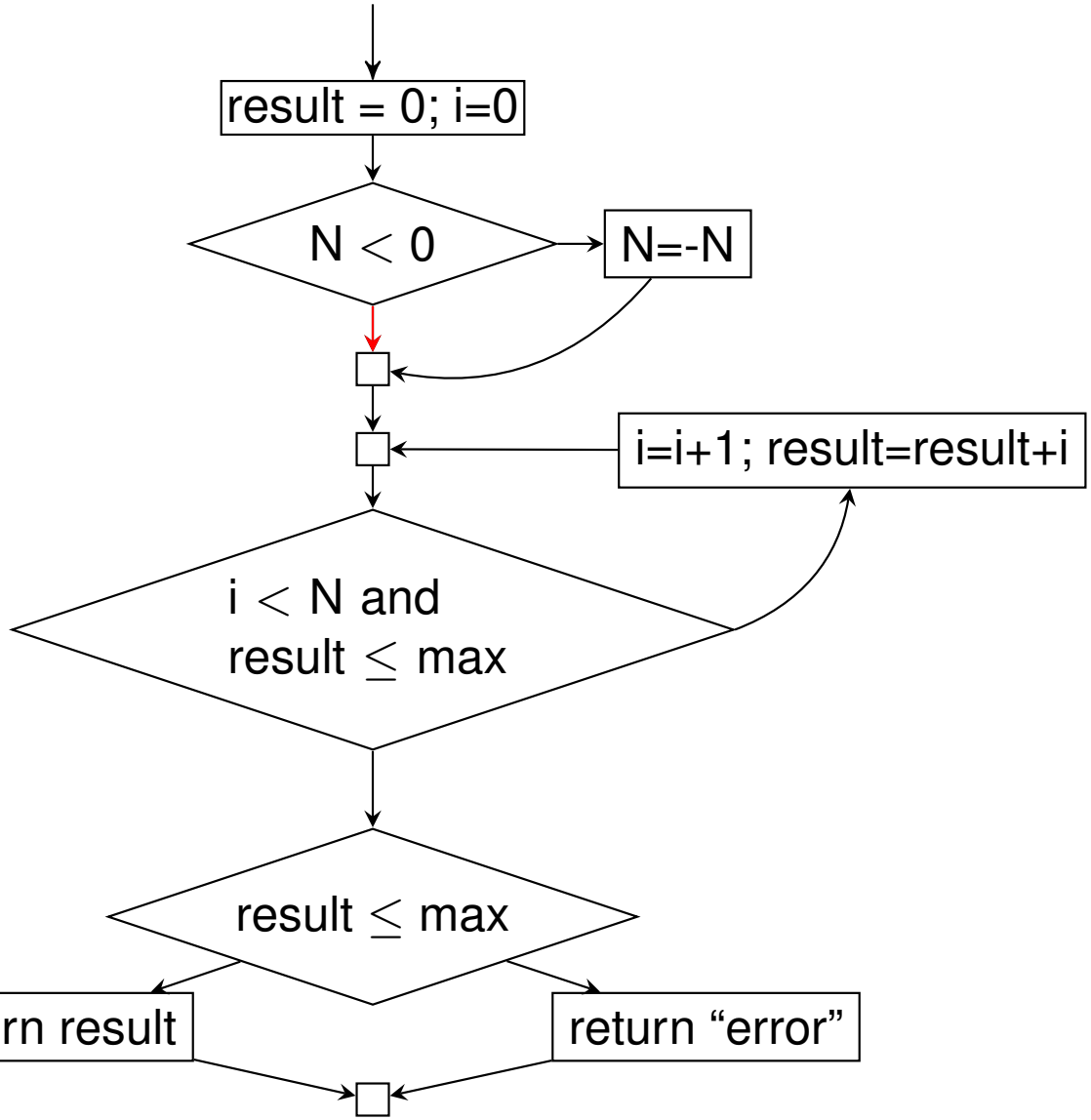
N	max
-1	0
-1	10

Flow diagram of example program



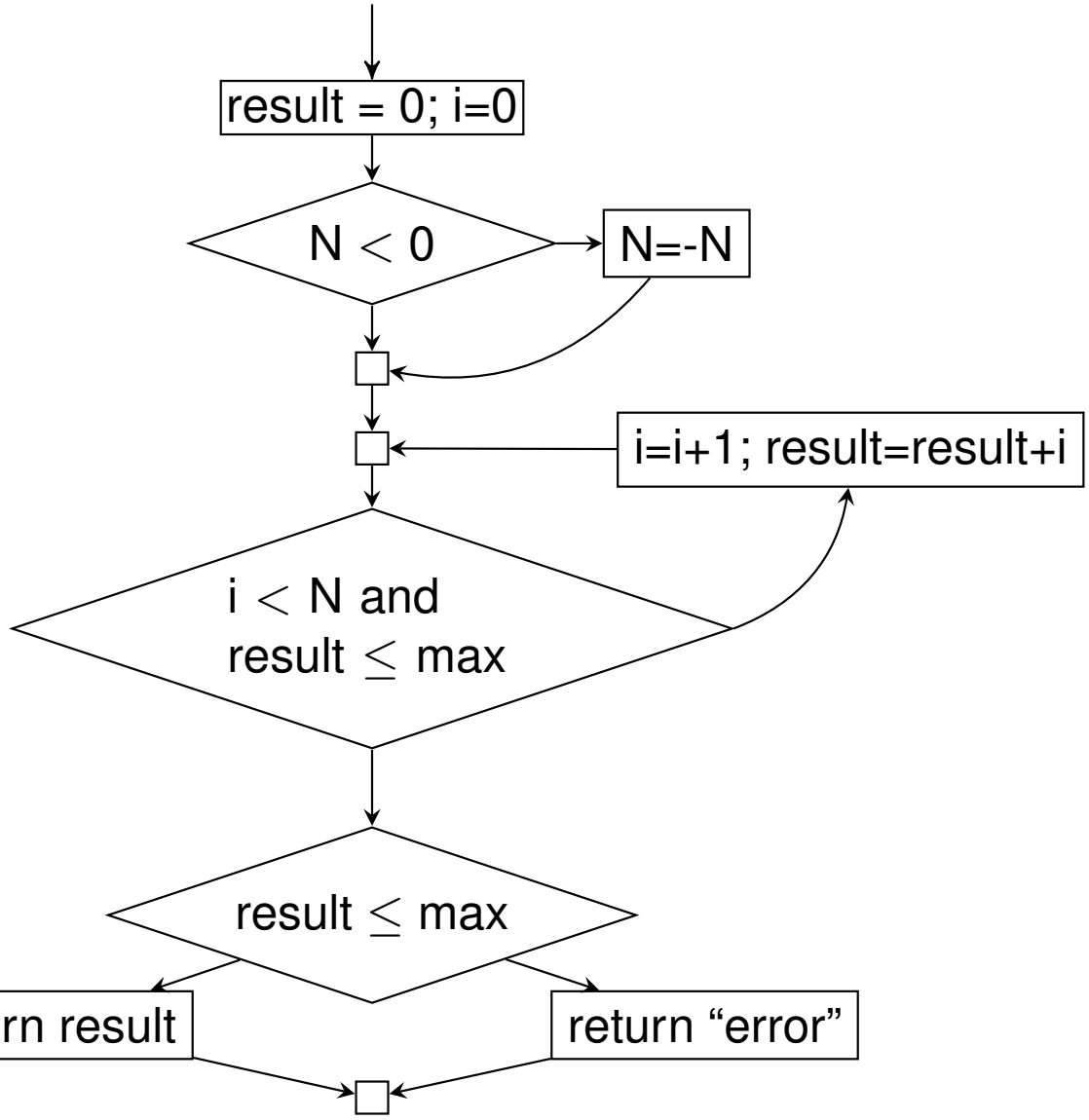
Branch (Decision) coverage:

Flow diagram of example program



Branch (Decision) coverage:

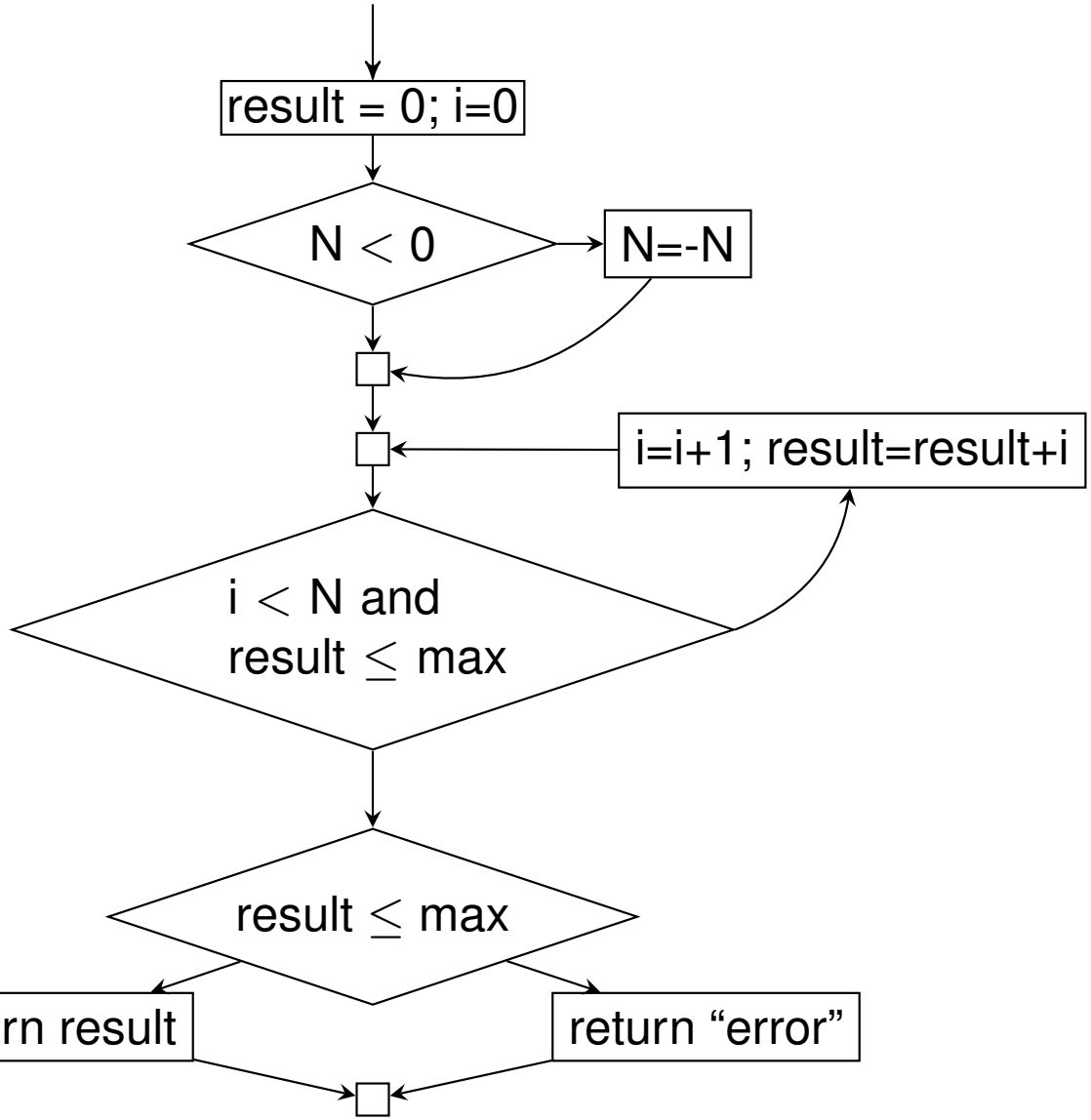
Flow diagram of example program



Branch (Decision) coverage:

N	max
-1	0
3	10

Flow diagram of example program



Path coverage: cover all paths...

Integration testing

Integration testing

- Test the integration of components
  - Component (in Java): a class or set of classes

Integration testing

- Test the integration of components
  - Component (in Java): a class or set of classes
  - Component interface (in Java): public methods used by other components

Integration testing

- Test the integration of components
  - Component (in Java): a class or set of classes
  - Component interface (in Java): public methods used by other components
  - Successful integration: component interface is used as expected by other component(s)

How to perform integration testing?

- Combine components incrementally
- Test each increment with integration tests

How to perform integration testing?

- Combine components incrementally
- Test each increment with integration tests
- Fix bugs of failing integration tests

How to perform integration testing?

- Combine components incrementally
- Test each increment with integration tests
- Fix bugs of failing integration tests
- Increment until all components have been combined

Test drivers and stubs

Test driver:

- A test program that calls the components to be tested
  - Provides input data
  - Returns test results

Test drivers and stubs

Test driver:

- A test program that calls the components to be tested
  - Provides input data
  - Returns test results

Test stub:

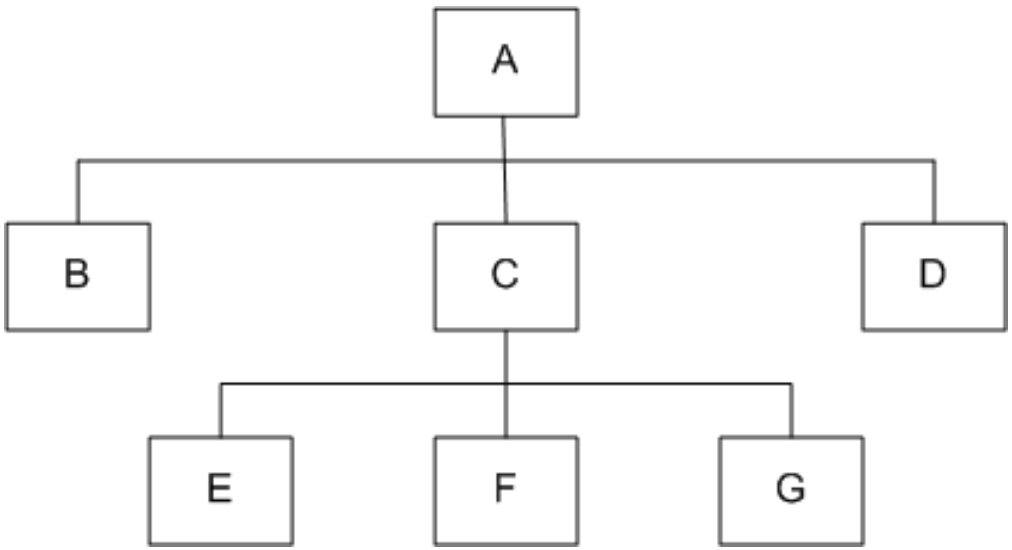
- Dummy program that receives calls from components
- It emulates a non-implemented component

Approaches to perform integration testing

- Top-down
- Bottom-up
- Sandwich
- Big-bang

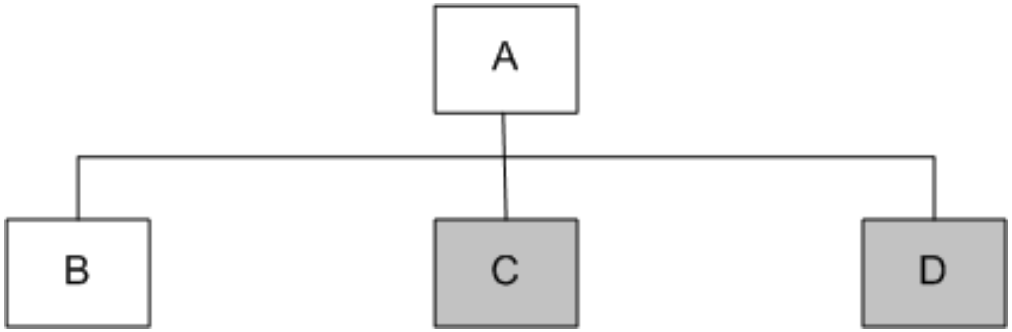
Approaches to perform integration testing

- Top-down
- Bottom-up
- Sandwich
- Big-bang



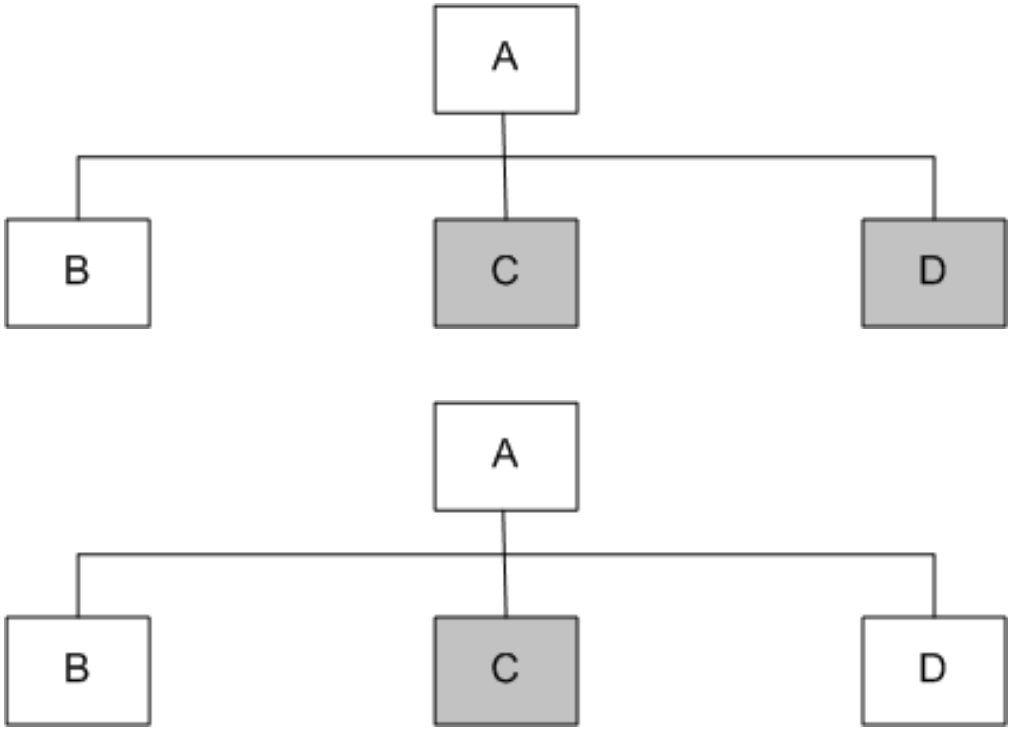
Top-down

Use stubs (gray):

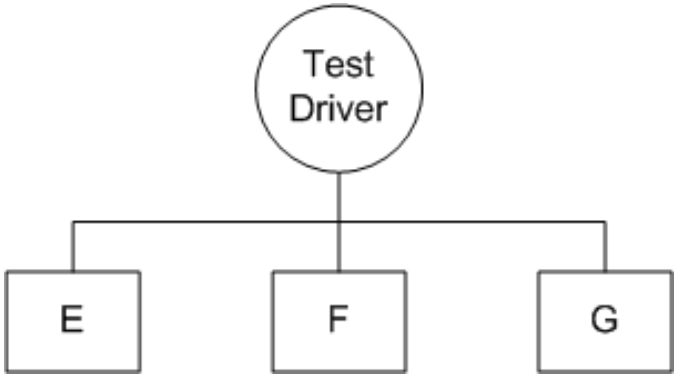


Top-down

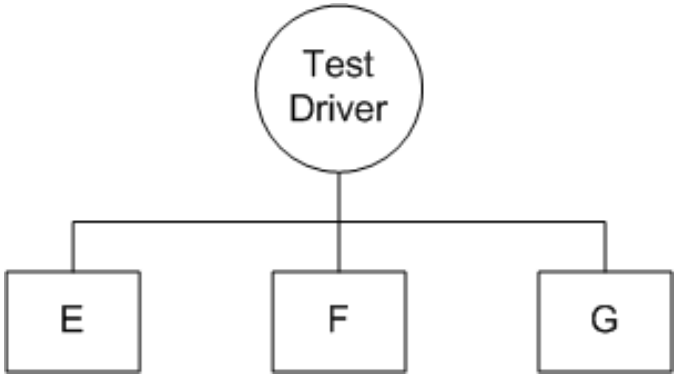
Use stubs (gray):

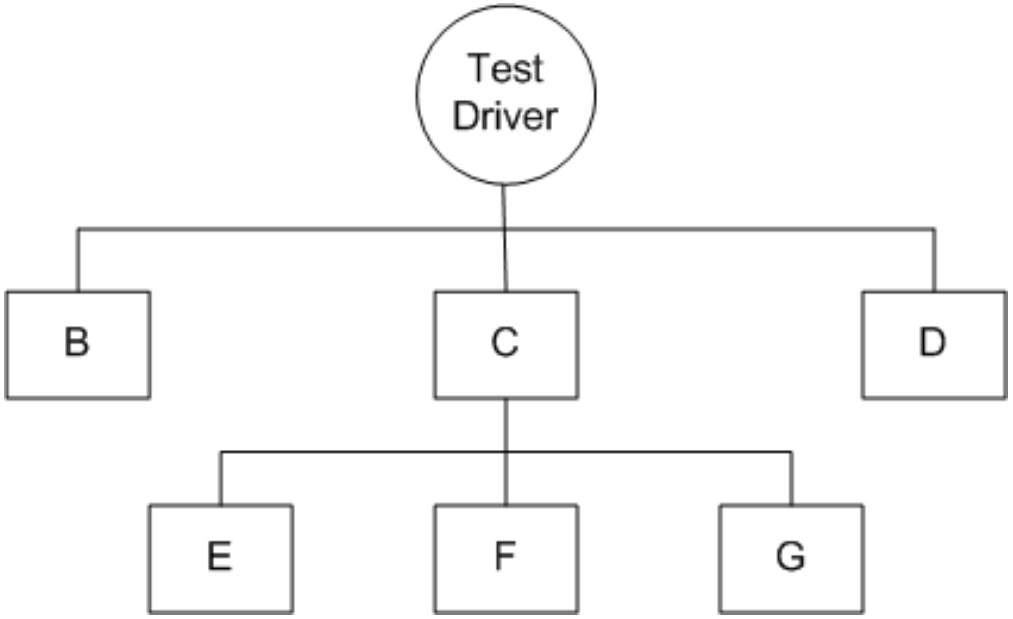


Bottom-up



Bottom-up





Sandwitch

- Use a mix of stubs and drivers

Big-bang

- Put all components together and test the whole system
- Essentially skips integration testing!

Testability of software

- Software design influences its testability
  - Especially for integration testing

Design for integration: Dependency injection

Dependencies are injected in the using class

- Intent: separating configuration from use

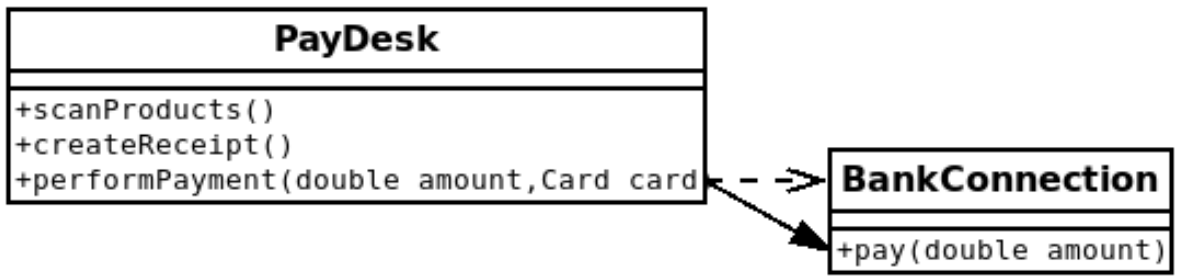
Design for integration: Dependency injection

Dependencies are injected in the using class

- Intent: separating configuration from use
- Apply to achieve:
  - Separation of concerns
  - Loose coupling
  - Flexibility needed for stubbing

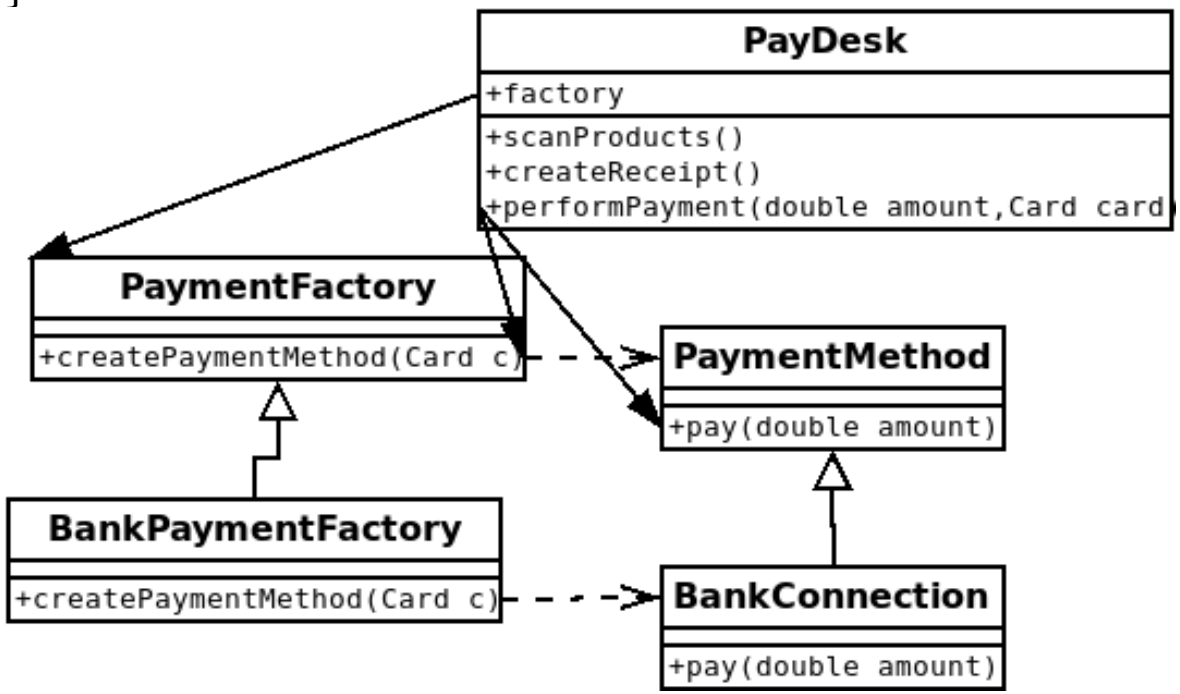
Design for integration testing: example

```
public void performPayment(double amount, Card card) {
    BankConnection connection = new BankConnection(card);
    connection.pay(amount);
}
```



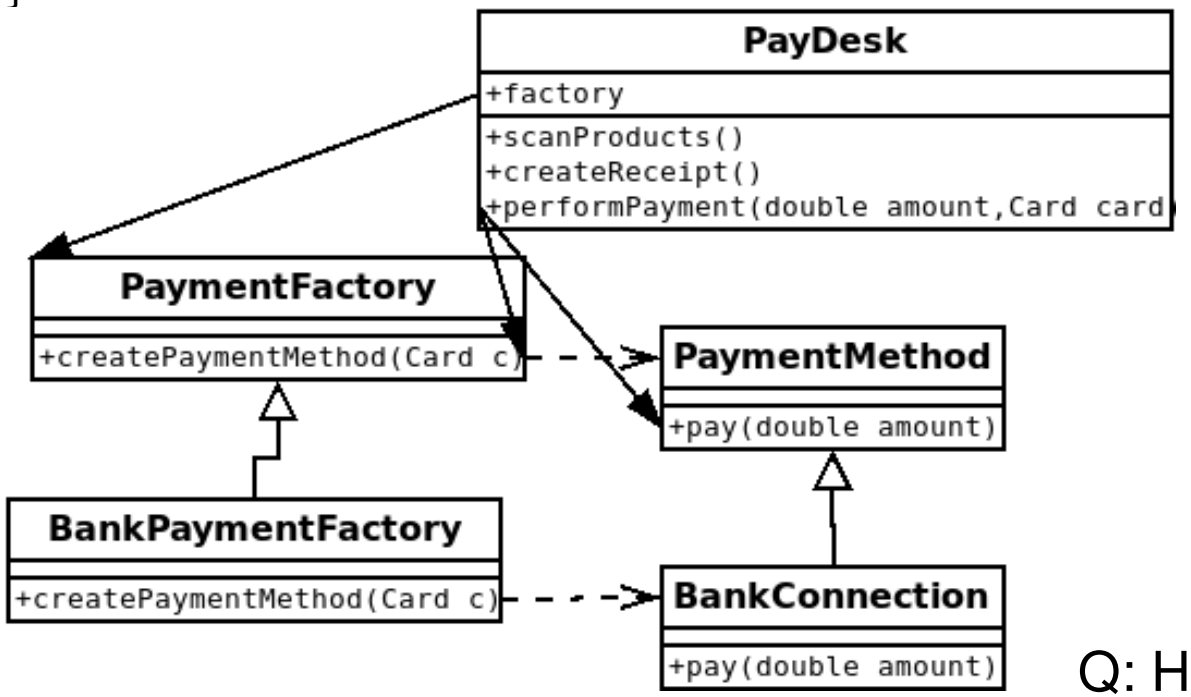
Design for integration testing: example

```
public PayDesk(PaymentFactory f) {
  factory = f;
}
public void performPayment(double amount, Card card) {
    PaymentMehod connection = factory.createPaymentMethod(card);
    connection.pay(amount);
}
```



Design for integration testing: example

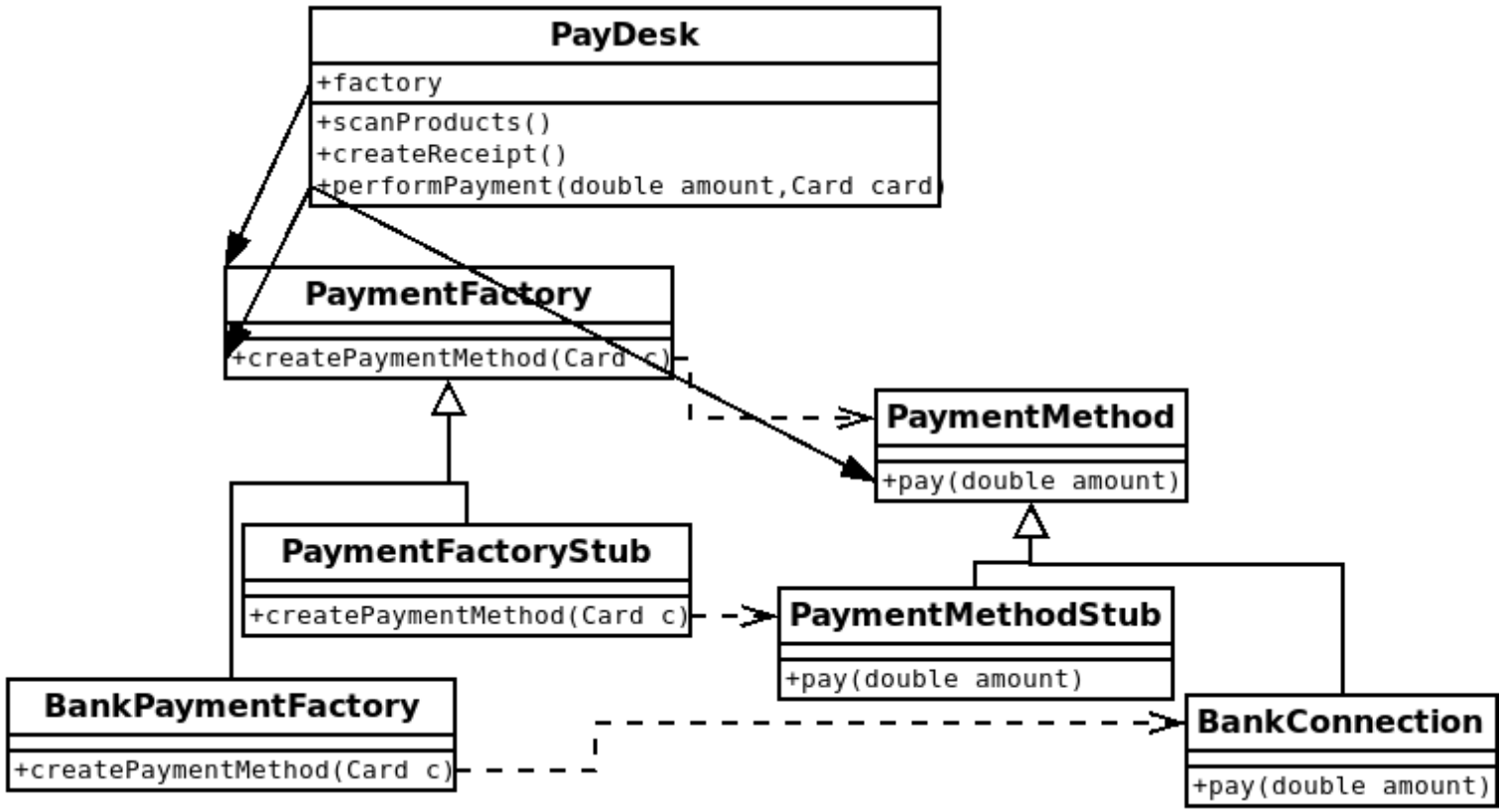
```
public PayDesk(PaymentFactory f) {
  factory = f;
}
public void performPayment(double amount, Card card) {
    PaymentMehod connection = factory.createPaymentMethod(card);
    connection.pay(amount);
}
```



Q: How to add stub?

Design for integration testing: example

```
PayDesk payDesk = new PayDesk(new PaymentFactoryStub());
```



Design for integration testing: example

```
class PayDesk {
    BankConnection connection;
    public PayDesk() {
        connection = new BankConnection();
    }
    public void performPayment(double amount, Card card) {
      connection.pay(amount, card);
    }
}
```

Design for integration testing: example

```
class PayDesk {
    BankConnection connection;
    public PayDesk() {
        connection = new BankConnection();
    }
    public void performPayment(double amount, Card card) {
      connection.pay(amount, card);
    }
}
```

Refactor into:

```
    PaymentFactory factory;
    public PayDesk(PaymentFactory f) {
        factory = f;
    }
    public void performPayment(double amount, Card card) {
      PaymentMethod connection = factory.createPaymentMethod(card);
      connection.pay(amount, card);
    }
```

When to apply factory method pattern for stubs

- Class A dynamically creates objects class B
  - These objects cannot be constructed in advance
- Class B needs to be replaced by a stub
- For simpler structure: remove factory classes and use `PaymentMethod` as argument for `PayDesk`

When to apply factory method pattern for stubs

- Class A dynamically creates objects class B
  - These objects cannot be constructed in advance
- Class B needs to be replaced by a stub
- For simpler structure: remove factory classes and use `PaymentMethod` as argument for `PayDesk`

Read more about dependency injection?

- `https://java-design-patterns.com/patterns/dependency-injection/`
- `https://martinfowler.com/articles/injection.html`

Difference between integration and system test

- Integration test tests subset of components
- System test tests from highest (i.e. user) level
- Some edge cases and erroneous behaviours cannot be tested on system level
  - For example: wrong input is prevented at highest level

Next lecture on Friday!

When: This Friday March 7 at 8:45
Topic: Model-based testing & Labeled Transition Systems

Extra slides below

For your information; won't be tested on the exam

Other testing technique: Mutation testing

- Goal: determine the quality of test suite
- How it works:
  - Make small changes to your program to create mutants, e.g.
    - change $<=$ into $<$
    - change $+$ into $-$
    - change `&&` into `||`
    - change `.get(0)` into `.get(1)`
    - delete statement
    - delete function body
    - ...
  - Execute your test suite
  - How many mutants does it *kill* (detect)?
  - Add tests to improve percentage of killed mutants
- Tool: e.g. Pitest, Stryker

Other testing technique: Property-based testing

- Goal: check property of function, e.g.
  - `add(x,y) == add(y,x)`
  - `myList == append(split(myList))`
  - `nonEmpty(computePermutations(someList))`
- How it works:
  - Generate random values for input parameters
  - Check property for all those values
  - Possibly: find a counterexample!
- Tool: e.g QuichCheck