



SOFTWARE DESIGN ESSENTIALS

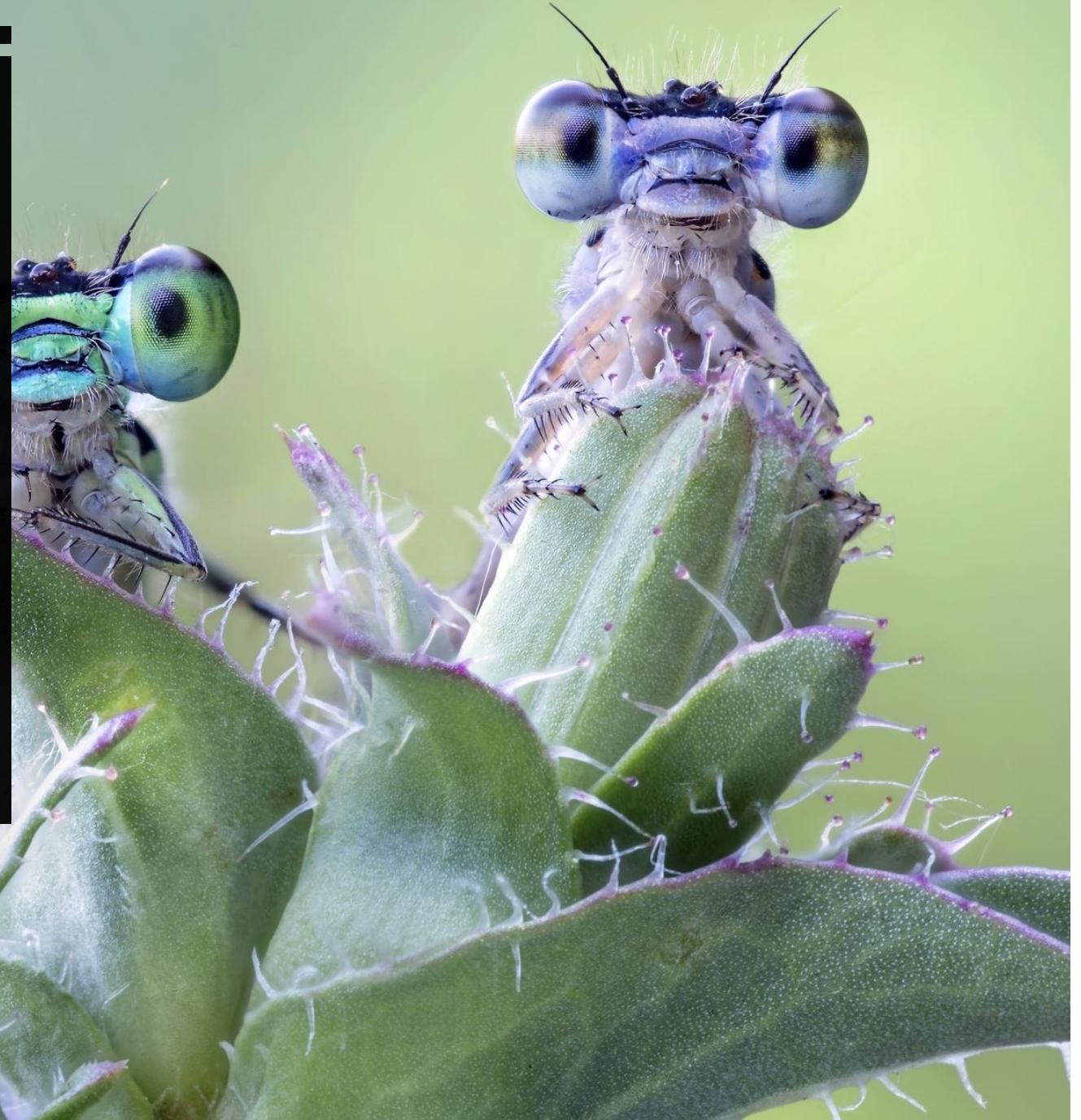
HOOFDSTUK 6: SOFTWARE TESTING

INLEIDING SOFTWARE TESTING

- Begrijpen waarom software testen belangrijk is
- Uitleggen wat fouten, gebreken en falen zijn
- Kennismaken met testtechnieken zoals blackbox en whitebox
- Eenvoudige testcases ontwerpen
- Verschillende testlevels situeren in het ontwikkelproces

WAAROM TESTEN?

- Bugs kunnen grote gevolgen hebben (veiligheid, geld, reputatie)
- Je eigen code werkt zelden perfect vanaf het begin
- Testen is essentieel voor kwaliteit en betrouwbaarheid



GROTE GEVOLGEN: CROWDSTRIKE-UPDATE VEROORZAAKT WERELDWIJDE IT-STORINGEN (2024)

Crowdstrike-update sloopt Windows-pc's wereldwijd, hinder voor NMBS en Brussels Airport

beveiliging 19.07.24 09:42 6 min Jens Jonkers, Michael Aussemens



featured

Open source in 2025: volwassen, Europees en klaar voor de toekomst software 28.03.25

recent in beveiliging

F5: 'Meer dan de helft van het webverkeer wordt nu gegenereerd door bots' beveiliging 31.03.25

World Backup Day 2025: sta nog eens extra stil bij je back-upstrategie beveiliging 31.03.25

Opgelet: je nieuwe

In juli 2024 bracht een foutieve update van de beveiligingssoftware van CrowdStrike wereldwijd computersystemen in de problemen. Deze update leidde tot het beruchte *Blue Screen of Death* op Windows-systemen, waardoor bedrijven en organisaties ernstige operationele verstoringen ondervonden. De financiële impact was aanzienlijk, met duizenden geannuleerde vluchten en onderbrekingen in kritieke diensten zoals ziekenhuizen en de hulpdiensten.

WAT BEDOELEN WE MET KWALITEIT EN BETROUWBAARHEID?

- Kwaliteit betekent: *de software doet wat het moet doen* – volgens de specificaties én volgens wat de gebruiker verwacht.
- Betrouwbaarheid betekent: *de software blijft correct werken*, ook onder ongewone omstandigheden (foutieve input, zware belasting, lange tijd draaien...).

WAAROM IS TESTEN ESSENTIEEL?

- Je ontdekt fouten voordat de gebruiker ze tegenkomt
 - Zonder testen worden fouten pas ontdekt *in productie*.
 - Dat is duurder, schadelijker voor vertrouwen én vaak moeilijker te fixen.
 - "Hoe vroeger je een bug vindt, hoe goedkoper hij is om op te lossen."
- Je valideert of de software doet wat de bedoeling is
 - Zelfs als er geen *technische fouten* zijn, kan de software *niet doen* wat de gebruiker verwacht.
 - Testen checkt: is de functionaliteit correct? Zijn de rand gevallen gedekt?

WAAROM IS TESTEN ESSENTIEEL?

- Je meet stabiliteit onder stress of uitzonderingen
 - Hoe reageert de software op rare input? Op overbelasting?
 - Testen zorgt dat je niet voor verrassingen komt te staan.
- Testen maakt kwaliteit zichtbaar en bespreekbaar
 - Niet getest = niet bewezen.
 - Dankzij tests kan je aan je team, klant of eindgebruiker tonen: “Kijk, dit werkt zoals afgesproken.”



FOUT, GEBREK, FALEN

WAT IS HET VERSCHIL?

FOUT (ERROR) – DE *DENKFOUT*

- “Ik dacht dat 365 dagen altijd een jaar is...”
- Een menselijke vergissing in analyse, ontwerp of codering.
- Voorbeeld: een programmeur schrijft if (month > 12) in plaats van if (month >= 12).

GEBREK (FAULT / BUG) – DE VERKEERDE CODE

- De fout is in de software terechtgekomen.
- De fout zorgt voor een stukje verkeerde logica in het programma.

- Voorbeeld: Door die fout werkt een kalenderfunctie verkeerd.

FALEN (FAILURE) – DE ZICHTBARE CRASH OF FOUT

- De gebruiker ziet dat er iets mis is.
- De software werkt niet zoals bedoeld of verwacht.

- Voorbeeld: De kalender app toont 32 december, of crasht.

VOORBEELD



Een programmeur vergeet in een bankapp te controleren of het saldo ≥ 0 (**fout**)



de code laat een negatieve overschrijving toe
(gebrek)



de gebruiker ziet plots -€5000 op z'n rekening
(falen).

NIET ELK GEBREK LEIDT TOT FALEN!

- Soms zit er een **bug** in de code die **niet wordt geactiveerd** omdat die functie zelden wordt gebruikt.
- Of de fout treedt pas op onder specifieke omstandigheden die nog niet getest zijn.

ONZICHTBAAR FALEN

- "Het systeem haalt de specificatie niet, maar niemand merkt het op."
- Bijvoorbeeld: een boekhoudprogramma rekent correct, maar vergeet een logbestand bij te houden, zoals vereist in de audit-specificatie.
- Er is géén crash. De gebruiker is tevreden. Maar volgens de vereisten faalt het systeem.
- Dit type falen is gevaarlijker, omdat het niet gedetecteerd wordt.

- Niet elk gebrek leidt tot een zichtbaar falen, en soms is **onzichtbaar falen zelfs gevaarlijker dan zichtbaar falen.**

- We weten nu waarom testen cruciaal is: fouten kunnen grote gevolgen hebben.
- Maar hoe ga je verstandig om met testen?
- Wat zijn de **basisprincipes** die élke tester in het achterhoofd moet houden?
- *We duiken nu in de fundamentele ideeën achter goed testwerk.*

TESTEN IS BELANGRIJK – MAAR WAT ZIJN DE BASISREGELS?

TESTPRINCIPES

7 DINGEN DIE JE
MOET
ONTHOUDEN OVER
TESTEN

TESTEN TOONT DE AANWEZIGHEID VAN FOUTEN, NIET DE AFWEZIGHEID

- Als je geen fouten vindt, betekent dat niet dat ze er niet zijn.
- Geen fout gevonden ≠ foutloos systeem.

EXHAUSTIEF TESTEN IS NIET MOGELIJKHEID

- Je kunt niet alle inputs, paden, scenario's testen.
- Dus: kies slim, test risicogedreven.

TEST VROEG EN REGELMATIG

- De testactiviteiten dienen zo vroeg mogelijk opgestart te worden in het software ontwikkelingsproces en dienen regelmatig herhaald te worden.
- Hoe vroeger een fout opgespoord is hoe eenvoudiger en goedkoper ze op te lossen.
- Begin tijdens analyse, niet pas na coderen.

ACCUMULATIE VAN FOUTEN

- Er is geen gelijkmatige verdeling van fouten binnen een testobject. Op de plaats waar men één fout terugvindt, is het waarschijnlijk dat er meerdere aanwezig zijn.
- Dus: Extra aandacht voor gevoelige zones.

AFNEMENDE EFFECTIVITEIT

- De effectiviteit van testen nemen af na een tijd. Indien testcases enkel herhaald worden, zullen ze geen nieuwe fouten blootleggen.
- Fouten die in niet geteste functies achterblijven worden op deze manier niet ontdekt.
- Testcases dienen dus regelmatig aangepast worden.

TESTEN ZIJN CONTEXTGEVOELIG

- Geen twee systemen zijn dezelfde en kunnen bijgevolg op dezelfde manier getest worden.
- De intensiteit van de tests, de definitie van de eindcriteria, etc moeten voor ieder systeem bepaald worden, aangepast aan de context.
- Bijvoorbeeld
 - Online banking ≠ game ≠ IoT-device.

ABSENCE-OF- ERRORS FALLACY

- Ook een foutloos systeem kan onbruikbaar zijn als het niet voldoet aan de verwachtingen.
- Test ook op gebruikservaring, niet enkel op bugs.

- Je kent nu de regels en denkwijzen achter testen.
- De volgende stap? **Zelf testcases leren opstellen.**
- Hoe kies je slimme testgevallen? Hoe weet je of je voldoende test?
- *We gaan kijken hoe je testgevallen bouwt op basis van specificaties en logica.*

VAN THEORIE NAAR PRAKTIJK: HOE ONTWERP JE GOEDE TESTCASES?

TEST CASE DESIGN

WAT, WAAROM,
HOE

WAT IS EEN TEST CASE?

Een test case bestaat uit:

- een input (of situatie),
- de verwachte output,
- en randvoorwaarden.

Doel: controleren of het systeem correct werkt in die situatie.

WAT MAAKT EEN GOEDE TEST CASE?

- Test iets relevants (specifiek doel)
 - Is herhaalbaar en duidelijk
 - Heeft een **verwachte output** (die je kunt vergelijken)
 - Kan een fout aan het licht brengen (destructieve mindset)
-
- Voorbeeld: "Invoer = 0, verwacht: foutmelding 'getal > 0 vereist'"

ONTWERPEN VAN TEST CASES

- Start bij de specificaties: wat moet het systeem doen?
- Kies slimme inputs: niet zomaar willekeurig
- Denk aan:
 - typische gevallen
 - grensgevallen
 - foute invoer
- Gebruik *blackbox* of *whitebox* technieken om testcases te genereren

CLASSIFICATIE VAN TESTONTWERPTECHNIEKEN

ER ZIJN
VERSCHILLENDE
MANIEREN OM
GOEDE TESTCASES
TE BEDENKEN

WAAROM TESTONTWERPTECHNIEKEN?

- Je wil testcases op een **systematische** en **efficiënte** manier bedenken.
- Doel: maximaal fouten opsporen met een beperkt aantal testcases.
- Daarom gebruiken we **testontwerptechnieken**: methoden om testgevallen af te leiden uit specificaties of code.

CLASSIFICATIE VAN TESTONTWERPTECHNIEKEN

We onderscheiden drie hoofdtypen:

1. Blackbox-testtechnieken

Zonder kennis van de interne werking (gebaseerd op *wat* het systeem moet doen)

2. Whitebox-testtechnieken

Met kennis van de interne werking (gebaseerd op *hoe* het systeem werkt)

3. Ervaringsgebaseerde technieken

Gebaseerd op intuïtie, ervaring en domeinkennis van de tester



BLACKBOX TESTONTWERPTECHNIEKEN

BLACKBOX TESTEN

- Bij **blackbox testing** test je de software zonder de *interne werking* te kennen.
- Het systeem wordt bekeken als een **zwarte doos**: je stopt er input in, kijkt naar de output, en controleert of die klopt met de verwachtingen.
- De tests worden opgesteld **op basis van specificaties**, niet van code.
- Je probeert een **representatieve selectie** van situaties te testen – gewone én uitzonderlijke gevallen.
- Je vergelijkt steeds de **werkelijke output** met de **verwachte output**.

BLACKBOX TESTEN

- Maar... is volledige dekking mogelijk?

De enige manier om écht alles te testen, zou zijn om **alle mogelijke inputs en scenario's** te proberen: dat heet **exhaustief testen**.

In de praktijk is dat **onhaalbaar**:

- Er zijn **te veel combinaties**.
- De **tijd en middelen** zijn beperkt.
- Soms zou je zelfs het hele geheugen volschrijven bij sequentiële inputs.

WAT IS DAN HET DOEL?

- Je wil zo veel mogelijk fouten vinden, met een beperkt aantal testcases.
- Daarom draait blackbox testing om het maken van **slimme keuzes**:
 - welke inputs zijn typisch?
 - waar verwacht je fouten?
 - wat zijn de randen van het gedrag?
- Het gaat dus om het **economisch maximaliseren** van foutendetectie met beperkte middelen.

EQUIVALENCE PARTITIONING

- Idee: verdeel de invoerruimte in **klassen** waarvan je aanneemt dat ze vergelijkbaar gedrag opleveren.
- **Voorbeeld:** test een methode die een getal tussen 1 & 1000 verwacht.
 - Testen 1...1000 is niet nuttig (1000 tests)
 - Gebruik 3 data klasses dmv **equivalence partitioning**
 - Klasse 1: een getal tussen 1 & 1000
 - Klasse 2: een getal < 1
 - Klasse 3: een getal > 1000
 - Je test **één waarde per klasse** (bv. 500, 0, 1200)

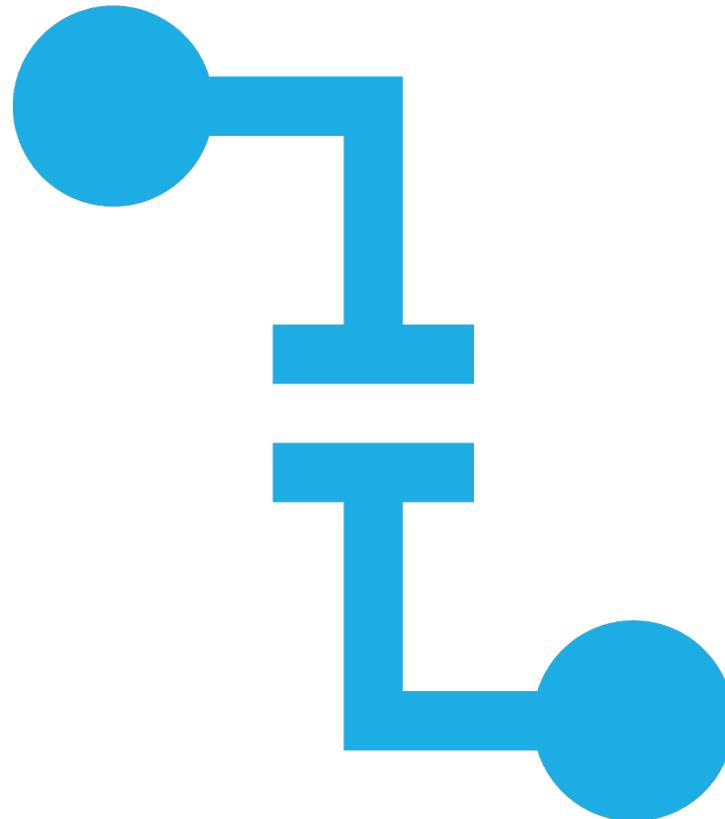
BOUNDARY-VALUE ANALYSIS

- Fouten treden vaak op aan de **randen** van toegelaten waarden.
- Test daarom **waarden net op, onder en boven** de grenzen.
- **Voorbeeld:** test een methode die een getal tussen 1 & 1000 verwacht.
 - Testen 1...1000 is niet nuttig (1000 tests)
 - Gebruik **boundary-value analysis**
 - Geldige grensgevallen: 1 en 1000
 - Net iets hoger dan de grensgevallen: 2 en 1001
 - Net iets lager dan de grensgevallen: 0 en 999
 - Test: 0, 1, 2, 999, 1000, 1001

VOORBEELDEN

- Equivalence Partitioning:
 - Input voor leeftijd = 0 t.e.m. 120
Testklassen: onder 0, 0-120, boven 120

- Boundary Value:
 - Minimum loonbedrag = €2000
Test: €1999, €2000, €2001



WHITEBOX TESTONTWERPTECHNIEKEN

WHITEBOX TESTEN

- Bij whitebox testing ontwerp je testcases op basis van **inzicht in de code**.
- Je gebruikt je kennis van **datastructuren, algoritmen en logica** om de software van binnenuit te testen.
- De focus ligt op het **analyseren van de interne werking** en het gericht testen van gevoelige of complexe stukken code.

WAT TEST JE?

- Je kijkt naar:
 - **control flow** (welke paden kunnen er doorlopen worden?)
 - **grenswaarden** (bijv. net boven of onder een limiet)
 - **mogelijke fouten zoals deling door nul, null-references, overflows, enz.**

VOLLEDIGE DEKKING?

- In theorie kun je streven naar **exhaustieve dekking** (alles testen), maar in de praktijk is dat:
 - vaak niet **realistisch**
 - **te complex** bij grote codebases
- Daarom gebruik je **coverage criteria** zoals:
 - *statement coverage*
 - *branch coverage*
 - *condition coverage*

CODE COVERAGE

- Statement Coverage
 - Zorg dat **elke regel code** minstens één keer wordt uitgevoerd
 - Eenvoudigste vorm van structurele dekking
- Branch (Decision) Coverage
 - Zorg dat **elke beslissing** minstens één keer **true** en één keer **false** is
 - Bijvoorbeeld: if ($x > 0$) → test $x = -1$ en $x = 1$
- Condition Coverage
 - Elke **subconditie** in samengestelde beslissingen moet alle mogelijke waarden aannemen.
 - Voorbeeld: if ($a > 1 \&& b == 0$)
 - Test:
 - $a > 1 \& b == 0$
 - $a > 1 \& b \neq 0$
 - $a \leq 1 \& b == 0$
 - $a \leq 1 \& b \neq 0$

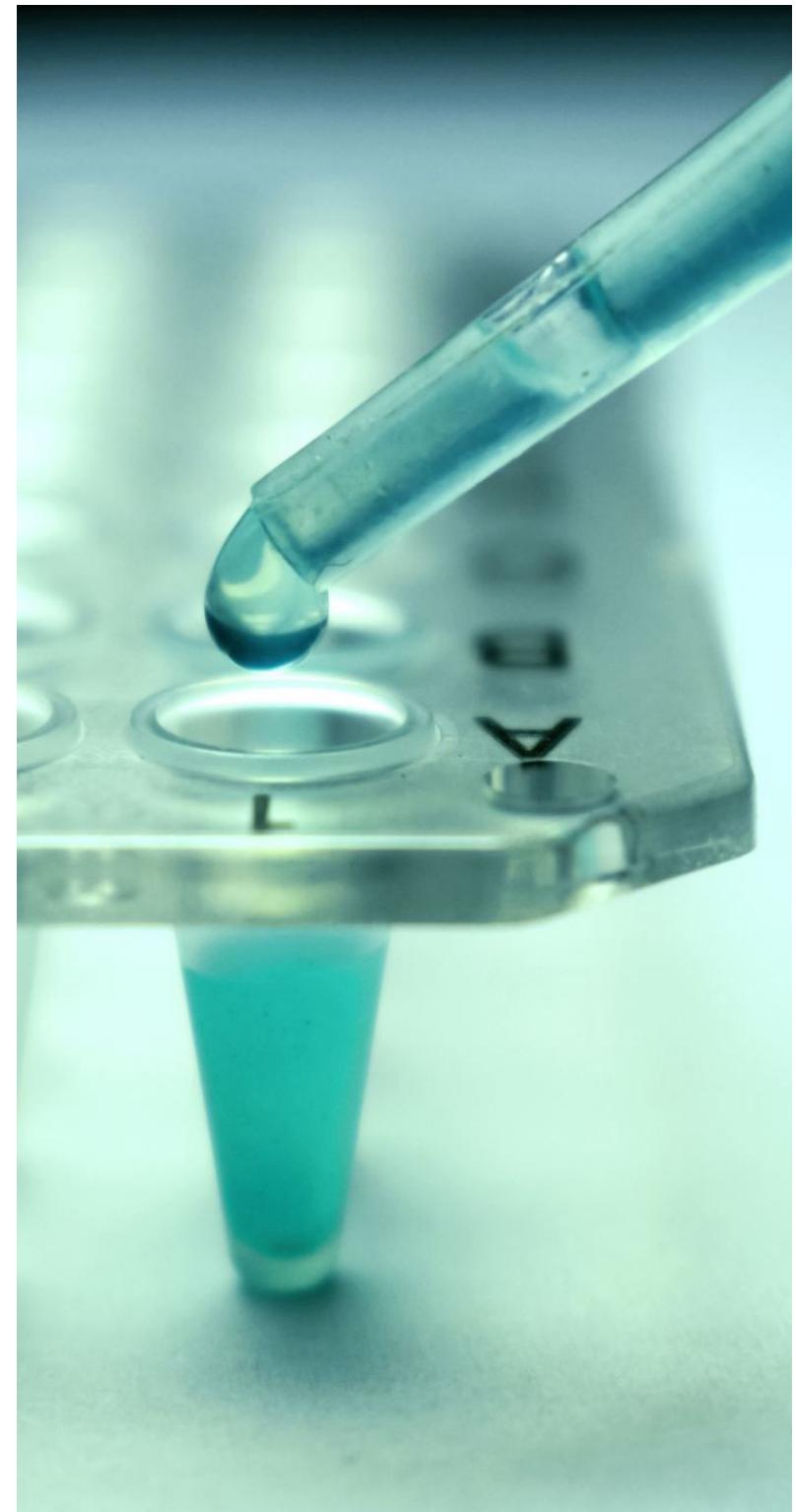
ERVARINGSGEBASEERDE TESTS

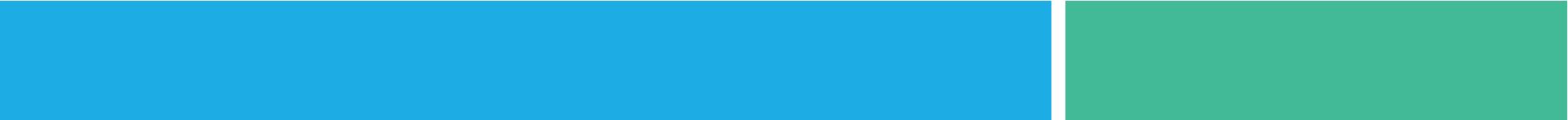
- Deze tests worden ontworpen op basis van de **ervaring, intuïtie en domeinkennis** van de tester.
- Ze zijn geen vervanging van formele methodes, maar dienen als **waardevolle aanvulling**.
- Vaak sporen ze fouten op die andere, meer gestructureerde technieken missen – zoals typische valkuilen of edge cases die je alleen leert kennen met de jaren.

ERVARINGSGEBASEerde TESTS

Let op:

- De kwaliteit en effectiviteit van deze tests hangen sterk af van de ervaring van de tester.
- Een beginnende tester zal minder kunnen inschatten waar fouten zich kunnen verstopen dan een doorwinterde ontwikkelaar of tester.





AANPAK

HOE COMBINEER JE TESTTECHNIEKEN SLIM?

Bij het ontwerpen van testcases kies je best een mix van technieken. Zo pak je verschillende soorten fouten aan en verhoog je je kans op foutdetectie:

- 1. Begin met boundary-value analysis**
 - 1.** Test altijd de **grenzen van toegelaten waarden**: net onder, op en net boven.
- 2. Gebruik equivalence partitioning**
 - 1.** Deel input (en eventueel output) op in **valide en invalide klassen**.
 - 2.** Test minstens één waarde per **klasse** om redundantie te vermijden.

HOE COMBINEER JE TESTTECHNIEKEN SLIM?

3. Voeg ervaringsgebaseerde tests toe

- Denk na over waar het systeem fout kan lopen:
 - lege invoer
 - onverwachte volgorde
 - speciale tekens
 - ontbrekende waarden, enz.

4. Onderzoek de code als je toegang hebt (whitebox)

- Check of je testcases voldoende **code-dekking** bieden:
 - Heb je **alle beslissingen** getest (true én false)?
 - Zijn **alle condities** eens als waar/onwaar geëvalueerd?
- Voeg testcases toe waar nodig om **gaten in de dekking** te dichten.

- We hebben nu geleerd *hoe* we testcases kunnen ontwerpen (equivalence partitioning, boundary analysis, error guessing...).
- Maar:
 - *Op welk moment* gebruik je welke test?
 - *Wat test je precies op welk niveau van het systeem?*
- Tijd om kennis toe te passen in de praktijkstructuur: **de testlevels!**

VAN TECHNIEKEN NAAR TESTNIVEAUS: WAT TEST JE, EN WANNEER?

TESTLEVELS

WAAROM TESTLEVELS?

- Tijdens de ontwikkeling van software testen we niet alles tegelijk.
- We bouwen het systeem **geleidelijk op**: eerst de kleine stukjes, dan de samenhang, en uiteindelijk het geheel.
- Elke stap vraagt om een ander soort test.
- Daarom spreken we van **testlevels**: verschillende lagen van testen, elk met hun eigen doel, focus en aanpak.

TESTEN DOORHEEN DE SOFTWARE LIFE CYCLE

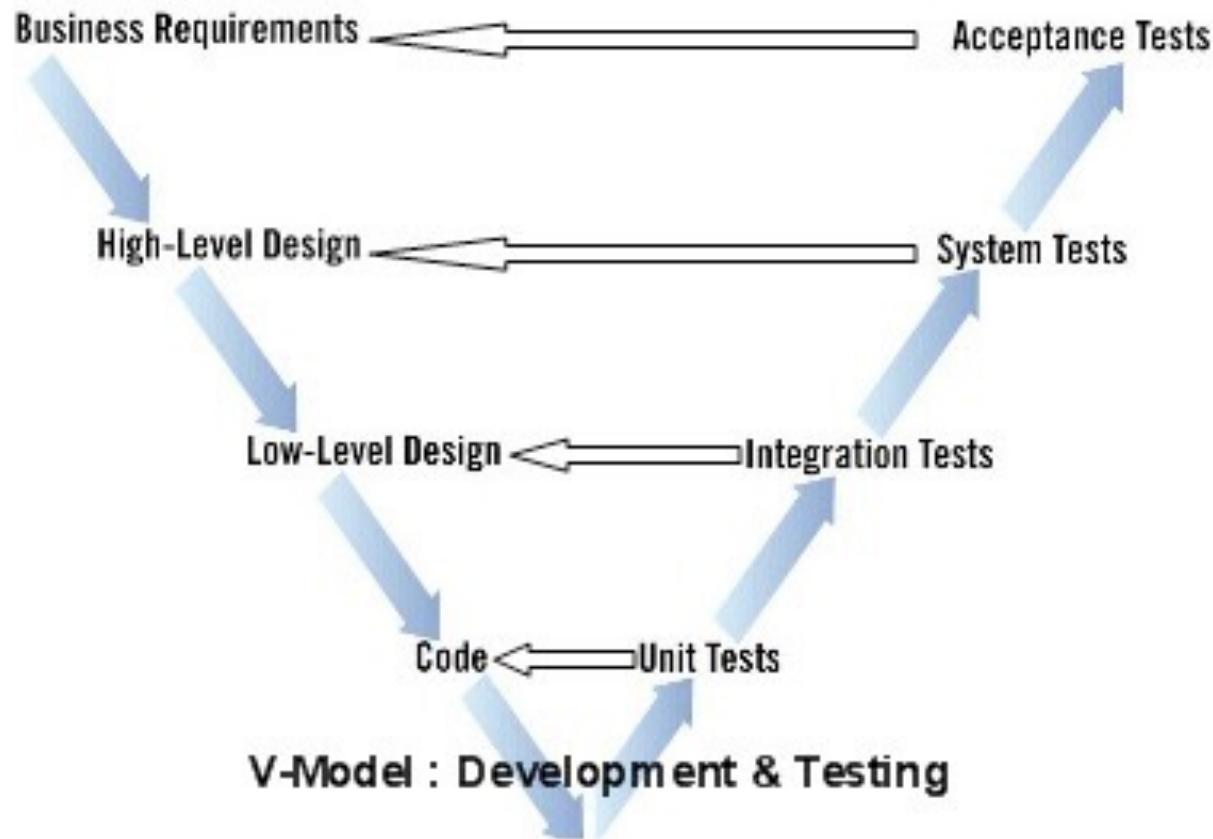
Testen staat nooit op zichzelf

- Testen gebeurt niet los van de ontwikkeling, maar maakt er integraal deel van uit.
- Elke testactiviteit is nauw verbonden met een ontwikkelfase: van analyse tot implementatie.
- Het ontwikkelmodel dat je gebruikt (zoals waterval, V-model of Agile) bepaalt mee hoe en wanneer je test.
- *Testen moet dus altijd aangepast worden aan de context van het project.*

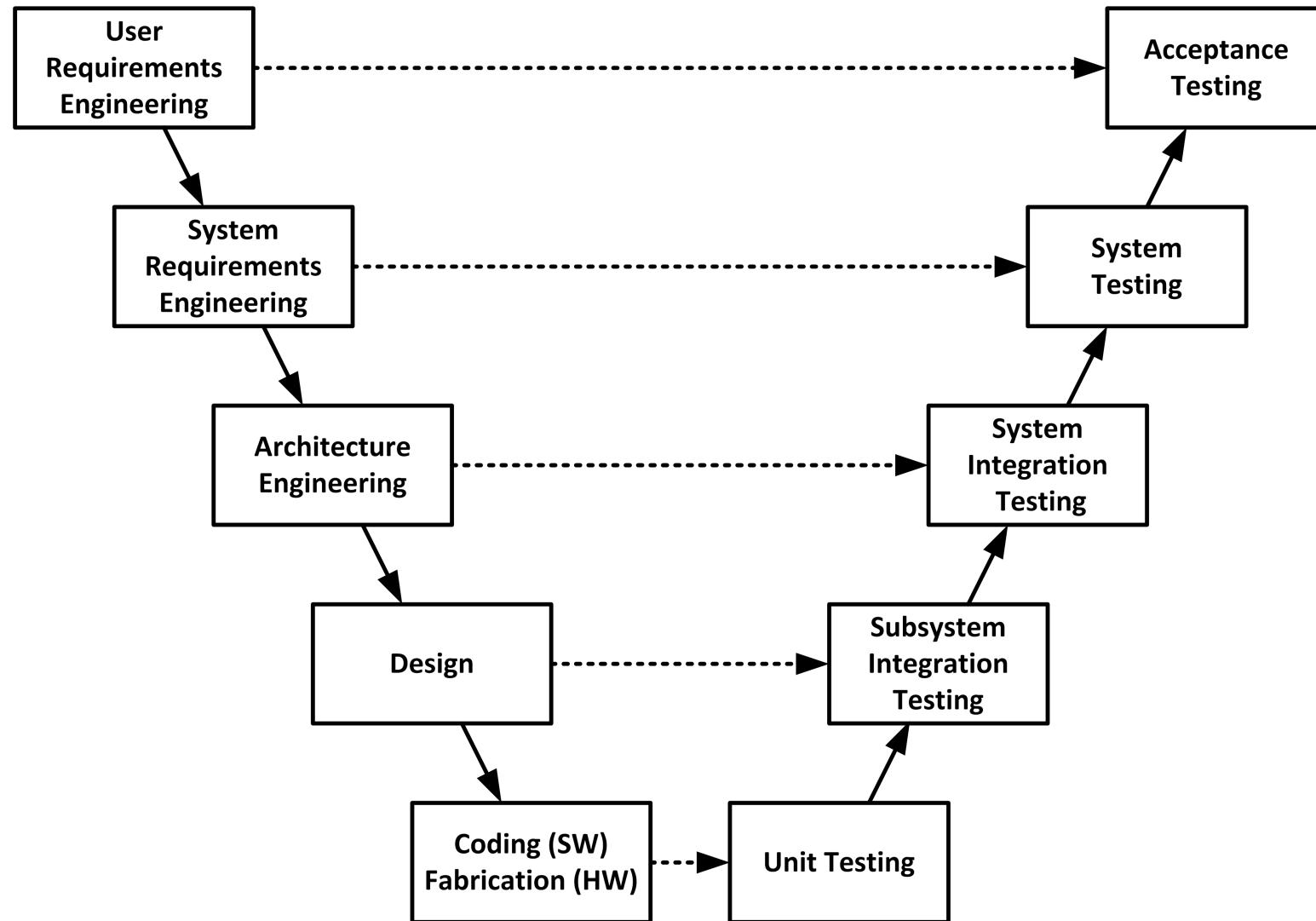
V-MODEL

- Het V-model is een uitbreiding van het klassieke **watervalmodel**, maar dan met een sterke focus op **testen**.
- De term verwijst niet naar één specifieke methode, maar naar een **familie van ontwikkelmodellen** die deze V-structuur volgen.
- Het is een **conceptueel model** dat helpt om op een eenvoudige manier de **complexiteit van softwareontwikkeling en testing** inzichtelijk te maken.
- Het is vooral handig om de **relatie tussen ontwikkelfasen en testfasen** te begrijpen.

V-MODEL (V1)



V-MODEL (V2)



V-MODEL VS. AGILE

- Het V-model is goed om het idee van testlevels te begrijpen.
- In moderne projecten (bv. Agile) loopt alles sneller en door elkaar.
- Toch blijven dezelfde soorten tests bestaan – alleen anders georganiseerd.
- Alle testtypes kunnen in elke sprint terugkomen. Testen gebeurt continu.

TESTLEVELS

- **Unit testing**
Testen van afzonderlijke functies of methodes
"Werkt dit stukje code correct op zichzelf?"
- **Integratietesting**
Testen hoe componenten samenwerken
"Praten de stukjes goed met elkaar?"
- **Systeemtesting**
Testen van het volledige systeem als één geheel
"Doet het systeem wat het moet doen volgens de vereisten?"
- **Acceptatietesting**
Testen door of voor de eindgebruiker
"Is dit systeem klaar om echt te gebruiken?"

WAT ZIJN TESTLEVELS

- **Testlevels** zijn verschillende niveaus waarop je software test tijdens het ontwikkelproces.
- Ze geven aan **wanneer** je test, **wat** je test en **met welk doel**.
- Elk testlevel bestaat uit een groep van **testactiviteiten** die samen uitgevoerd en beheerd worden.
- Zo krijg je een gestructureerde aanpak, van kleine stukjes code tot het volledige systeem.

TESTLEVEL ≠ TESTFASE

- Een **testlevel** verwijst naar wat je test (bijvoorbeeld: unit, integratie, systeem...).
- Een **testfase** verwijst naar wanneer je die test uitvoert binnen het project.
- In **iteratieve of incrementele ontwikkelmodellen** worden **dezelfde testlevels herhaald** in meerdere fasen.
 - Bijvoorbeeld: in elke sprint voer je opnieuw unit tests en integratietests uit.
- Een testlevel is dus **inhoudelijk**, een testfase is **tijdgebonden**.

TESTLEVELS

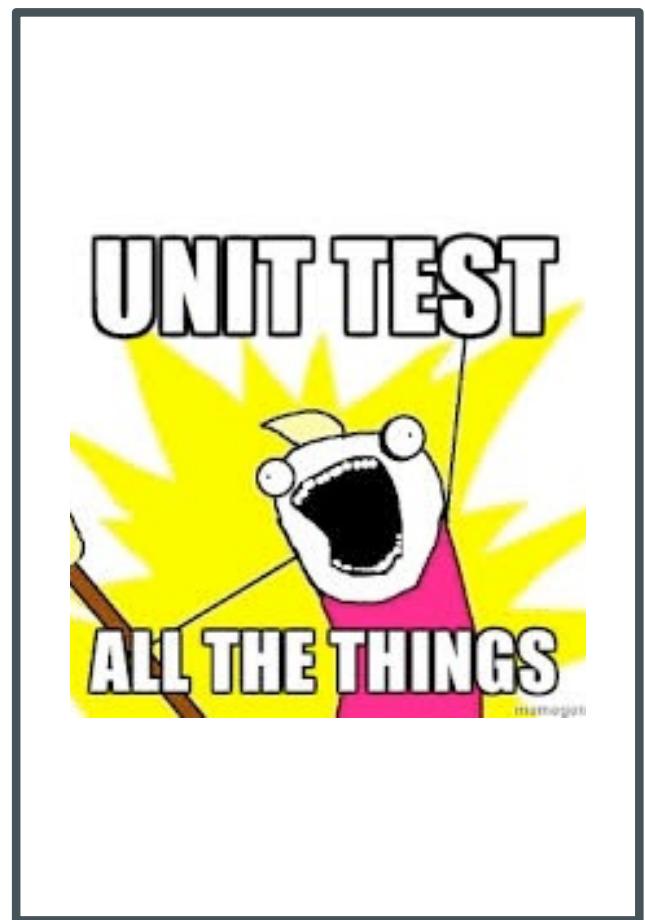
- Er zijn vier testlevels
 - Unit testing
 - Integratie testing
 - System testing
 - Acceptance testing

VROEG FOUTEN OPSPOREN MET REVIEWS

- Het nakijken van vereisten (requirements reviews) is een effectieve manier om al in een vroeg stadium fouten op te sporen.
- Door goed te kijken naar wat het systeem moet doen, kun je **onduidelijkheden, tegenstrijdigheden of ontbrekende info** vinden *nog vóór er code wordt geschreven*.
- Dit maakt het opsporen én oplossen van fouten **veel goedkoper en sneller** dan wanneer ze pas tijdens het testen worden ontdekt.

UNIT TESTING

- Unit testing focust op het testen van **één afzonderlijk deel van de software** – zoals een functie, methode of klasse.
- Dit gebeurt meestal los van de rest van het systeem, met gesimuleerde invoer en gecontroleerde uitvoer.
- Unit tests kunnen geschreven worden vanuit:
 - een **whiteboxbenadering** (met kennis van de interne werking), of
 - een **blackboxbenadering** (gebaseerd op de verwachte interface en output).
- Doel: controleren of de unit **in isolatie correct werkt**, ongeacht de rest van de applicatie.



UNIT TESTING

- In de eerste plaats test je de **functionele werking** van de unit:
Doet de functie of methode precies wat ze moet doen?
- Soms bekijk je ook **niet-functionele aspecten**, zoals:
Hoe wordt er omgegaan met geheugen of resources? (bijv. opsporen van memory leaks)
- Daarnaast kun je **structurele testen** uitvoeren:
Heb je alle mogelijke paden in de code getest? (bijvoorbeeld met **decision coverage**)
- Het doel is om de **unit grondig en in isolatie** te testen, zodat je zeker weet dat dit onderdeel op zichzelf correct werkt.

UNIT TESTING

Je leidt testcases af uit:

- de **specificaties** van het component (wat moet het precies doen?)
- het **softwareontwerp** (hoe is de logica opgebouwd?)
- het gebruikte **datamodel** (welke structuren en types worden verwacht?)
- Deze bronnen helpen je om gerichte en relevante testgevallen te formuleren.

HOE VOER JE UNIT TESTS UIT?

- In de meeste gevallen heeft de tester (vaak de ontwikkelaar zelf) toegang tot de broncode.
- Unit tests worden dan uitgevoerd met behulp van een **unit test framework** (zoals JUnit, NUnit, Pytest, enz.).
- Een mogelijke aanpak is **Test-Driven Development (TDD)**:
 - Eerst schrijf je de test, daarna pas de code die nodig is om die test te doen slagen.

ONAFHANKELIJKHEID VAN UNIT TESTS

- Unitests moeten volledig onafhankelijk van elkaar zijn.
- Elke test moet **op zichzelf kunnen draaien**, zonder afhankelijk te zijn van de uitvoering of het resultaat van andere tests.
 - Zo weet je zeker dat een fout in de ene test **geen invloed heeft op andere tests**, en dat je testresultaten betrouwbaar zijn.

INTEGRATIETESTEN

- **Integratietests** controleren of verschillende componenten correct samenwerken.
- Ze testen de interfaces tussen modules, maar ook de **interacties met externe onderdelen** zoals:
 - het besturingssysteem,
 - het bestandssysteem,
 - hardware,
 - en zelfs andere systemen of applicaties.
- Het doel is om fouten op te sporen die ontstaan door verkeerde communicatie of onverwachte koppelingen tussen onderdelen.

TWEE VORMEN VAN INTEGRATIETESTEN

Component-integratietesten

- Testen de samenwerking tussen meerdere softwarecomponenten binnen één systeem.
- Doel: nagaan of de interfaces tussen modules **correct functioneren**.
- Worden uitgevoerd **na** de unit tests (of module tests).
- *Bijvoorbeeld: werkt de communicatie tussen een login-module en een gebruikersprofielmodule?*

TWEE VORMEN VAN INTEGRATIETESTEN

Systeem-integratietesten

- Testen de interactie tussen verschillende systemen, of tussen software en hardware.
- Gaan vaak over grotere, complexe omgevingen (bv. koppeling met externe diensten, netwerken, hardwaredrivers...).
- Worden uitgevoerd **na de systeemtests**.
- *Bijvoorbeeld: werkt een softwaretoepassing correct samen met een extern betaalsysteem of een printersysteem?*

INTEGRATIETESTEN

- Tijdens integratietesten worden zowel **functionele** als **niet-functionele** eigenschappen getest.
- *Bijvoorbeeld: prestaties, zoals responsijd of verwerkingsnelheid.*
- De nadruk ligt op de **interactie tussen componenten**, niet op de interne werking van elke component afzonderlijk.
- *Het is belangrijk om te testen hoe goed onderdelen met elkaar samenwerken, niet of elk onderdeel op zich correct is – dat is al gebeurd bij de unit tests.*

SYSTEEMTESTS

- Bij systeemtesting wordt het volledige systeem getest als één geheel, op basis van de vereisten en specificaties.
- Het doel is om te controleren of het systeem zich gedraagt zoals verwacht vanuit het perspectief van de gebruiker of opdrachtgever.
- Er is geen kennis van de interne werking nodig: je test het systeem van buitenaf, alsof je een eindgebruiker bent.

REALISTISCHE TESTOMGEVING BIJ SYSTEEMTESTING

- Tijdens systeemtests is het belangrijk dat de **testomgeving zo goed mogelijk overeenkomt met de uiteindelijke productieomgeving**.
- Zo verklein je het risico op fouten die enkel optreden door **omgevingsspecifieke verschillen**, zoals configuraties, netwerkinstellingen of systeemversies.

DOEL VAN DE SYSTEEMTESTS

- Het doel van systeemtesting is om de leverancier of het ontwikkelteam ervan te overtuigen dat het systeem volledig voldoet aan alle gestelde vereisten.
- Pas wanneer dit aangetoond is, kan het systeem **met vertrouwen worden overgedragen aan de klant**.

VOORBEELDEN VAN SYSTEEMTESTS

END-TO-END FUNCTIONAL TESTING

- **Wat?** Test een volledige gebruikersflow van begin tot einde.
- **Voorbeeld:** In een webshop:
 - Product zoeken → toevoegen aan winkelmandje → afrekenen → bevestiging ontvangen.
- **Doel:** Nagaan of het systeem **volledig en correct werkt** zoals de eindgebruiker het zal gebruiken.

INSTALLATIE- EN CONFIGURATIETESTS

- **Wat?** Test of het systeem correct kan worden geïnstalleerd en geconfigureerd op een doelsysteem.
- **Voorbeeld:** Werkt de installatie op Windows én macOS? Wat als een afhankelijkheid ontbreekt?
- **Doel:** Fouten voorkomen bij uitrol naar klanten of eindgebruikers.

PERFORMANCE TESTING

- **Wat?** Test hoe het systeem presteert onder belasting.
- **Voorbeeld:** Kan een online platform 500 gelijktijdige gebruikers aan zonder vertraging?
- **Doel:** Detecteren van vertraging, vertragingen bij database, geheugenproblemen...

SECURITY TESTING

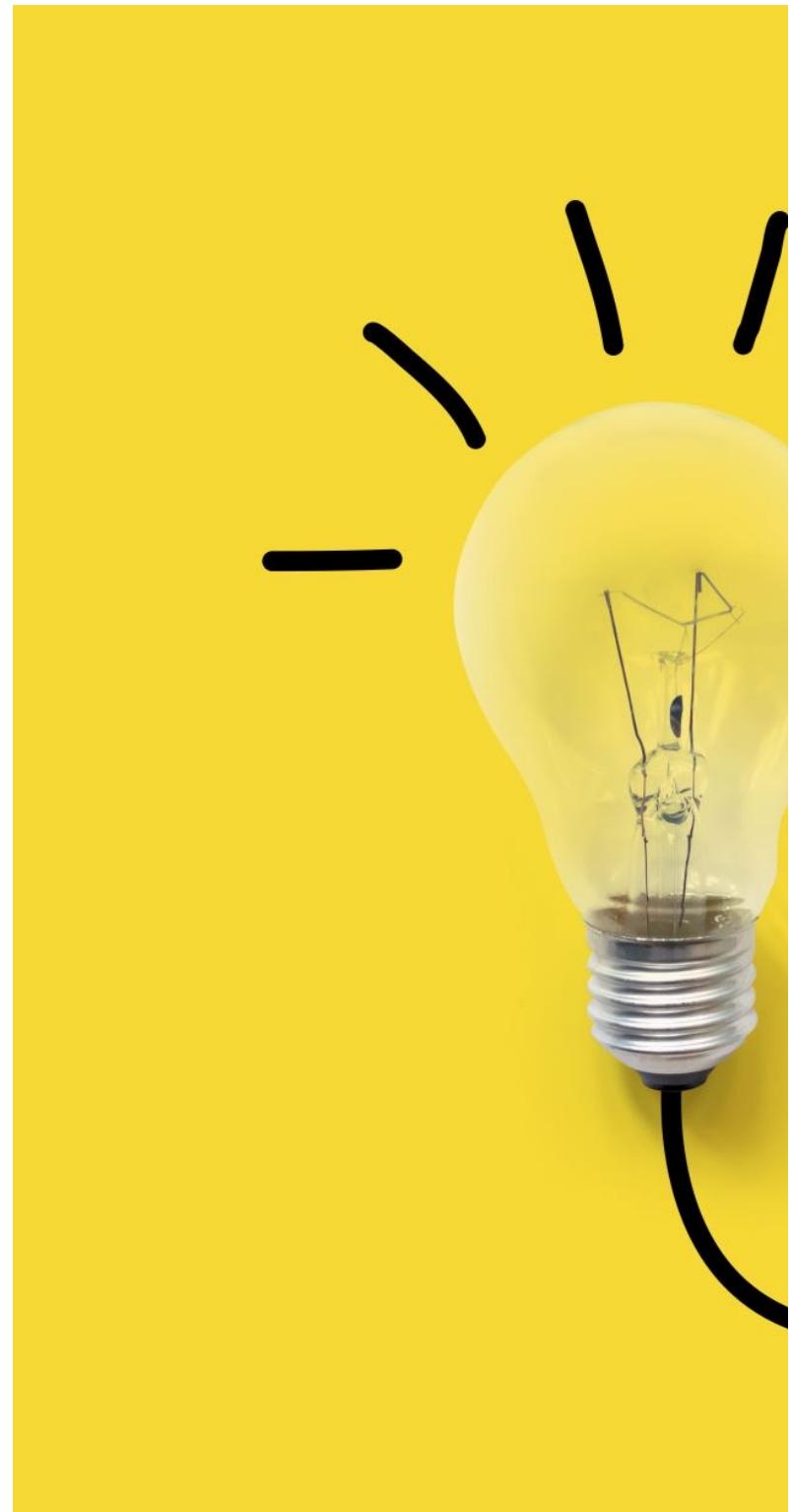
- **Wat?** Test of het systeem beveiligingsrisico's correct afhandelt.
- **Voorbeeld:** Wat gebeurt er als een gebruiker probeert in te loggen met SQL-injectie? Is er toegangscontrole op admin-functies?
- **Doel:** Voorkomen dat gevoelige gegevens uitlekken of misbruikt worden.

USABILITY TESTING

- **Wat?** Test of de software bruikbaar en begrijpelijk is voor de doelgroep.
- **Voorbeeld:** Kan een nieuwe medewerker zonder uitleg een werknemer toevoegen in een HR-systeem?
- **Doel:** Vaststellen of het systeem ook in de praktijk werkt zoals bedoeld.

ACCEPTATIETESTS

- Acceptatietests worden meestal uitgevoerd door de klant of **eindgebruikers**, al kunnen ook andere betrokkenen (zoals businessanalisten of key users) hierbij betrokken worden.
- Het belangrijkste doel is om **vertrouwen te krijgen** dat het systeem of een onderdeel ervan **klaar is voor gebruik in de praktijk**.
- De focus ligt dus niet op het actief zoeken naar fouten, maar op het **bevestigen dat het systeem voldoet aan de verwachtingen** en bruikbaar is zoals bedoeld.
- Met acceptatietesten toets je of het systeem **bruikbaar, volledig en geschikt is voor oplevering**.



ALFA- EN BETATESTS EN ACCEPTATIETESTEN?

- Alfa- en betatests zijn in feite vormen van acceptatietesten, maar dan uitgevoerd in een **realistische gebruikerscontext**, vaak buiten het formele testproces om.
- Net zoals bij acceptatietesten is het doel om:
 - te beoordelen of het systeem **klaar is voor gebruik**,
 - vertrouwen op te bouwen bij de eindgebruiker,
 - en feedback te verzamelen over de **bruikbaarheid en functionaliteit**.

VOORBEELD LOGINPAGINA

Testlevel

Wat test je?

Unit

Werkt de functie valideerWachtwoord()?

Integratie

Wordt een correcte login doorgestuurd naar backend?

Systeem

Komt gebruiker na login op het dashboard?

Acceptatie

Vindt de klant dat de login intuïtief werkt?

TEST TYPES

WAT ZIJN FUNCTIONELE TESTS?

- Functionele tests richten zich op “wat” het systeem moet doen en dus op de gedrag en functies zoals beschreven in de vereisten of specificaties.
- Ze controleren of een systeem, subsysteem of component de juiste output geeft bij een bepaalde input, ongeacht hoe het intern werkt.
- Het gaat om het verifiëren van functionaliteit, niet om prestaties, veiligheid of structuur.

WAT ZIJN NIET-FUNCTIONELE TESTS?

- Niet-functionele tests richten zich op “hoe” het systeem werkt, eerder dan wat het precies doet.
- Ze beoordelen de kwaliteitskenmerken van een systeem, zoals prestaties, gebruiksvriendelijkheid of betrouwbaarheid.
- Net als functionele tests kunnen niet-functionele tests uitgevoerd worden op elk testlevel:
van unit tot acceptatie, afhankelijk van het aspect dat je wil beoordelen.

- We hebben besproken wat we kunnen testen:
 - functioneel gedrag,
 - prestaties,
 - gebruiksvriendelijkheid, enz.
- Maar software verandert. Altijd. Nieuwe features, verbeteringen, bugfixes...
- De vraag is dan: **Hoe weet je zeker dat alles nog werkt zoals het daarvoor werkte?**
- Dat is waar **regressietesten** in beeld komen.

TESTEN STOPT NIET NA DE EERSTE KEER: INTRODUCTIE TOT REGRESSIETESTS

TESTEN IN DE TIJD

WAT GEBEURT
ER NADAT ER
IETS IS
AANGEPAST?

WAT IS REGRESSIETESTING?

- Het opnieuw testen van eerder geteste onderdelen van de software, om na te gaan of wijzigingen (in code of omgeving) onbedoeld fouten hebben veroorzaakt in andere, niet-aangepaste delen.

WAAROM REGRESSIETESTEN?

Regression:
"when you fix one bug, you introduce several newer bugs."

- Wijzigingen kunnen **onverwachte bijwerkingen** hebben.
- Ook als iets kleins verandert, kan elders iets stuk gaan.
- Regressietests helpen je **zeker te zijn dat alles nog werkt zoals voordien.**



DE TESTPIRAMIDE

DE TESTPIRAMIDE – SLIM TESTEN IN LAGEN

De **testpiramide** is een visueel model dat aangeeft **hoe je je testen best verdeelt over verschillende niveaus van een softwaretoepassing**. Het helpt teams om efficiënt, snel en betrouwbaar te testen.



Afbeelding van <https://www.headspin.io/blog/the-testing-pyramid-simplified-for-one-and-all>

OPBOUW VAN DE PIRAMIDE (VAN ONDER NAAR BOVEN)

Unit tests (basis van de piramide)

- Veel kleine, snelle tests
- Testen individuele functies of methodes in isolatie
- = Snel, goedkoop, stabile

Integratietests (middenlaag)

- Testen hoe onderdelen met elkaar samenwerken
- Bijvoorbeeld: communicatie tussen twee modules, of tussen backend en database
- = Minder talrijk, iets trager, complexer

End-to-end tests / UI-tests (toplaag)

- Testen het systeem van begin tot einde via de gebruikersinterface
- Bijvoorbeeld: een gebruiker logt in, zoekt een product en rekent af
- = Duur, traag, gevoelig voor fouten → spaarzaam gebruiken

WAAROM EEN PIRAMIDEVORM?

- Hoe lager in de piramide, hoe sneller en goedkoper de tests.
- Hoe hoger, hoe meer afhankelijkheden en dus kwetsbaarder de tests.
- Je wil dus veel unit tests, minder integratietests, en slechts een paar end-to-end tests.

VEELGEMAAKTE FOUT: DE OMGEKEERDE PIRAMIDE

- Sommige teams schrijven vooral end-to-end tests en verwaarlozen unit tests. Dit leidt tot:
 - Trage feedback
 - Instabiele builds
 - Hoge onderhoudskosten

- De testpiramide moedigt aan tot een gebalanceerde, onderhoudbare teststrategie.



TESTCASES AFLEIDEN UIT UML- DIAGRAMMEN

WAT GEBRUIK JE?

USE CASE – EEN BRON VOOR TESTCASES

Wat kun je eruit halen voor testen?

- Normale flow → "happy path" testcase
 - Test dat de **verwachte acties correct verlopen** en het gewenste resultaat opleveren.
- Alternatieve flows → varianten van het gedrag
 - Test of het systeem ook correct werkt in **andere toegelaten scenario's**.
 - Bijvoorbeeld: een gebruiker betaalt met PayPal i.p.v. kredietkaart.
- Exceptions → foutgevallen testen
 - Use cases bevatten vaak ook **uitzonderingssituaties**:
 - bijv. "Betaling mislukt" of "Geen voorraad beschikbaar"
 - Dit vormt een perfecte input voor negatieve testcases of foutafhandeling.

CLASS DIAGRAM

- Toont de **structuur** van klassen: attributen, methodes, relaties.
- Helpt bepalen **welke methodes** je moet testen.
- Je ziet ook grenzen van types (bijv. int age, String name, enz.) → handig voor boundary tests.

INTERACTION SEQUENCE DIAGRAM

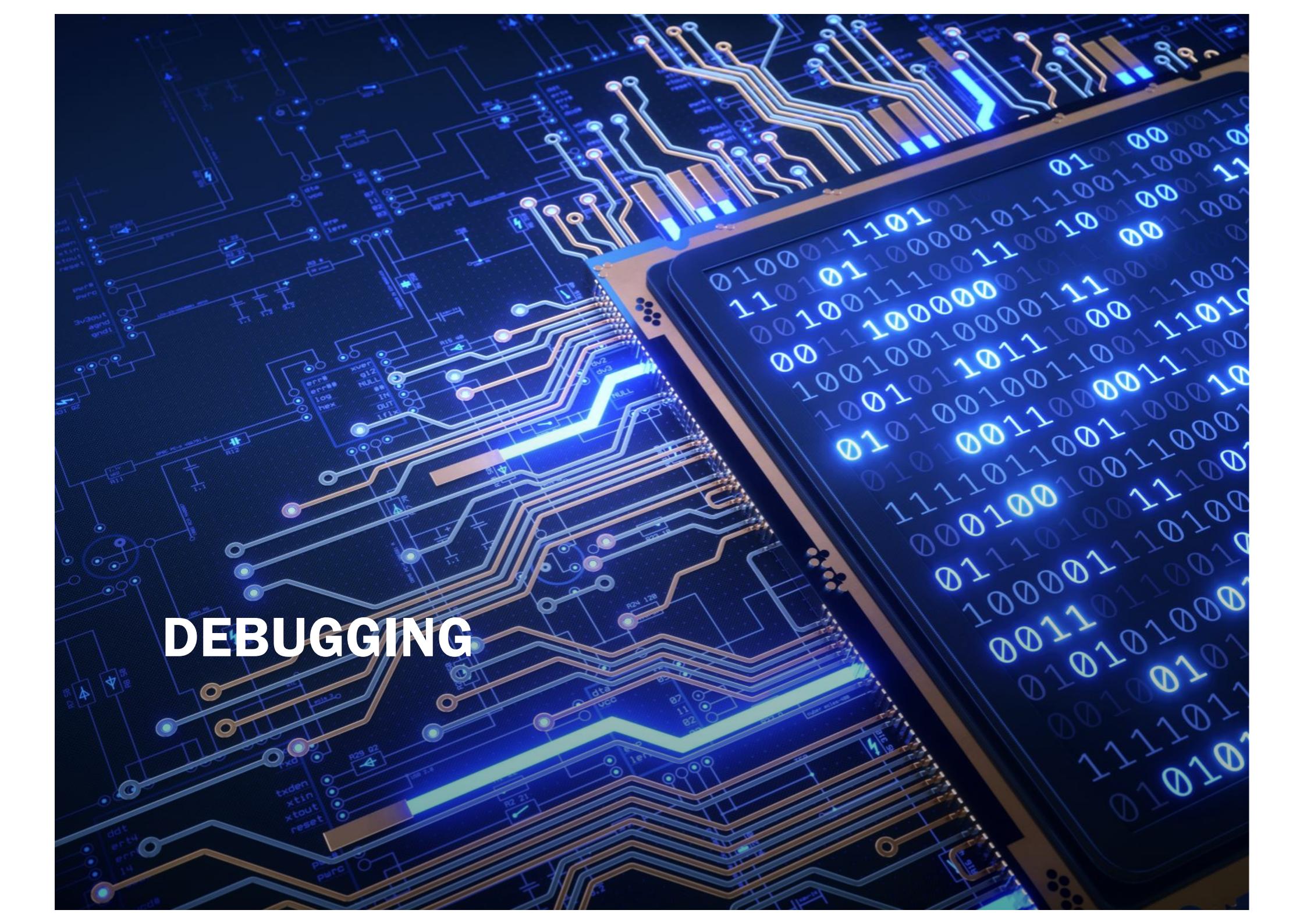
- Toont de interactie tussen objecten over de tijd.
- Je kan testcases ontwerpen om volgorde van aanroepen te controleren.
- Ook goed voor integratietests (wanneer meerdere objecten samenwerken).

STATE MACHINE DIAGRAM

- Geeft de toestanden van een object weer, en hoe het van de ene naar de andere gaat.
- Ideaal voor testcases die controleren of de **juiste overgang gebeurt bij een bepaalde input**.
- Voorbeeld: test dat een betaling van “pending” naar “confirmed” gaat bij succesvolle verwerking.

ACTIVITY DIAGRAM

- Beschrijft logische stappen of workflows.
- Kan helpen om testcases te bedenken voor elk pad in het proces.
- Ook nuttig voor whitebox-achtige tests (control flow).



DEBUGGING

DEBUGGEN EN TESTEN

- Debugging is het **analyseren en oplossen van fouten** die tijdens het testen of gebruik aan het licht komen.
- Testen toont aan *dát er iets fout is* – debugging toont *wáár en waarom*.

TOOLS EN TECHNIEKEN

- Breakpoints in IDE (bv. VS Code, IntelliJ, PyCharm...)
- Logging en foutmeldingen analyseren
- Variabeleinhoud inspecteren tijdens runtime
- Stack traces lezen