

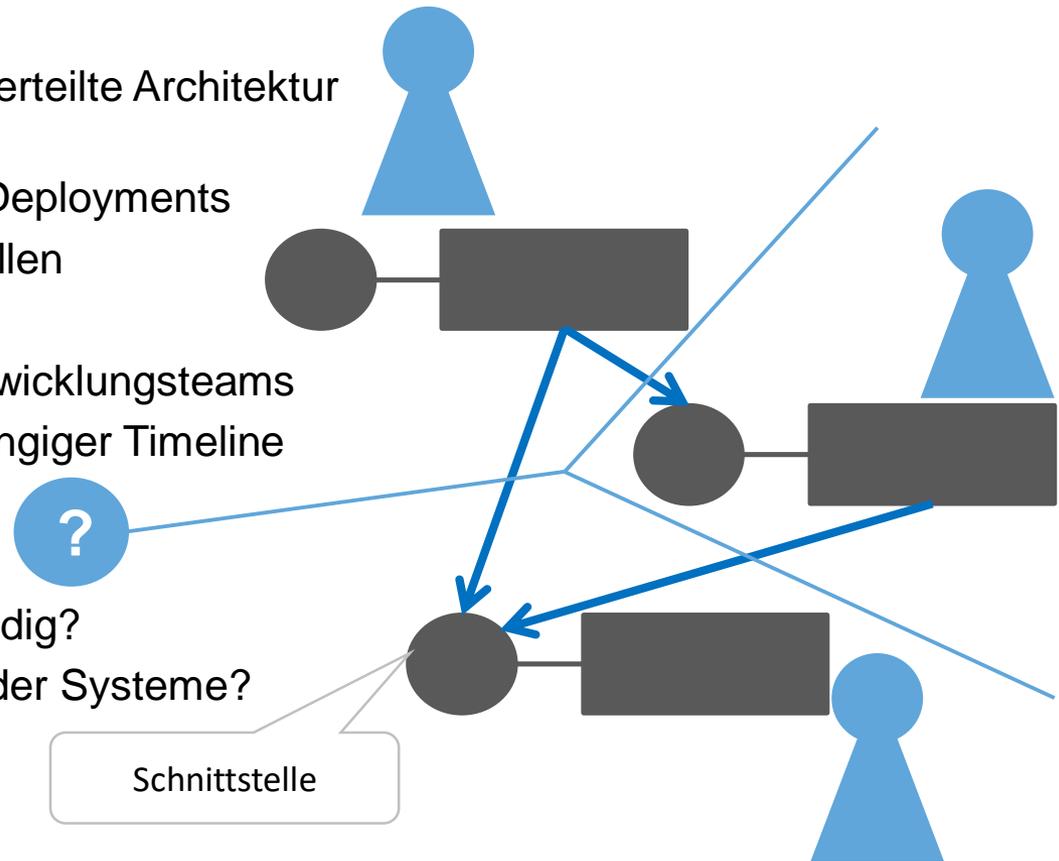
Consumer-Driven Contract Testing

Joachim Nelz, Principal Consultant, Architektur

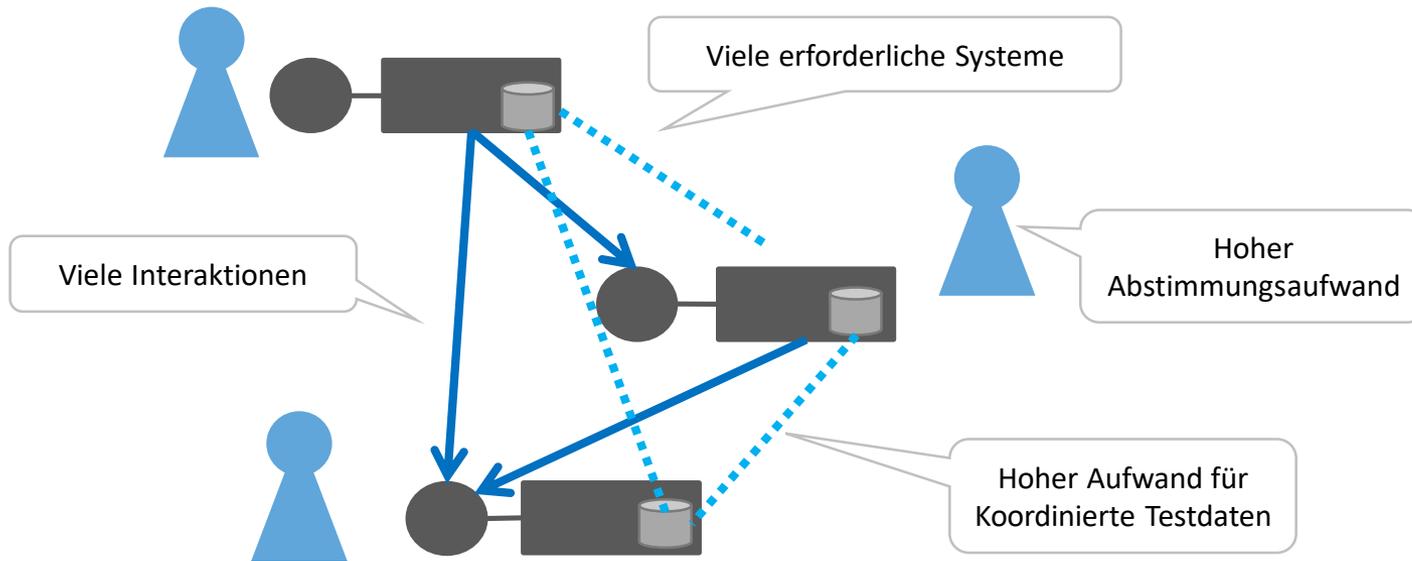
1	Intro
2	Basiskonzepte und Problemstellung
3	Methode: Consumer-Driven Contract Testing
4	Tool: Spring Cloud Contract
5	Fazit

Microservices bringen Komplexität

- Höherer Verteilungsgrad durch Verteilte Architektur als bei Monolithen
 - Zahlreiche SCS, als separate Deployments
 - Mehr out-of-process Schnittstellen
- Größere Unabhängigkeit der Entwicklungsteams
 - Kleine agile Teams mit unabhängiger Timeline
- Die Komplexität beherrschen
 - Wer ist für welche Tests zuständig?
 - Speziell: Das Zusammenspiel der Systeme?

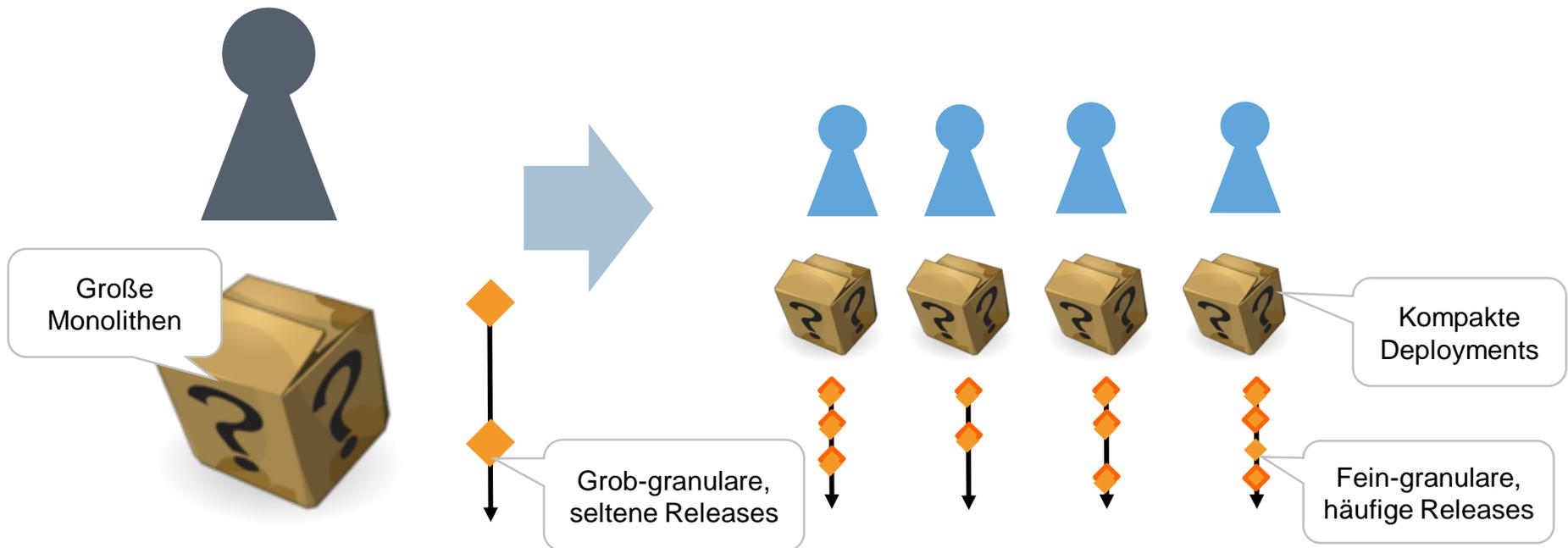


Herkömmliche System-Integrationstests sind nicht skalierbar



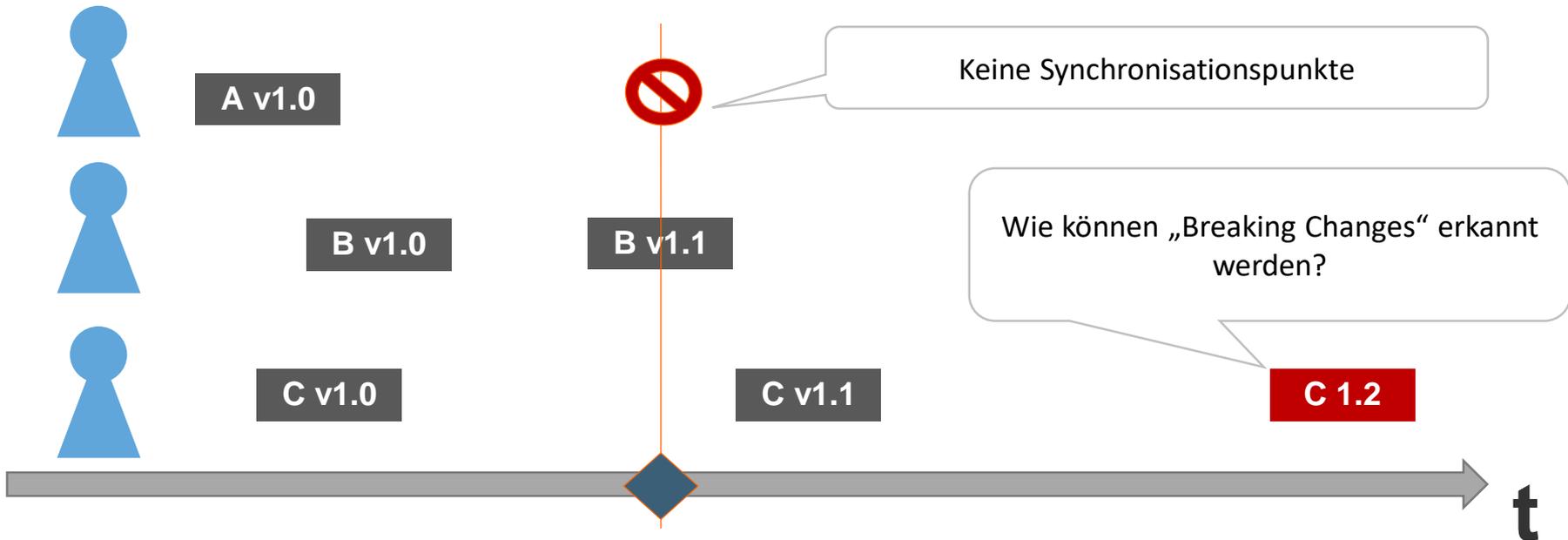
- Volle Integrationstests erfordern hohen Aufwand an Ressourcen und Abstimmung
 - Verfügbarkeit aller abhängigen Systeme – inklusive abgestimmter Testdaten
 - Wächst überproportional mit der Anzahl der Beteiligten
 - → **Keine Skalierbarkeit**

Organisatorische Skalierung mit Microservices



- Organisatorische Skalierung
 - Statt 1 Team aus 100 Mitarbeitern, 10 kompakte Teams aus 10 Mitarbeitern
 - Motiviert durch Agiles Vorgehen und Geschwindigkeit
 - Jedes Team liefert unabhängig

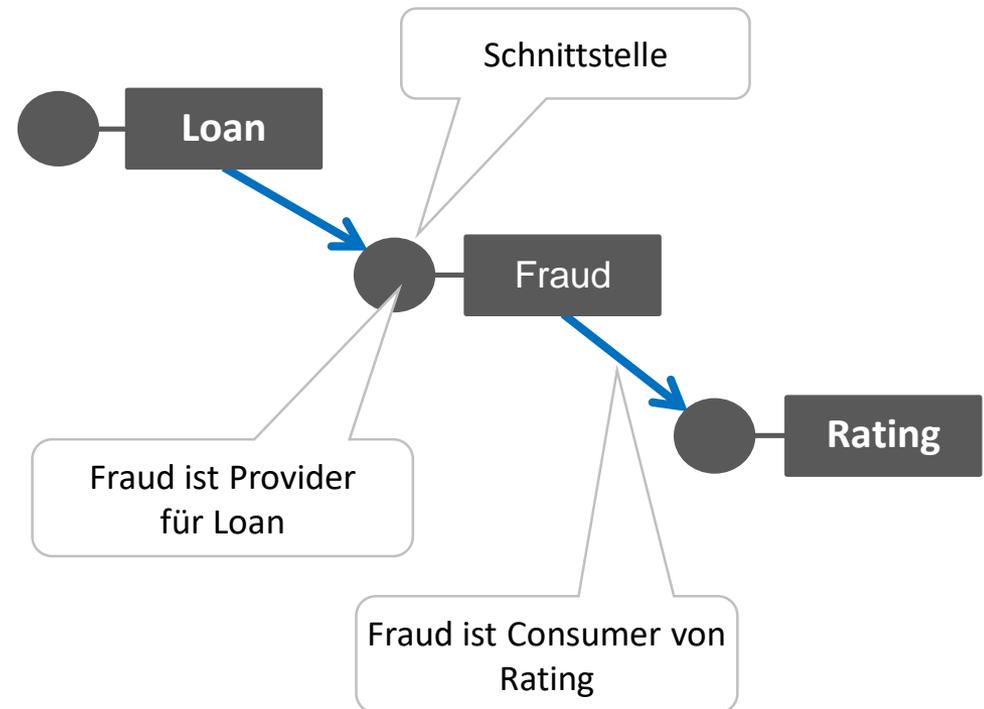
Geschwindigkeit durch Entkopplung



- Um schnell zu sein, sollen die Teams so unabhängig wie möglich arbeiten.
 - Daher ohne erzwungene Synchronisations-Punkte durch Integrationstests
- Trotzdem muss sichergestellt werden, dass die Änderungen verträglich sind.
 - Schwieriger als im herkömmlichen Weg, aufgrund höherer Entkopplung

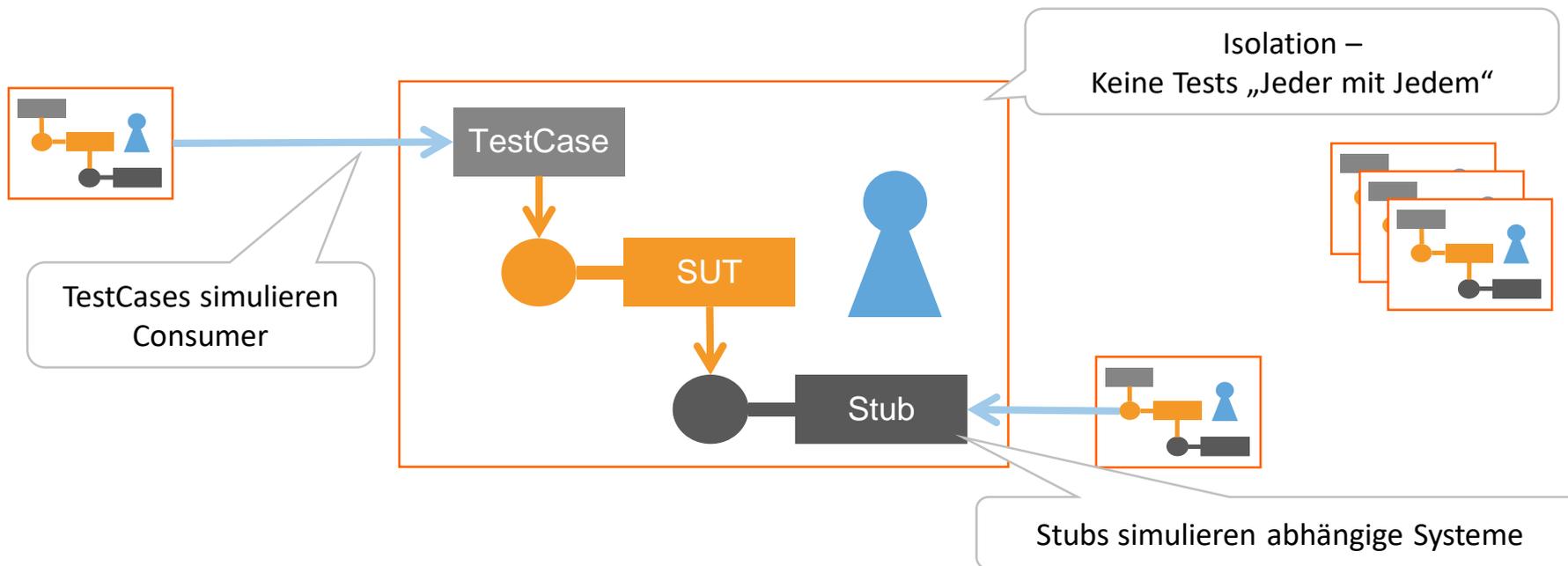
Rollen in Bezug auf Schnittstellen

- Provider
 - Bereitsteller einer Schnittstelle
- Consumer
 - Nutzer einer Schnittstelle



- Consumer / Provider sind **Rollen** in Bezug auf eine Schnittstelle
 - Ein Komponente kann Provider seiner Schnittstelle
 - und Consumer anderer Schnittstellen sein

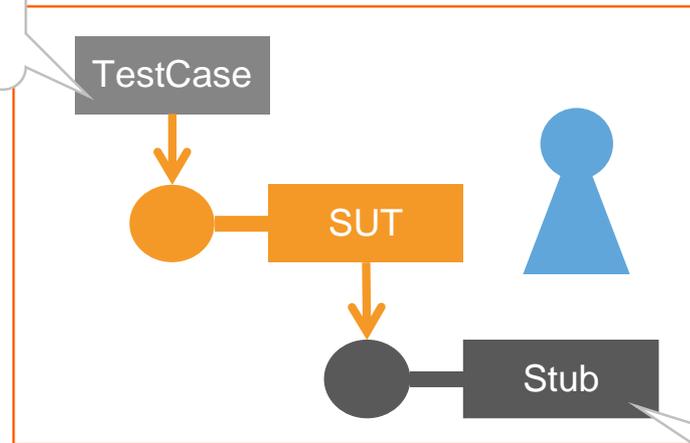
Isoliertes Testen statt Integrationstests



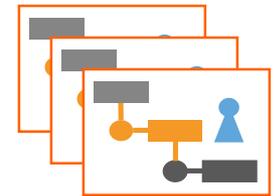
- Divide-et-impera beim Testen - Den „Monolith“ Integrationstest herunterbrechen
- **Isoliertes Testen** reduziert den Aufwand und Abhängigkeit zu anderen Teams
 - Schnittstellen zwischen Systemen sind entscheidend
 - → Orientierung am Contract der Schnittstelle

Isoliertes Testen – aber realistisch?

Wie kommt man zu „realistischen“ TestCases?

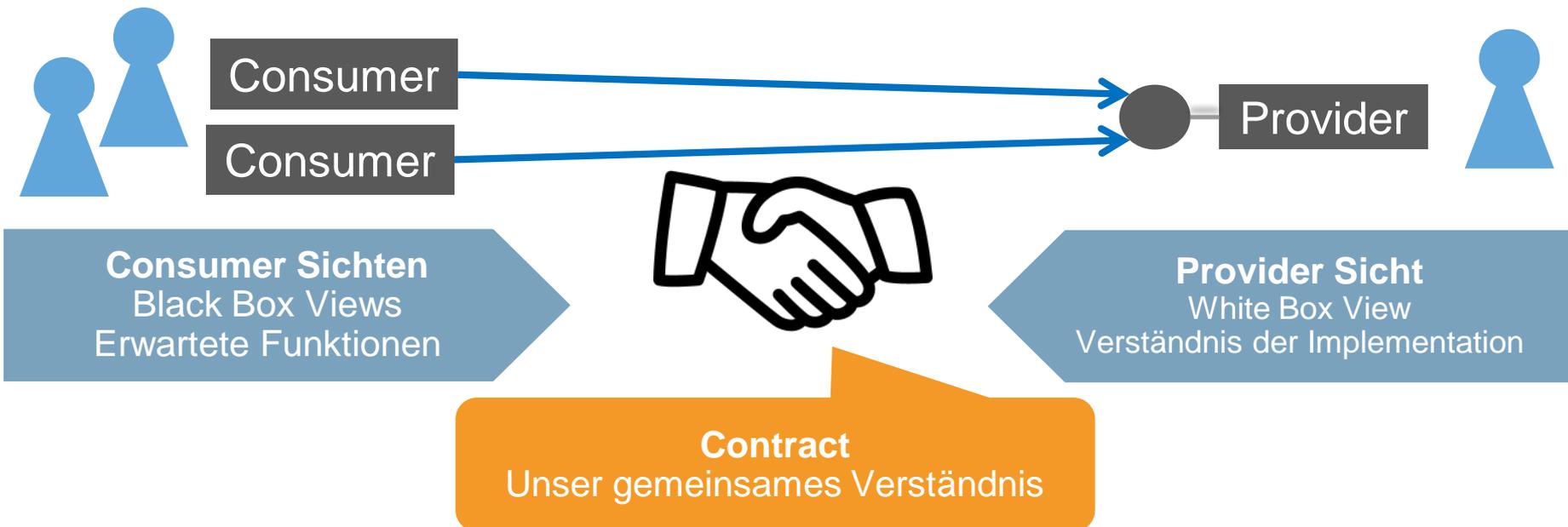


Wie kommt man zu „realistischen“ Stubs?



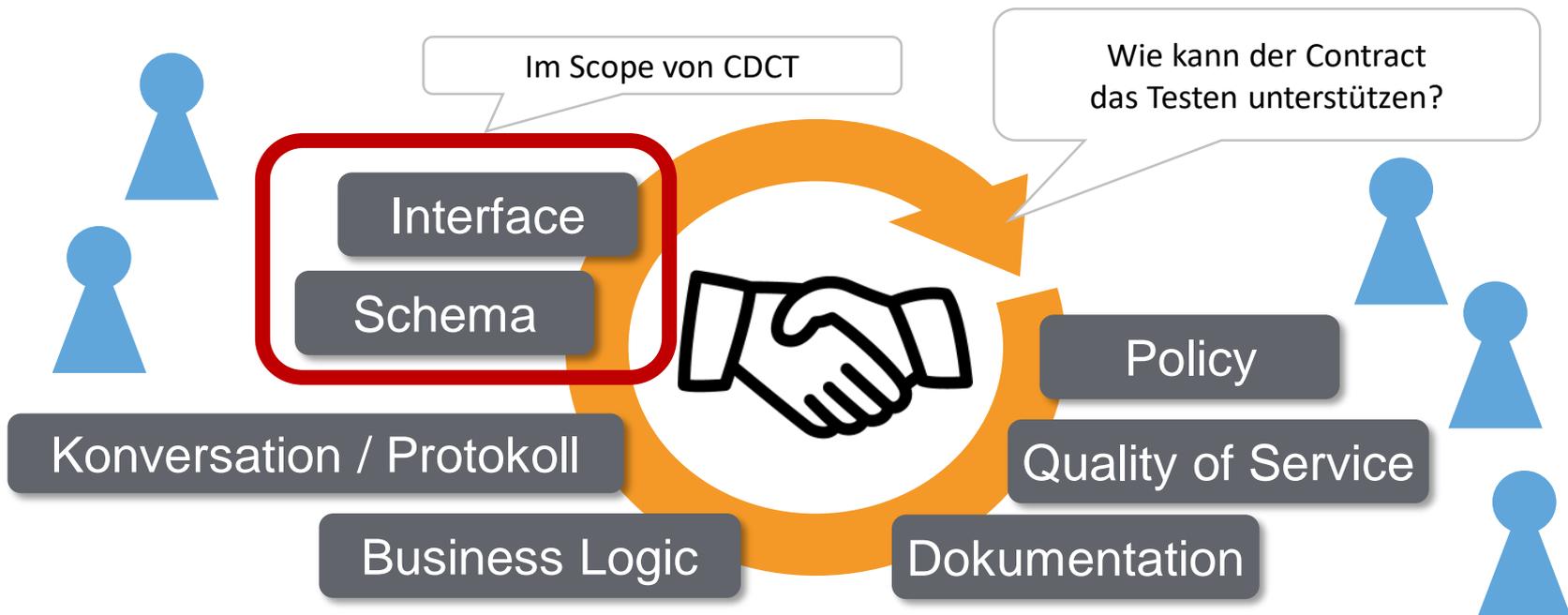
- Realistische Tests bedeuten:
 - **TestCases** spiegeln die Erwartungen der Service Consumer wieder
 - **Stubs** verhalten sich wie die simulierten Systeme, entsprechen somit den Erwartungen des Providers

Verschiedene Sichten konsolidieren



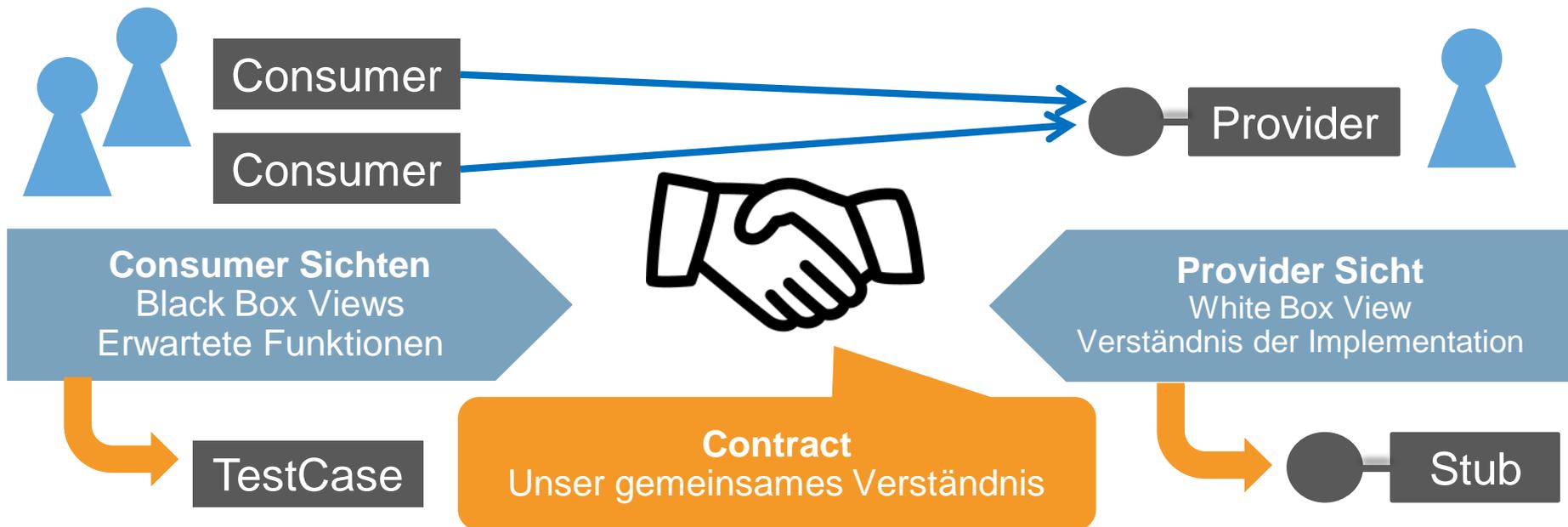
- Der Contract beschreibt das Verhalten einer Schnittstelle
- Consumer und Provider haben unterschiedliche Sichten auf eine Schnittstelle
- Das Konzept des Contracts dient dazu, ein gemeinsames Verständnis zu schaffen

Eine Schnittstelle hat viele Aspekte



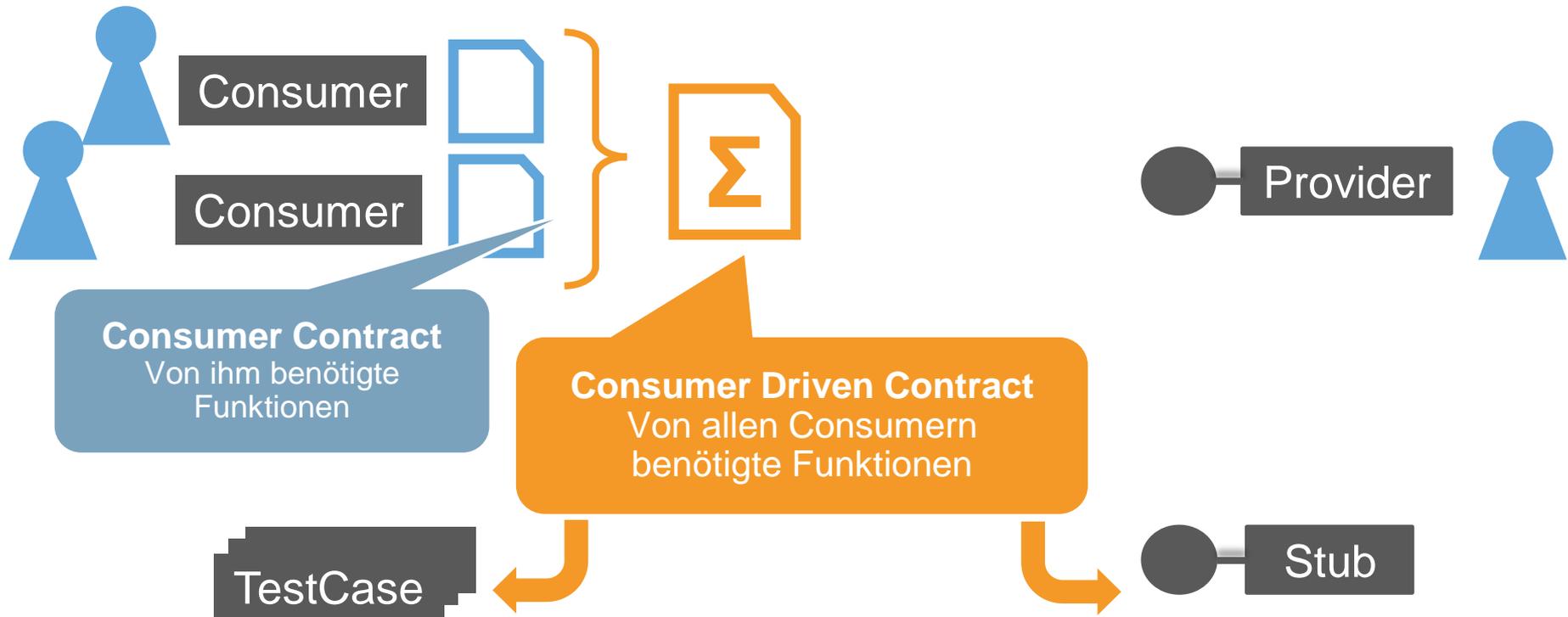
- Verständlichkeit: Alle Beteiligten brauchen einen einfachen, vollständigen Zugang
- Testbarkeit: Der Contract soll automatische Tests ermöglichen
- Tests sollten sich automatisch aus dem Contract ableiten lassen

Tests aus Sichten des Contracts ableiten



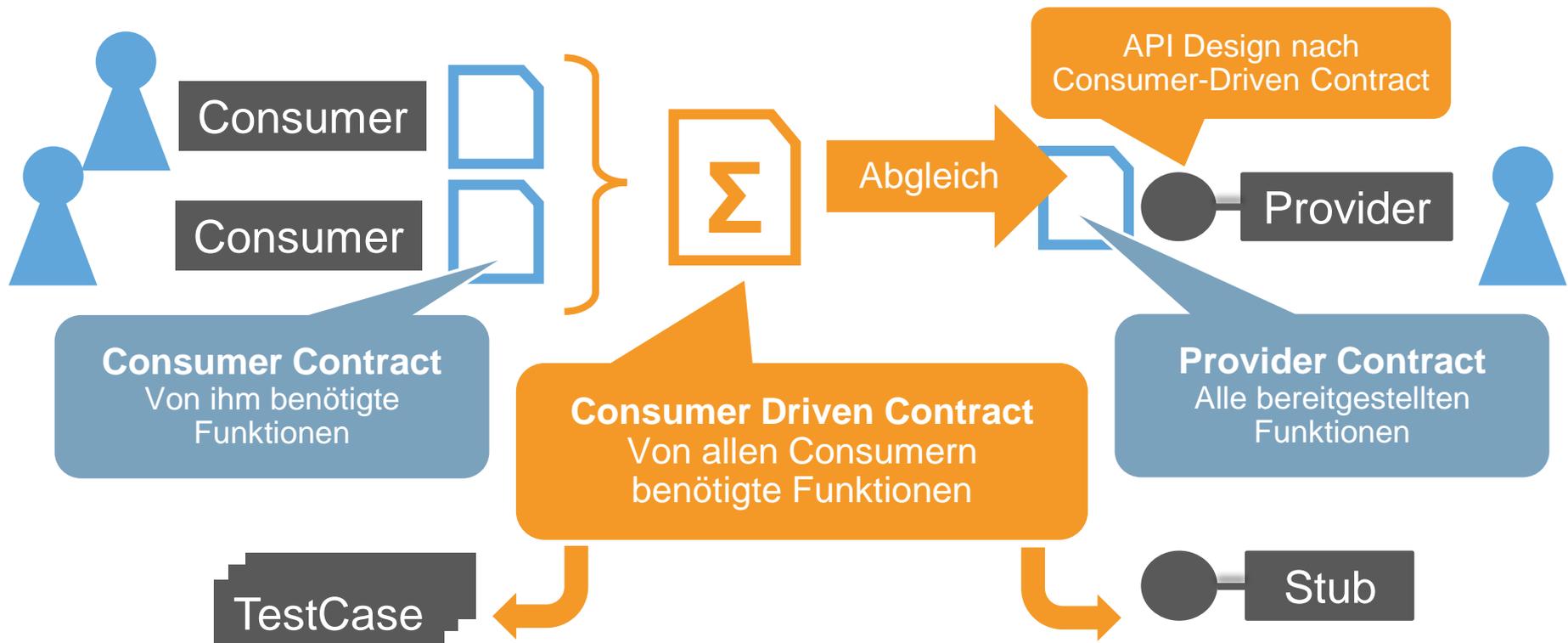
- Consumer Sichten führen zu realistischen TestCases, anhand erwarteter Funktionen
- Provider Sicht führt zu realistischen Stubs, anhand der bekannten Implementierung

Consumer Driven Contract als Angelpunkt



- Konsolidierung aller Consumer Sichten im Consumer Driven Contract
- Ableitung von TestCases und Stubs vom Consumer Driven Contract

Contract als Treiber der API Entwicklung



- Consumer-Driven Contract als Master beim Abgleich Consumer mit Provider
- Was von niemand benötigt wird, kann aus API entfernt werden

Contract Definition mit Spring Cloud Contract Verifier

- Definition des Contracts in DSL (Domain Specific Language)
 - Beispiele mit Request und Response

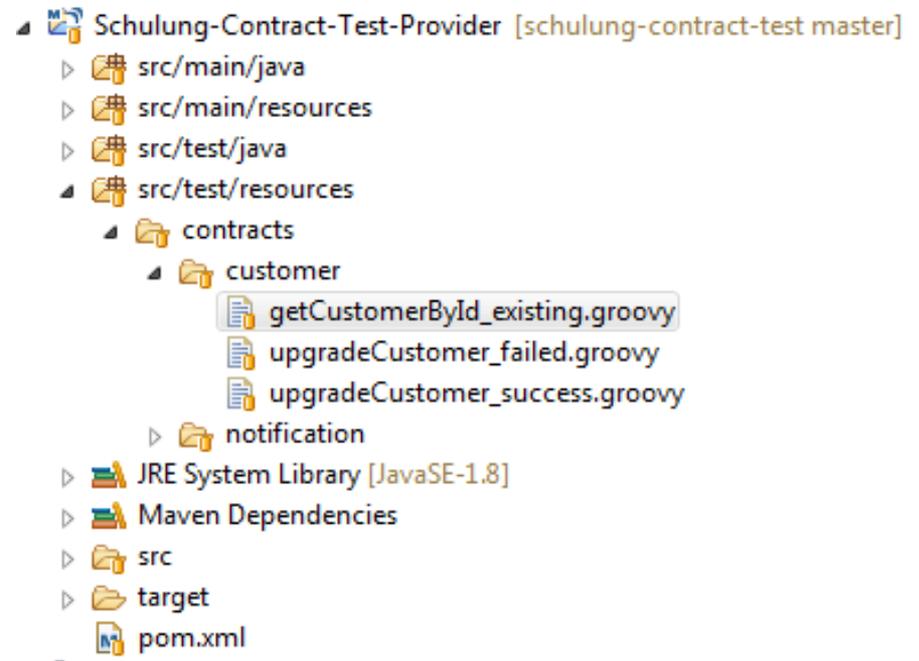
```
package contracts
```

```
org.springframework.cloud.contract.spec.Contract.make {
```

```
    request {  
        method 'PUT'  
        url '/customer/42'  
        <body>  
    }  
}
```

```
    response {  
        status 200  
        <body>  
    }  
}
```

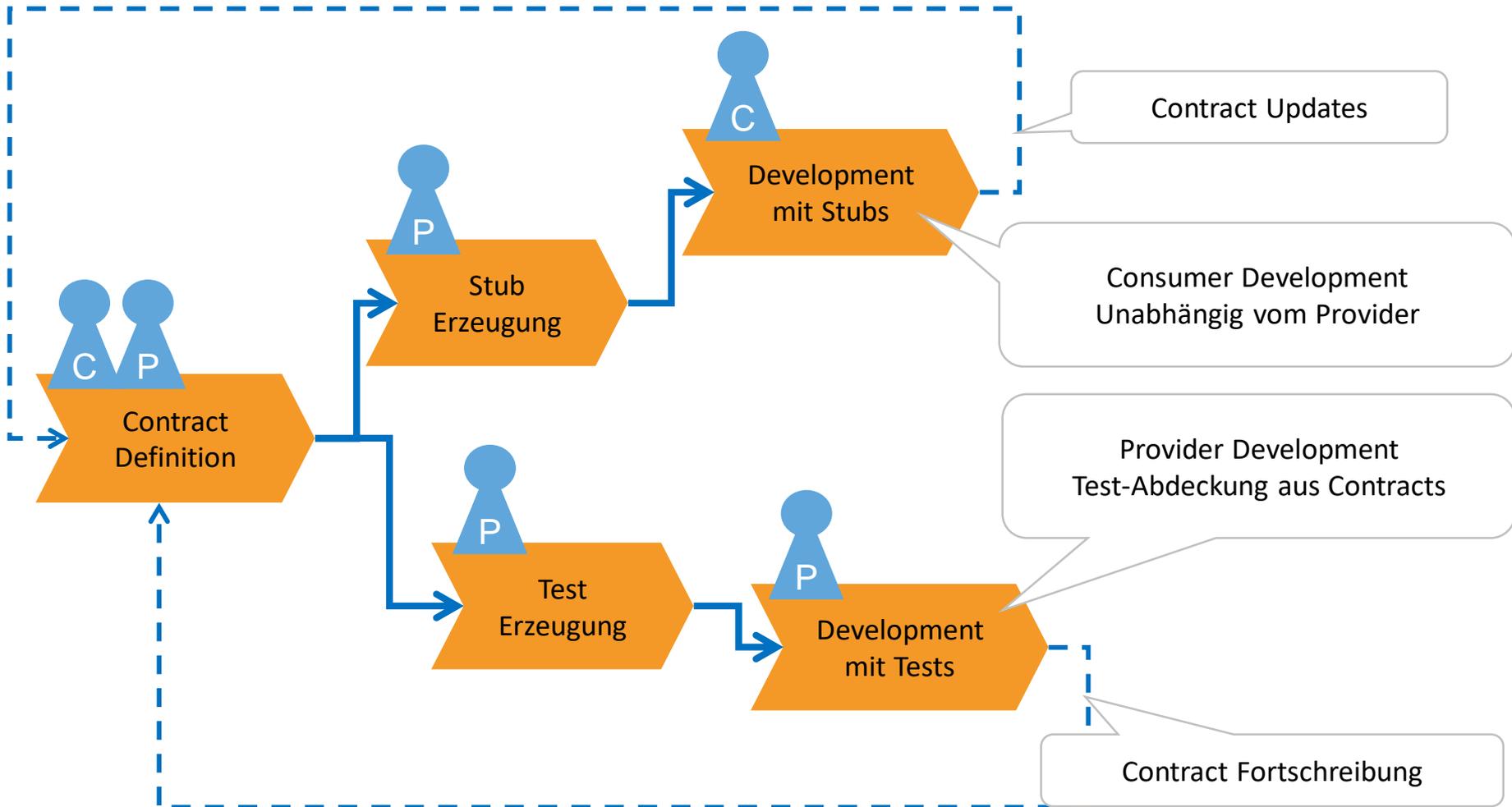
```
}
```



Beispiel in Spring Cloud Contract DSL

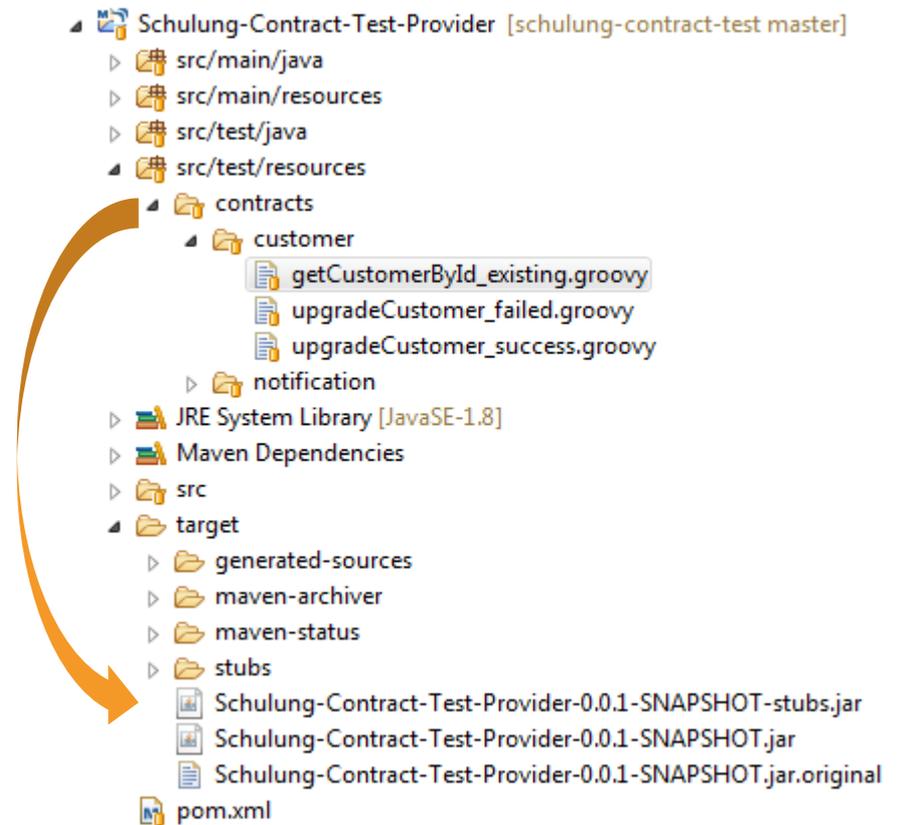
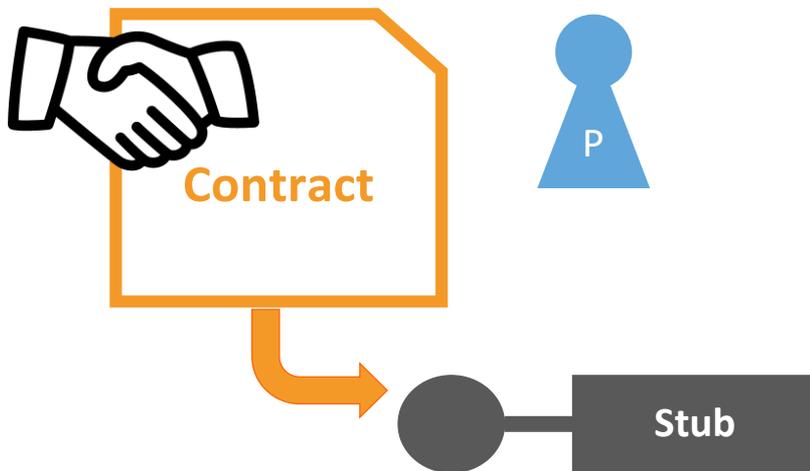
```
package contracts
org.springframework.cloud.contract.spec.Contract.make {
    request {
        method 'GET'
        url '/customer/42'
    }
    response {
        status 200
        body(
            id: 42,
            age: 55
        )
        headers {
            contentType(applicationJson())
        }
    }
}
```

Entkoppelte Workflows Consumer / Provider



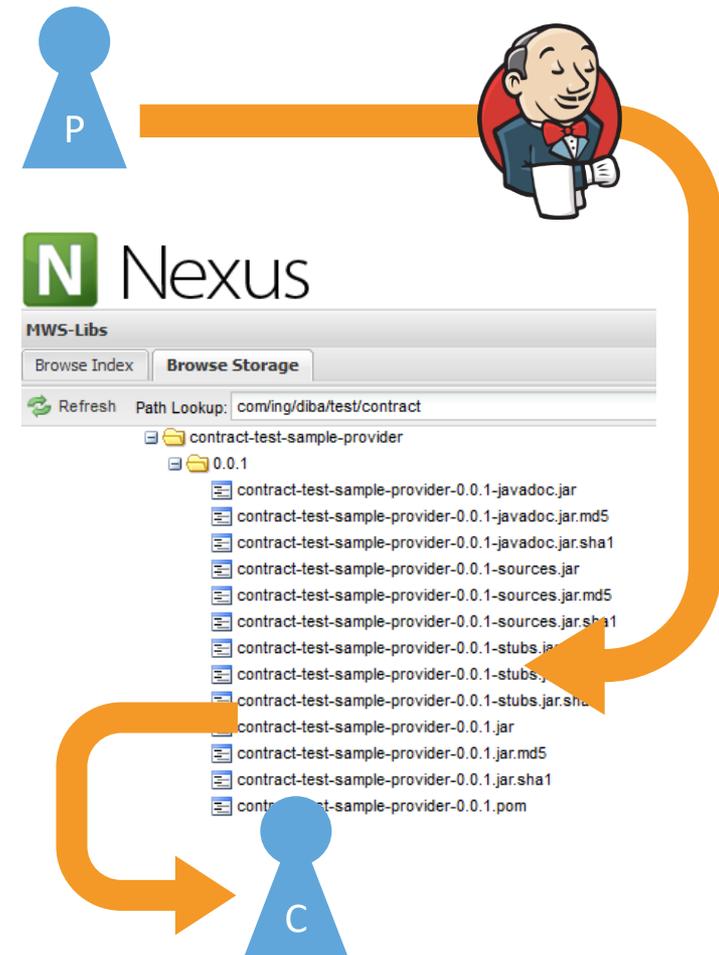
Provider: Stub Generierung aus Contract

- Generierung von Stubs aus dem Contract
 - mvn clean install -D skipTests



Verteilung der Stubs zum Consumer

- Provider stellt Contracts und Stubs auf Nexus bereit
 - Aktuelle Entwicklungs-Version als –SNAPSHOT
 - Produktions-Version als Release



- Consumer bekommt Stubs aus Nexus geliefert

Consumer: Verwendung der Stubs

- Konfiguration des StubsRunners
 - Referenzierung auf Maven-Koordinaten: Artefakt Name und Version des Producer-Stubs



The screenshot shows an IDE interface with the Package Explorer on the left and the application.yml file open on the right. The Package Explorer shows a project structure for 'Schulung-Contract-Test-Consumer' with folders for 'src/main/java', 'src/main/resources', 'src/test/java', and 'src/test/resources'. The 'application.yml' file in the 'src/test/resources' folder is highlighted. The code in the file is as follows:

```
1 server.port: 0
2 stubrunner:
3   ids: 'com.ing.diba.test:Schulung-Contract-Test-Provider:0.0.1-SNAPSHOT:stubs'
4   # repositoryRoot: http://nexus:8081/nexus/content/groups/MWS-Libs
5   work-offline: true
6   idsToServiceIds:
7     Schulung-Contract-Test-Provider: customer-service
```

- Ausführen der Tests
 - mvn clean test
 - StubRunner wird automatisch in der Test-Phase gestartet
 - Lädt generierte Stubs aus lokalem (work-offline: true) oder Nexus
 - Startet WireMock mit diesen Stubs als separaten Thread

Provider: Generierte Tests aus dem Contract

- Generierung von Tests aus dem Contract
 - mvn clean test



Package Explorer

- Schulung-Contract-Test [schulung-contract-test master]
 - Schulung-Contract-Test-Consumer [schulung-contract-test m...
 - Schulung-Contract-Test-Provider [schulung-contract-test ma...
 - src/main/java
 - src/main/resources
 - src/test/java
 - src/test/resources
 - contracts
 - customer
 - getCustomerById_existing.groovy
 - upgradeCustomer_failed.groovy
 - upgradeCustomer_success.groovy
 - notification
 - JRE System Library [JavaSE-1.8]
 - Maven Dependencies
 - src
 - target
 - generated-sources
 - generated-test-sources
 - contracts
 - com
 - ing
 - diba
 - contract
 - tutorial
 - provider
 - contract
 - CustomerTest.java
 - NotificationTest.java

CustomerTest.java

```

1 package com.ing.diba.contract.tutorial.provider.contract;
2
3 import com.ing.diba.contract.tutorial.provider.contract.CustomerBase;
4
5 public class CustomerTest extends CustomerBase {
6
7     @Test
8     public void validate_getCustomerById_existing() throws Exception {
9         // given:
10         MockMvcRequestSpecification request = given();
11
12         // when:
13         ResponseOptions response = given().spec(request)
14             .get("/customer/42");
15
16         // then:
17         assertThat(response.statusCode()).isEqualTo(200);
18         assertThat(response.header("Content-Type")).matches("application/json.*");
19         // and:
20         DocumentContext parsedJson = JsonPath.parse(response.getBody().asString());
21         assertThatJson(parsedJson).field("age").isEqualTo(55);
22         assertThatJson(parsedJson).field("id").isEqualTo(42);
23     }
24
25     @Test
26     public void validate_upgradeCustomer_failed() throws Exception {
27         // given:
28         MockMvcRequestSpecification request = given()
29             .header("Content-Type", "application/json")
30             .body("{\"reason\": \"account_created\"}");
31
32         // when:
33         ResponseOptions response = given().spec(request)
34             .post("/customer/44/upgrade");
35     }
36
37 }
                
```

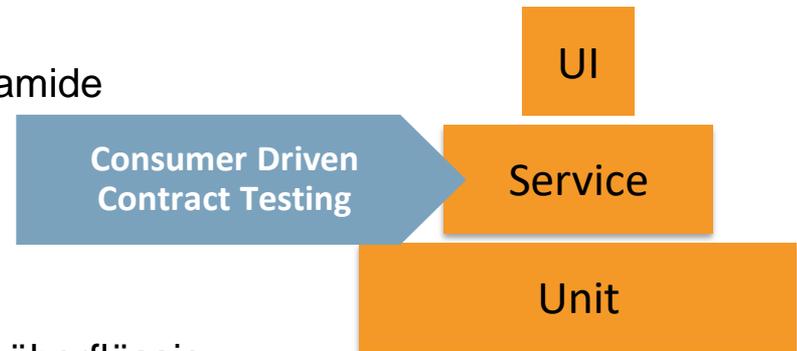
- Automatische Test-Ausführung

Warum Spring Cloud Contract Verifier?

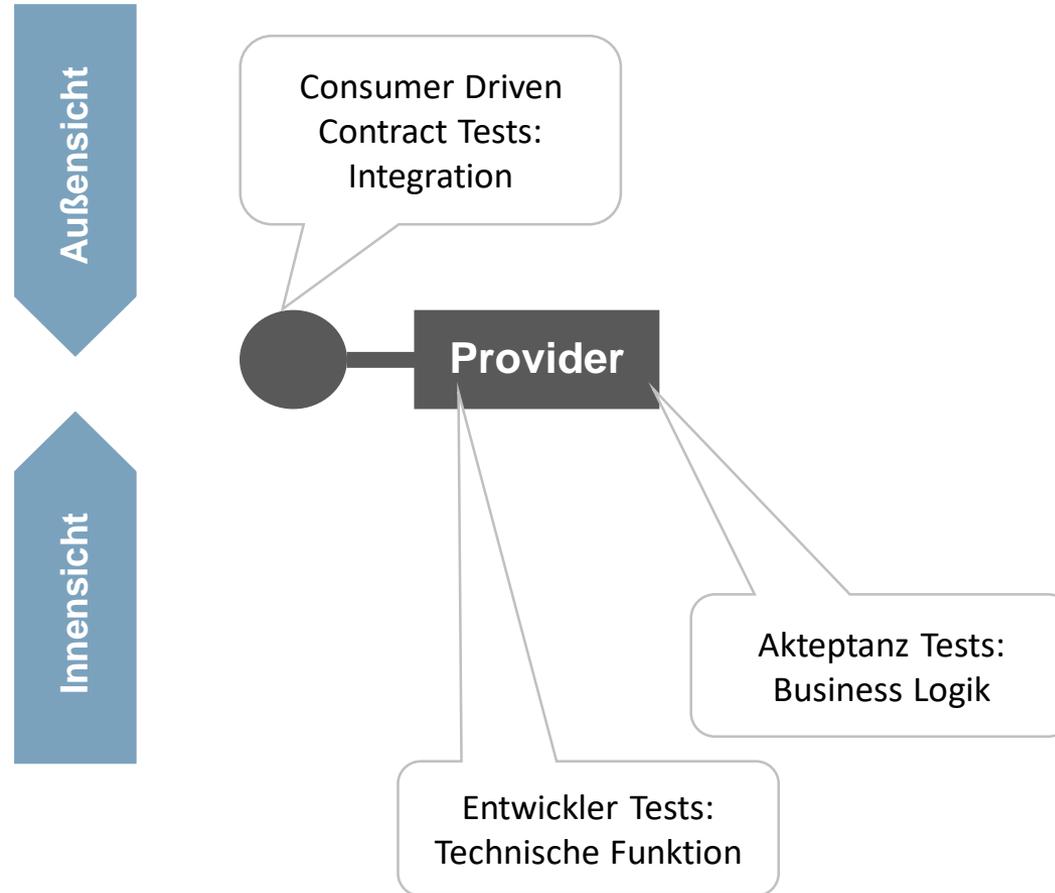
- Nahtlose Integration in Spring-Boot basierte Anwendungen, Maven Integration
- Kernfeatures
 - Test Generierung aus den Contracts – Das Kernstück des Contract Testens
 - Stub Generierung aus den Contracts
 - Verteilung der Stubs via Maven/Nexus
 - Automatisches Herunterladen und Starten der Stubs auf Seiten des Consumers
- Abgrenzung
 - Keine Generierung von Dokumentation (wie Swagger)
 - Keine Generierung von Clients (z.B. via Spring RestTemplates)

Consumer-Driven Contract Tests im Kontext

- Consumer Driven Contract Testing in der Testpyramide
 - Automatisierte Integrationstests
 - Technisch-orientiert, nicht fachlich
 - Black-Box Tests, also nicht vom Autor erstellt
- Ergänzen andere Test-Arten, machen diese nicht überflüssig
 - Fachliche Logik als Akzeptanztests, z.B. in FitNesse
 - Unit Tests (als White-Box Tests auf technischer Ebene), z.B. in JUnit
- Abgrenzung von CDCT gegen Fachliche Tests
 - Keine Business-Logik** testen – dafür ist FitNesse besser geeignet
 - Nur jeweils **typische Ausprägungen** jeder möglichen Request/Response Interaktion
 - Technischer Fokus** auf API Level: Methoden und Datenstrukturen
 - Auch Messaging ist im Scope: Versandte Messages müssen vom Empfänger lesbar sein



Verschiedene Tests, sich ergänzende Ziele



Nutzen des Consumer-Driven Contracts

- Reduzierter Aufwand / Overhead im Vergleich zu vollen Integrationstests
 - Isolierte Consumer-Driven Contract Tests ersetzen volle Integrations-Tests
 - Skalierbar auch bei steigender Anzahl der Systeme
- Unabhängige Entwicklung von Consumer und Provider
 - Consumer kann starten, sobald die Schnittstelle durch Contract definiert ist
 - Provider kennt die Erwartung seiner „Kunden“ /Consumer als automatische Tests
- Lebende Dokumentation
 - Contracts dokumentieren das Verhalten der Schnittstelle in Form von Beispielen
 - Contracts veralten nicht, sondern bilden die Basis der Tests dieser Schnittstelle