Testen & Integrieren

Phase V

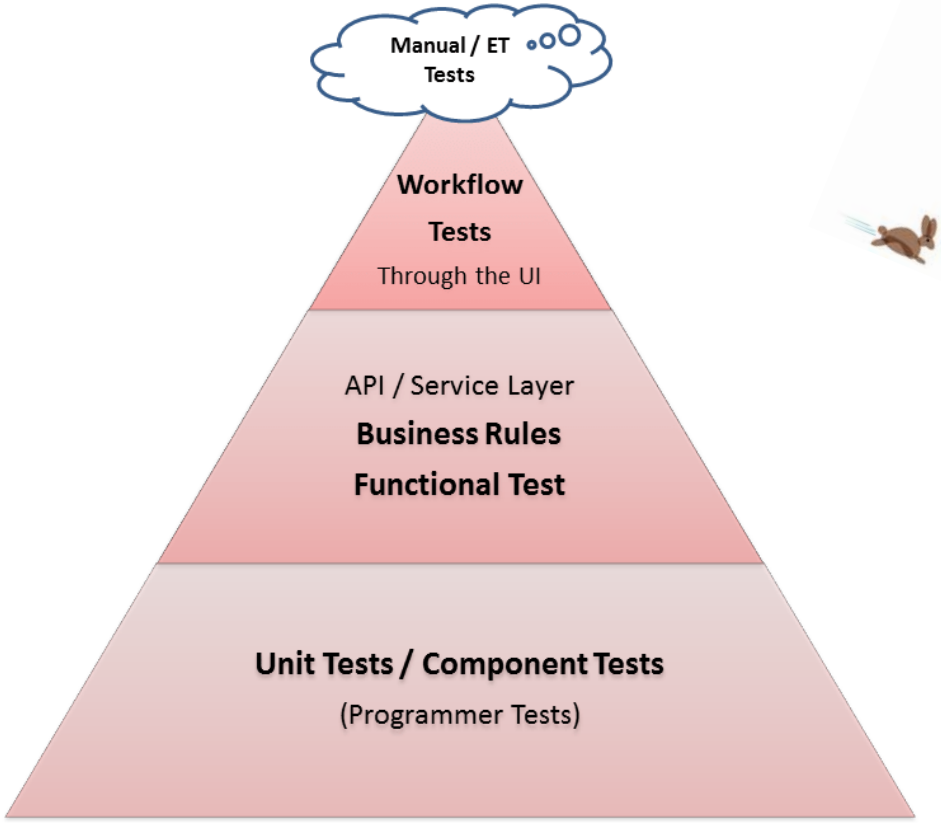
Fehlertypen Basics

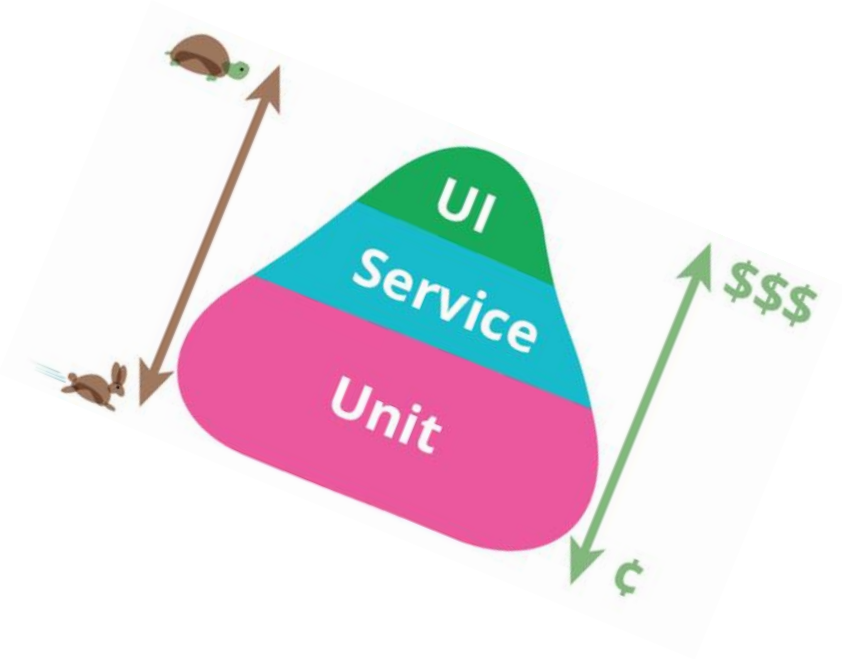
- Cristian (1991) und Hadzilacos, Toueg (1993) präsentiert
- durch Tanenbaum und van Steen zusammengefasst
  - **Crash Failure**: Ein Server stürzt unerwartet ab – hat bis zu diesem Zeitpunkt aber fehlerfrei gearbeitet. Wenn der Server abgestürzt ist, hört man nichts mehr von ihm.
  - **Omission Failure**: Dieser Fehler tritt auf, wenn der Server e.g. auf einen Request nicht antwortet. Eine Ursache könnte sein, dass der Server den Request nie bekommen hat.
  - **Timing Failure**: Dieser Fehler tritt auf, wenn die Antwort außerhalb eines definierten Zeitfensters ist.
  - **Response Failure**: Der Fehler tritt auf, wenn die Antwort des Server inkorrekt ist.
  - **Arbitary Failure**: auch bekannt als Byzantine Failures (Server schickt gelegentlich falsche Antwort) → einer der schwierigst zu erkennenden Fehler

Warum Fehlertypen?

- Sollten auch im Test berücksichtigt werden
  - Wie verhält sich das System, wenn eine Komponenten nicht antwortet, falsch antwortet, zu spät antwortet?
- https://github.com/Netflix/chaosmonkey
  - Chaos Monkey randomly terminates virtual machine instances and containers that run inside of your production environment. Exposing engineers to failures more frequently incentivizes them to build resilient services.

Wiederholung: Die Testpyramide





Unit-Test

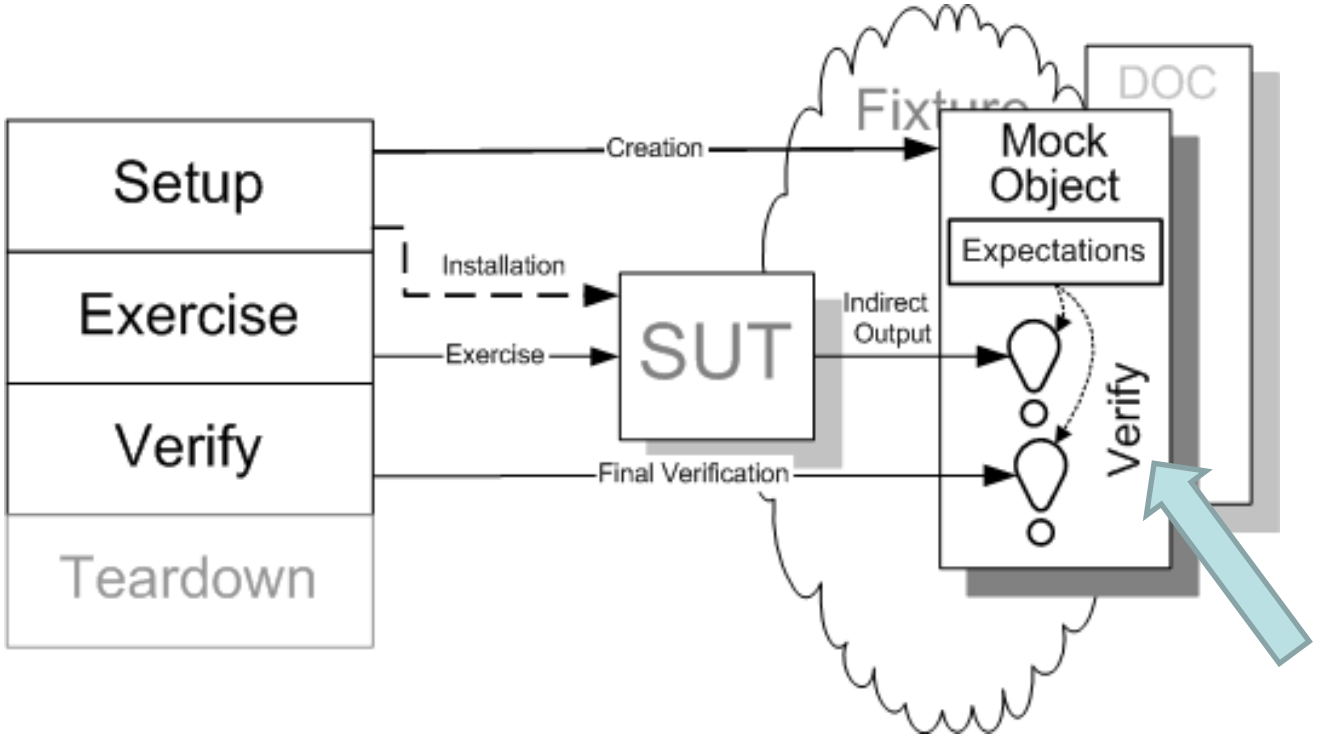
- Meist auf Klassen-Ebene
- Abhängigkeiten werden „gemocked“
- Daher auch wichtig: Abhängigkeiten mockbar machen (Stichwort Interfaces und Dependency Injection)
- Siehe auch http://xunitpatterns.com/
- In .NET 2 wichtige Pakete:
  - Moq (https://github.com/moq/moq)
  - xUnit (https://github.com/xunit/xunit)
  - Es gibt viele Alternativen

Exkurs: Mock vs. Stub?

- Unterschied: Mock & Stub
- Faustregel: Ein Test sollte immer einen Mock haben und kann durchaus mehrere Stubs haben („Focus on One Thing at a Time“)

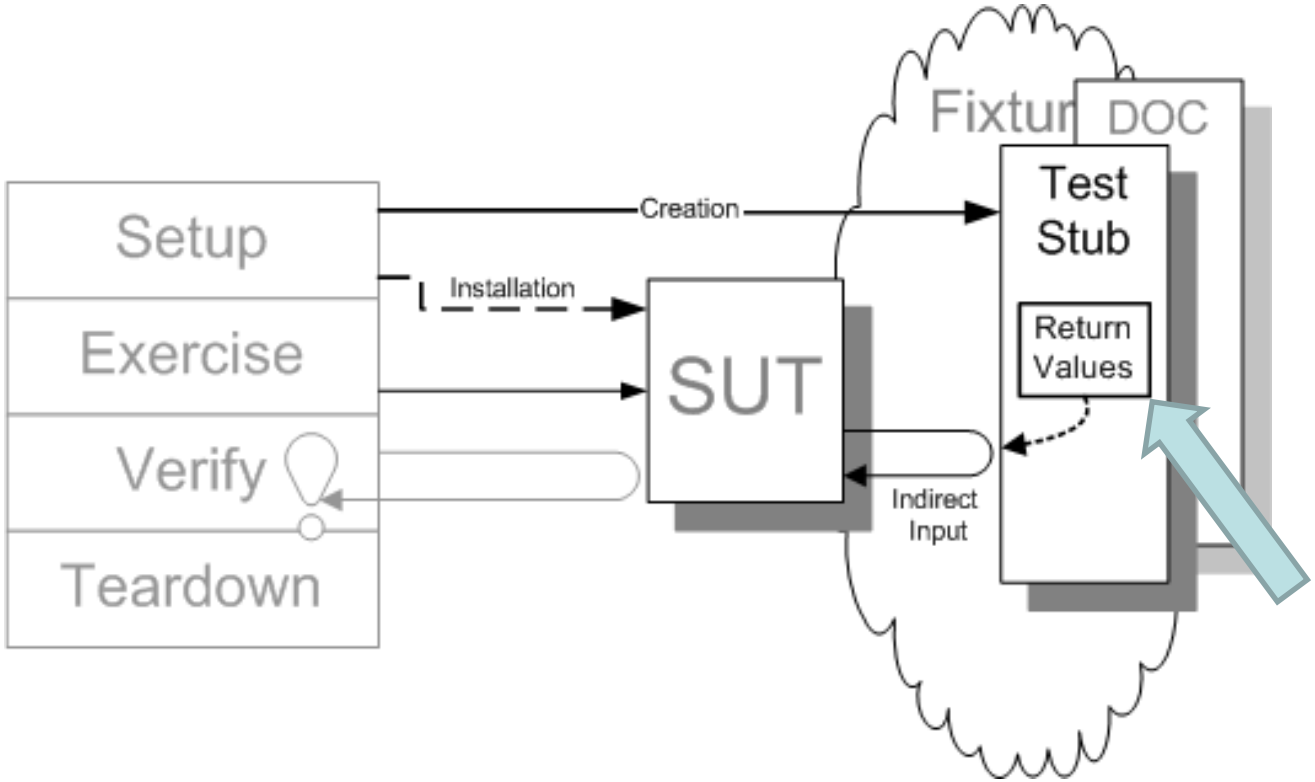
Mock

- Wir sind am Verhalten interessiert: E.g. Wurden die Methoden in der erwarteten Reihenfolge aufgerufen?



Stub

- „Ersetzt“ uns eine Abhängigkeit – uns ist aber egal, was dort aufgerufen wurde und ob etwas aufgerufen wurde



Beispiel

```
public double Method1(int value1, int value2)
{
    if (value2 == 0)
    {
        throw new ArgumentException("Second value should not be 0", nameof(value2));
    }

    var convertedValue1 = _classB.ConvertValue(value1);
    var convertedValue2 = _classB.ConvertValue(value2);
    var correctionValue = _classC.GetCorrectionValue(value1);

    return convertedValue1 / convertedValue2 + correctionValue;
}
```

Beispiel

```
public class MainTests
{
    private readonly IContainer _container;
    private readonly Mock<IClassC> _mockClassC;

    public MainTests()
    {
        _mockClassC = new Mock<IClassC>();
        _container = new Bootstrapper().BuildContainer(_mockClassC.Object);
    }

    [Fact]
    public void Instance_Method1WithValidParemters_CorrectResult()
    {
        // Given
        _mockClassC.Setup(i => i.GetCorrectionValue(It.IsAny<int>())).Returns(5.6);

        var instance = _container.Resolve<IClassA>();

        // When
        var result = instance.Method1(2, 4);

        // Then
        Assert.Equal(5.85, result);
    }
}
```

Dependency Injection

- Wir haben ein ähnliches Konzept auf höherer Ebene schon mit MEF kennen gelernt
- Wikipedia: „Entwurfsmuster bezeichnet, welches die Abhängigkeiten eines Objekts zur Laufzeit reglementiert“
- Beispiel: Autofac (https://github.com/autofac/Autofac)

```
var builder = new ContainerBuilder();

builder.Register(c => new TaskController(c.Resolve<ITaskRepository>()));
builder.RegisterType<TaskController>();
builder.RegisterInstance(new TaskController());
builder.RegisterAssemblyTypes(controllerAssembly);

var container = builder.Build();
```

Business Rule testen

- Viele Möglichkeiten …
  - Eine Möglichkeit Behavior Driven Development (BDD)
  - In .NET e.g. durch SpecFlow (https://github.com/SpecFlowOSS)
  - Meist in Gherkin Notation

```
Scenario: Add two numbers
    Given the first number is 50
    And the second number is 70
    When the two numbers are added
    Then the result should be 120
```

FH University of Applied Sciences TECHNIKUM WIEN

Specflow Bindings

```
[Given("the first number is (.*)")]
public void GivenTheFirstNumberIs(int number)
{
    _calculator.FirstNumber = number;
}
```

UI-Tests

- Entweder manuell (auch explorativ)
- Automatisiert:
  - Ranorex (geht auch für WPF)
  - Selenium
- Beide Methoden machen Sinn ….
- Meist werden damit mehrere Komponenten getestet …
  - Wir erinnern uns: Fehlertypen (Komponenten Interaktion)

Wichtig beim Testen

- Deployment!
- Test-Systeme sollten nicht „zusammen kopiert und konfiguriert“ werden
- Es soll klar sein, unter welchen Bedingungen getestet wurde
- Gute Testdaten (Qualität und Quantität) auch oft sehr herausfordernd
  - Tester testet mit 10 Datensätzen
  - Praxis: Millionen

Nicht funktionale Anforderungen testen

Lasttests mit Gatling

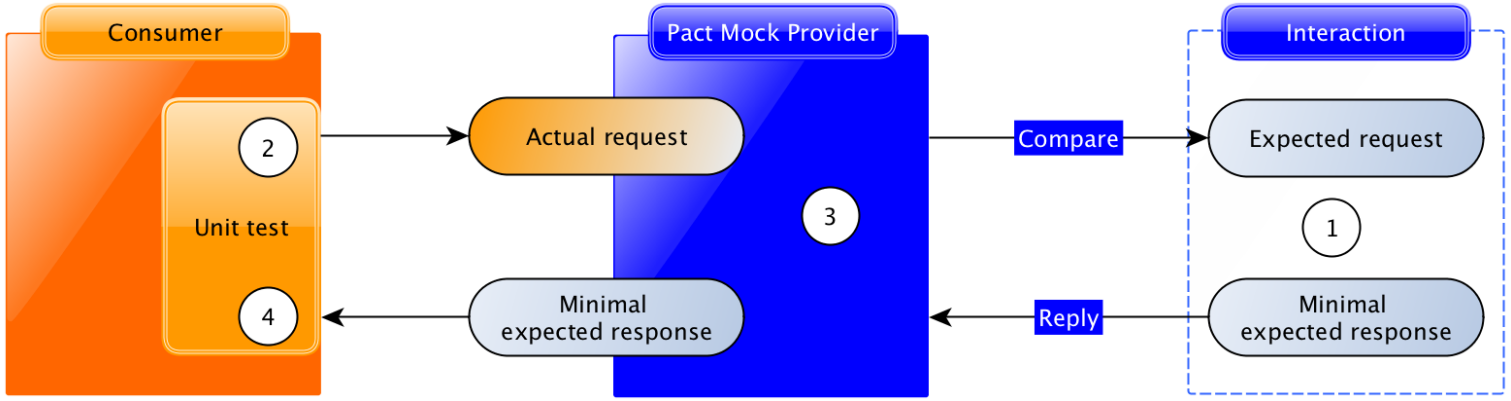
- „Gatling is a highly capable load testing tool. It is designed for ease of use, maintainability and high performance.”
- “Out of the box, Gatling comes with excellent support of the HTTP protocol that makes it a tool of choice for load testing any HTTP server.”
- https://www.youtube.com/watch?v=xtee7r5-ztY
- https://gatling.io/docs/current/advanced_tutorial/#advanced-tutorial

Teams parallel arbeiten lassen

- Oft stört die Abhängigkeit zu realen Services (e.g. Service ist noch nicht fertig und blockiert anderes Team oder Service hat wieder viele Abhängigkeiten – viel Aufwand um es zum Laufen zu bekommen)
- https://docs.pact.io/
- “Contract testing ensures that a pair of applications will work correctly together **by checking each application in isolation** to ensure the messages it sends or receives conform to a shared understanding that is documented in a "contract".”

Pactlo

- Man unterscheidet zwischen
  - **Consumer**: Komponente, welche die Funktionalität einer anderen Komponente nutzt
  - **Provider**: Service / Komponenten, welche eine Funktionalität anbietet



FH University of Applied Sciences TECHNIKUM WIEN

Integration / Deployment

Deployment

- Es gibt drei Begriffe, die am häufigsten verwendet werden:
  - Continuous Integration
  - ContinuousDelivery
  - Dev-Ops

Continuous Integration

- Ziel ist es, dass Entwickler so oft wie möglich deren Arbeit in das Software-Produkt integrieren
  - mindestens am Ende des Tages
- Nach jedem Integrieren (e.g. durch push / pull-request) erfolgt ein automatischer Build bzw. das Exekutieren der Tests
- Vermeidet das bekannte Merge Massaker
- Tools: https://jenkins.io/, https://azure.microsoft.com/de-de/services/devops/server/
- Wichtig ist, dass der Build schnell bleibt und Fehler sofort behoben werden
- Build-Scripts in .NET automatisieren: https://cakebuild.net/

Continuous Delivery

- Software wird so gebaut, dass sie jederzeit in die Production-Stage gehoben werden kann
- Per Knopfdruck kann Version X in die Production Stage deployed werden (erfordert hohen Grad an Automation)
- Schnelles Feedback durch Testautomation
- Im Kontrast zu Continuous Deployment erfolgt Deployment nur nach einem Gate (e.g. menschliche Freigabe)
- Vorteile:
  - Durch das häufige deployen werden kleinere Änderungen deployed (im Kontrast: yearly Big-Bang)
  - User-Feedback: Da schnell im Real-Betrieb → schnelles Feedback und man baut nicht ewig – möglicherweise etwas sinnloses

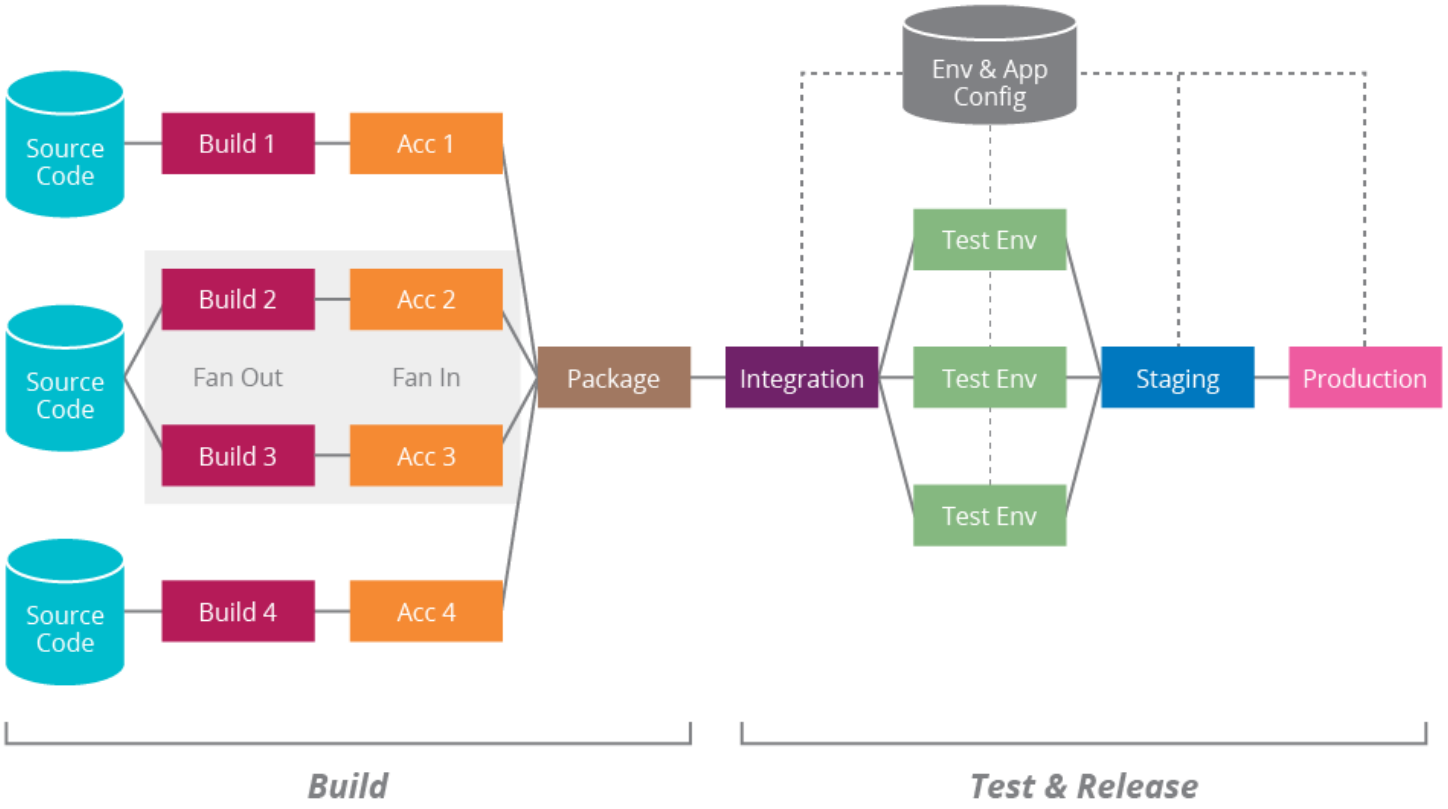
DevOps

- Versucht die Kluft zwischen Entwicklung und Operations zu stopfen
- Ist weniger ein Tool Thema als ein **Kultur / Mindset** Thema im Team
- Entwickler haben oft die Denkweise „Software bauen und über die Mauer werfen – Operation ist mir egal“ → in DevOps: Entwickler kümmert sich auch um Teilaspekte der Operation / Maintenance
  - E.g. durch automatisiertes Deployment, verbessertes Logging, Monitoring / Maintance Schnittstellen
- Fazit: Kollaboration

Stages

- Man schiebt die Software oft durch Stages → Einbahnstraße – man kann aber auch entscheiden, dass es nicht release-würdig ist
- Ein Beispiel für Stages:
  - Development: Unit-Tests, UI-Tests, Integration Tests, …
  - Staging: Performance & Load-Tests
  - User-Acceptance: Ad-Hoc Smoke Tests
  - Pre-Release: Disaster Recovery Tests (Umgebung sehr, sehr ähnlich dem Release)
  - Release: Penetration-Tests, System & Performance Monitoring
- Die wichtigsten sind: https://en.wikipedia.org/wiki/Deployment_environment#Environments

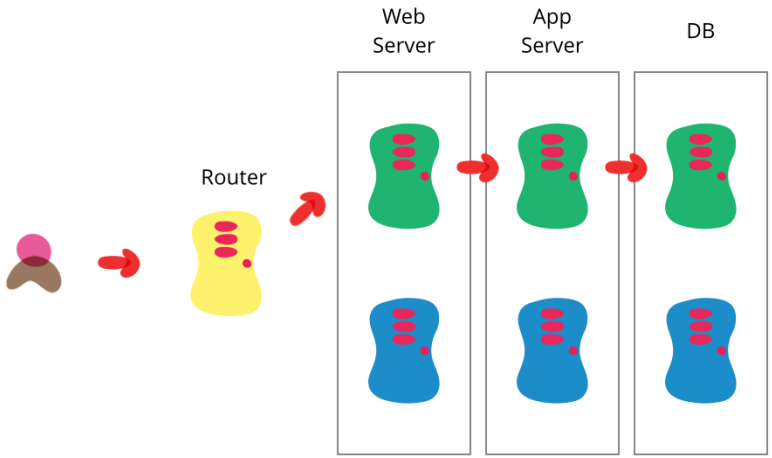
Stages



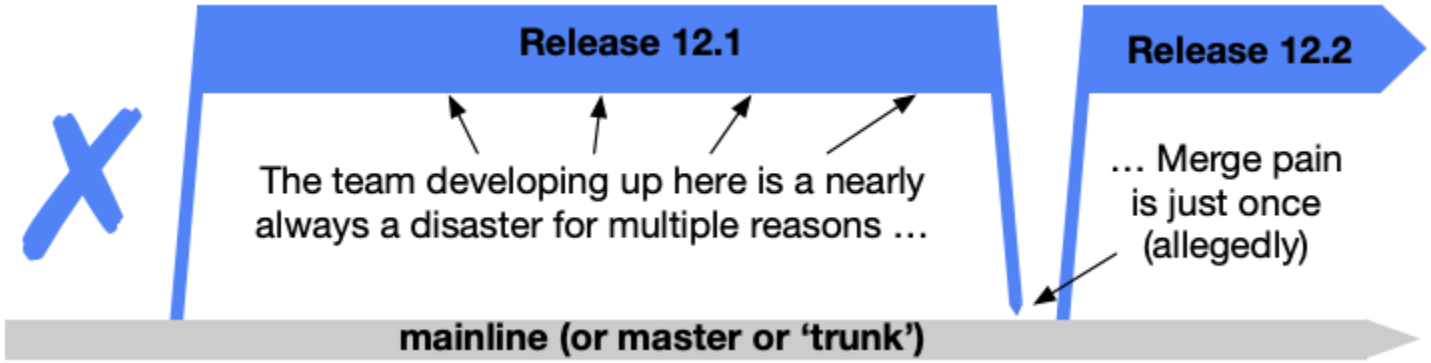
Quelle: thoughtworks

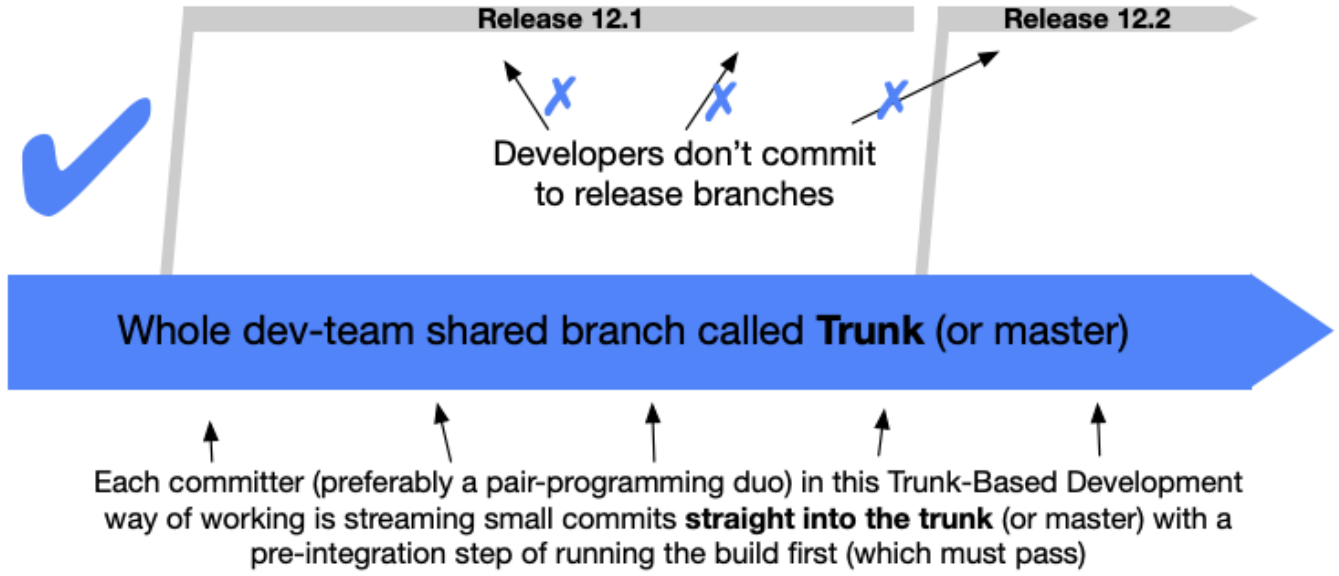
Blue-Green-Deployment

- Große Herausforderung: Automatisches Deployment
- Von der letzten Test-Phase → Production – und das möglichst unterbrechungsfrei
- Blue-Green-Deployment macht dies durch 2 Production-Systeme: Dadurch auch schnelles **Rollback** bei Problemen
- Herausforderungen: Datenbanken (Daten sollen ja nicht verloren gehen → und Schema kann sich ändern)



Trunk Based Development



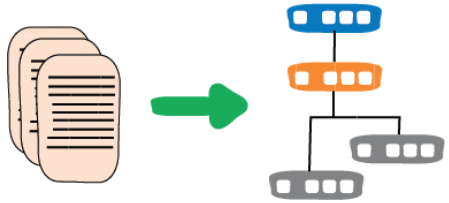


Trunk Based Development

- „Trunk-Based Development will always be **release ready”**
- Wichtig ist ein hoher Grad an Automatisierung
  - Vor allem Tests & Deployment
- “Quality-Gates” vor dem Commit: der Entwickler kann nicht einfach in die Mainline einchecken:
  - Einchecken: Es laufen automatisch Tests
  - Pull-Request: Code-Review durch Kollegen

Snowflake Server

- Server brauchen Updates und Konfiguration
- Durch manuelle Tätigkeit → Server mit der Zeit unterschiedlich
- Die Konfiguration ist schwer zu reproduzieren
  - Hat man ein Hardware-Problem → schwer einen neuen Server mit dem selben Stand schnell aufzusetzen
- Diskimages schaffen keine Abhilfe → Konfiguration ist trotzdem „versteckt“ enthalten
- Wenn man nach 2 Jahren die Konfiguration ändern will → die Ist-Konfiguration ist meist schwer zu verstehen



Infrastructure As Code

- Alte Welt: Server werden durch „Klicks“ konfiguriert
  - Person kann krank werden bzw. vergisst Schritte
  - Bei Neuaufsetzen / Aufsetzen einer anderen Stage (e.g. weitere Test-Stage) → alle Schritte nochmals?!
- Besser: Umgebung (Server, Netzwerk, …?) per Source-Code konfigurieren
  - Sourcecode kann in Git versioniert werden
  - Dadurch reviewed werden
  - Und beliebig oft wiederverwendet werden
- Tools: Ansible, Chef, Puppet

Cloud

- NIST definiert Cloud als Pool von Ressourcen (Server, Speicher, Netzwerk, Services, …) welche schnell provisioniert werden können
- Beispiele: Amazon AWS, Azure, SAP
- Unterschiedliche Servicemodelle
  - **Software as a Service (SaaS)**: Software, die von einem Dienstleister gehostet wird
  - **Platform as a Service (PaaS)**: zum Ausführen von eigenes entwickelten Software-Anwendungen
  - **Infrastructure as a Service (IaaS)**: Nutzungszugang von virtualisierten Computerhardware-Ressourcen wie Rechnern, Netzen und Speicher

Feature Toggles (aka Feature Flags)

- System verhalten kann geändert werden (durch Konfiguration), ohne den Code zu ändern
- Man will Branching so gut wie es geht vermeiden → siehe Trunk-Based-Development
- Durch Konfiguration kann man e.g.
  - Einen Menüpunkt ausblenden, weil die darunterliegende Funktionalität noch nicht fertig ist
  - Zwischen neuen und alten Algorithmus selektieren, weil der neue Algorithmus noch nicht ausgiebig getestet wurde
  - Das Feature nur einer Sub-Usergruppe zum Testen anbieten
- https://martinfowler.com/articles/feature-toggles.html