8 Teamorganisation verbessern

Um ein Legacy-System in einen besseren Zustand zu versetzen, muss mehr beachtet werden als »nur« die technischen und fachlichen Dimensionen von Domain-Driven Transformation. Häufig lohnt es sich, auch die Teamstrukturen und das Vorgehen zu überdenken. In vielen Organisationen ist agiles Vorgehen bereits gut etabliert, was uns immer wieder sehr freut. Ist das bei Ihnen noch nicht der Fall, ist jetzt der Moment, darüber nachzudenken, denn agile Transition und Domain-Driven Transformation gehen oft Hand in Hand.

In diesem Kapitel liefern wir Ihnen Konzepte und Argumente, wie Sie in Ihrer Organisation Diskussionen über die vorhandenen Teamstrukturen führen und das bei Ihnen sinnvolle Vorgehen für den Umbau Ihres Legacy-Systems einleiten können.

8.1 Software als soziotechnisches System

Nicht nur die Architektur eines Systems spielt eine wichtige Rolle, wenn es darum geht, das System über viele Jahre hinweg nutzbar, wartbar und erweiterbar zu halten. Entscheidend für den Erfolg sind auch die Organisations- und Kommunikationsstrukturen der Teams, die das System bauen und weiterentwickeln. Der Zusammenhang zwischen Teamstrukturen und Architektur wurde erstmals von Mel Conway formuliert:

> »[…] organizations which design systems […] are constrained to produce designs which are copies of the communication structures of these organizations« (s. [Conway 1968]).

Conway hat damit beschrieben, was man in vielen Organisationen beobachten kann: Die Grenzen von Softwaresystemen stimmen häufig mit den Grenzen zwischen Teams überein. Wobei die gelebten Teamgrenzen oft nicht mit den im Organigramm vorgegebenen Grenzen übereinstimmen. Weil dieser Zusammenhang so wichtig ist, wird Software heute nicht mehr nur als technisches System allein, sondern als **soziotechnisches System** betrachtet. Software existiert nicht unabhängig von den sie programmierenden und einsetzenden Menschen, deshalb sind Softwarearchitektur und Teamorganisation untrennbar miteinander verbunden.

Der Umbau einer Legacy-Software hin zu fachlichen Bounded Contexts kann daher nur gelingen, wenn auch Teams nach fachlichen Kriterien strukturiert sind.

Machen wir uns das an einem einfachen Beispiel mit einer Architektur klar, die aus drei Schichten besteht (s. Abb. 8–1): Frontend, Backend und Datenbank. In jeder dieser Schichten gibt es verschiedene Komponenten, in denen das System organisiert ist. Der Einfachheit halber haben wir sie hier mit Buchstaben benannt.

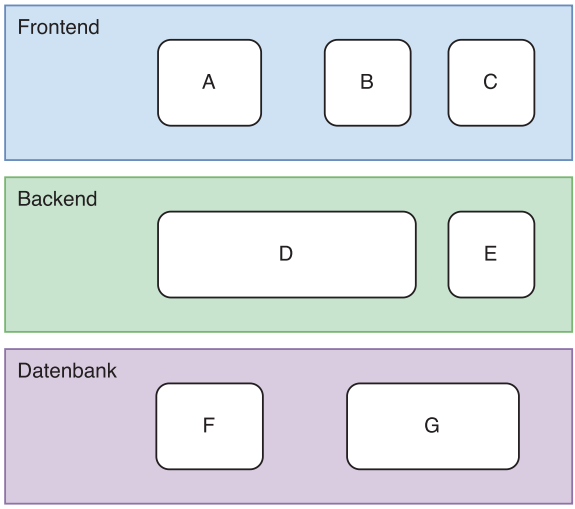

Abb. 8–1 *Schichten mit Komponenten*

8.1.1 Architekturschnitt/Teamorganisation horizontal

Teilt man diese Architektur nach technischen Kriterien auf Teams auf, so erhält man **Schichtenteams** (engl. **Layer Teams**). Der (erhoffte) Vorteil: Alle Expertinnen für ein Thema sind in einem Team versammelt. Hier in unserem Beispiel führt das jeweils zu einem *Frontend*-, *Backend*- und *Datenbank*-Team (s. Abb. 8–2).

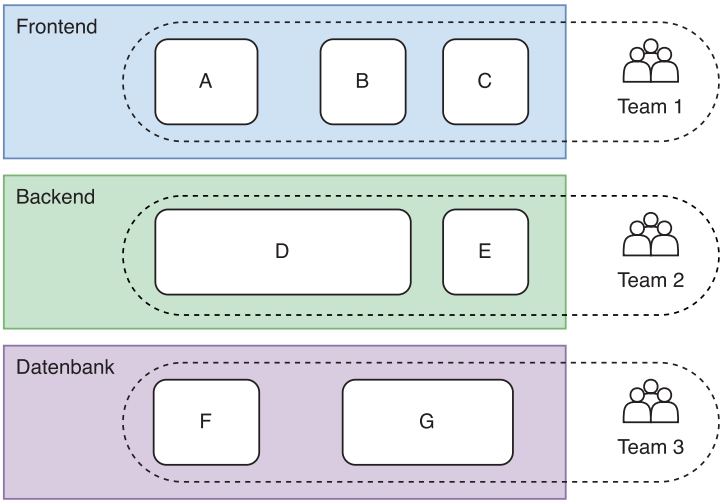

Abb. 8–2 *Führt zu Abhängigkeit: Schichten mit horizontalen Teams*

In so einer technischen Teamaufteilung ist das Frontend-Team davon abhängig, dass das Backend-Team die Frontend-Schnittstelle um benötigte Features erweitert. Das Backend-Team selbst hat auch keine Freiheit und muss seinerseits auf Anpassungen durch das Datenbank-Team warten. Neue Features betreffen in einer solchen Teamstruktur fast immer mehrere Teams, die bei der Umsetzung zeitlich voneinander abhängig sind und viel miteinander kommunizieren müssen, damit die Schnittstellen stimmen.

8.1.1.1 Spezialfall Framework-Team

Noch schwieriger wird diese Art der Teamorganisation in großen Unternehmen, in denen es häufig nicht nur Frontend-, Backend- und Datenbank-Teams gibt, sondern zusätzlich noch ein oder mehrere Framework-Teams, die für mehrere Softwareentwicklungen Basis-Features zur Verfügung stellen. Wenn eine Organisation eine solche Teamaufteilung gewählt hat, so sind Frontend-, Backend- und Datenbank-Team, die gemeinsam eine Applikation bauen, nicht nur voneinander, sondern auch vom Framework-Team abhängig.

Baut das Framework-Team nur technische Basisdienste, mag die Behinderung für die anderen Teams durch Wartezeiten auf neue technische Funktionalität nicht so sehr ins Gewicht fallen. Frontend-, Backend- und Datenbank-Team können trotz Verzögerung bei den technischen Basisdiensten einfach auf Grundlage der alten technischen Dienste weitere Features für die Anwenderinnen implementieren. Der Treiber für technische Änderungen ist in der Regel das Framework-Team selbst, das neue Technologien einführen will oder Sicherheits- bzw. Performance-Anforderungen umsetzen muss.

Gibt es in einer Organisation aber Framework-Teams, die fachliche Basisklassen und Dienste zur Verfügung stellen, z. B. die überall verwendete Kunden-Klasse, dann schlagen die Verzögerungen und die inhaltlichen Missverständnisse direkt auf die Teams durch, die neue Features für die Anwenderinnen bauen. Im Gegensatz zu den technischen Neuerungen erwachsen die fachlichen Neuerungen üblicherweise aus Feature-

Wünschen der Anwenderinnen. Das heißt, Frontend-, Backend- und Datenbank-Team treten mit einem Wunsch nach fachlicher Erweiterung und Veränderung an das Framework-Team für fachliche Basisdienste heran. Hier müssen einerseits die fachlichen Wünsche vermittelt werden und andererseits müssen Frontend-, Backend- und Datenbank-Team mit den zeitlichen Plänen des Framework-Teams klarkommen. Im Gegensatz zur Erweiterung der technischen Basisdienste sind in diesem Fall der fachlichen Basisdienste häufig die Teams die Treiber, die die Features für die Anwenderinnen bauen, also Frontend-, Backend- und Datenbank-Team. Einfach auf den vorhandenen technischen Basisdiensten aufzusetzen, ohne auf die Neuerungen zu warten, ist im Fall von fachlichen Erweiterungen häufig nicht möglich und verschärft die Abhängigkeit vom Framework-Team noch mehr.

8.1.1.2 Probleme mit dem horizontalen Schnitt

Mit so einer horizontalen Aufteilung hat man einerseits Teams, die ständig auf Lieferungen von anderen Teams warten, von denen sie abhängig sind. Andererseits fangen die Teams – Conway's Law folgend – ganz automatisch an, die Komponenten, an denen sie arbeiten, zu vereinheitlichen und zusammenzuführen.

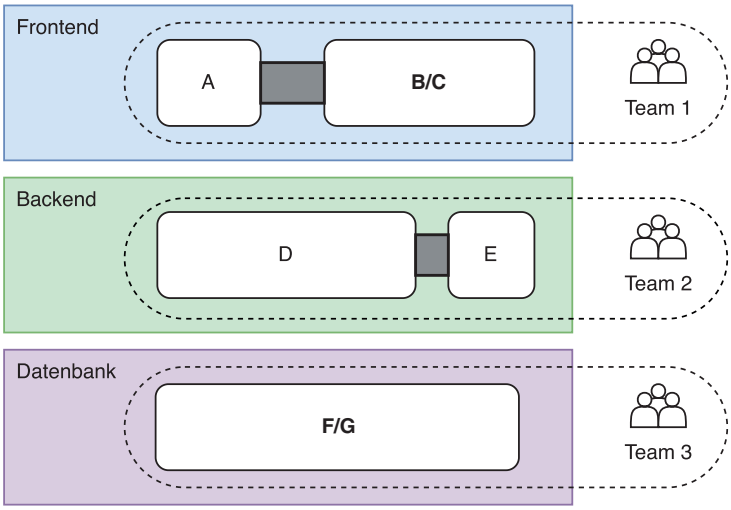

Abb. 8–3 *Problematisch: Schichten mit zusammenwachsenden Komponenten bei horizontal organisierten Teams*

In Abbildung 8–3 ist angedeutet, was Teams üblicherweise tun:

- Sie finden Gemeinsamkeiten in den von ihnen bearbeiteten Komponenten und legen sie komplett zusammen, wie B + C sowie F + G.
- Außerdem schaffen sie in ihren Komponenten gemeinsame Frameworks und Bibliotheken, die die einzelnen Teile weiter miteinander verbinden und aneinanderkoppeln, wie die Verbindung zwischen A und B/C sowie zwischen D und E.

Will man diese Effekte vermeiden, muss man seine Teams vertikal organisieren.

8.1.2 Architekturschnitt/Teamorganisation vertikal

In Abschnitt 2.3.1 im Grundlagenkapitel zu Domain-Driven Design haben wir erfahren, dass ein Bounded Context vertikal geschnitten sein sollte – also quer zu der technischen Schichtung. Folgt man der Logik von Conway, dass Architektur die Teamstruktur abbildet, dann müssen auch die Teams, die Bounded Contexts betreuen, vertikal geschnitten sein.

Dann brauchen wir in jedem Team Mitglieder mit Know-how aus allen Schichten und nicht »nur« z.B. Backend-Entwicklerinnen oder Frontend-Entwicklerinnen. In der agilen Sprechweise sagt man dazu, dass die Teams *cross-funktional* zusammengesetzt sein sollen.

Eine vertikale Aufteilung von Teams nach fachlichen Kriterien macht es möglich, dass ein Team für einen Bounded Context in der Software zuständig ist, der sich durch alle technischen Schichten von der Oberfläche bis zur Datenbank und den Frameworks zieht (s. Abb. 8–4).

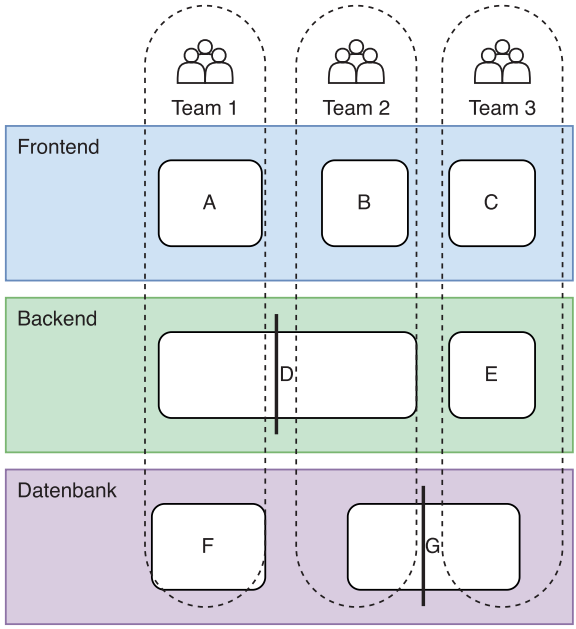

Abb. 8–4 *Macht Unabhängigkeit möglich: Schichten mit cross-funktionalen Teams*

Damit Team 1 und Team 2 wirklich getrennt entwickeln können, muss in unserer Architektur das Modul D in der Backend-Schicht in zwei getrennte Komponenten aufgeteilt werden. Auch in der Datenbank müssen die Datentabellen, die zusammen G ausmachen, aufgeteilt werden. Wie das geht, werden wir in den folgenden Kapiteln zu strategischer Domain-Driven Transformation erfahren.

8.1.3 Wie führt man diese Teamreorganisation durch?

Organisatorische Domain-Driven Transformation bedeutet oft, horizontal organisierte Teams (s. Abb. 8–2) zu vertikal organisierten Teams (s. Abb. 8–4) weiterzuentwickeln. Um den Teams einen Anhaltspunkt zu geben, wie die geplanten Veränderungen ablaufen werden, haben wir die folgenden soziotechnischen Refactorings[1] beschrieben:

- *Form Cross-Functional Team out of Layer-Team Members*[2]
- *Form Second Cross-Functional Team out of Partly Layer-Team and First-Team Members*[3]
- *Form Second Team out of Only Layer-Team Members*[4]
- *Assign Context to Existing Team*[5]

Organisatorische Domain-Driven Transformation geht oft mit strategischer Domain-Driven Transformation Hand in Hand. Für ein Beispiel verweisen wir daher auf Abschnitt 10.5.

In den meisten Organisationen ist diese einfache Sicht auf Architektur und Teamorganisation ein guter Ausgangspunkt bzw. ein gutes erstes Erklärmodell, um den Anstoß für Diskussionen über die Organisation zu geben. Um einen nachhaltigen Wandel in den Teamstrukturen zu erreichen, braucht es allerdings mehr. Das Buch von Matthew Skelton und Manuel Pais mit dem Titel *Team Topologies* (s. [Skelton & Pais 2019]) hat an dieser Stelle einen weit beachteten Beitrag geleistet. Im nächsten Abschnitt stellen wir den Ansatz vor und verknüpfen ihn mit DDD, sodass Sie beide Ansätze für die Veränderungen in Ihrem Unternehmen in Kombination verwenden können.

8.2 Team Topologies

Nach Skelton und Pais sollte es das Ziel einer Organisation sein, schlagkräftige und effiziente Teams zu haben, um möglichst viel Mehrwert für die Organisation zu erwirtschaften. Teams sind dann schlagkräftig und effizient, wenn sie bei der Softwareentwicklung in den *Flow* kommen und so neue Ideen und Features schnell in die Umsetzung und in Produktion bringen. Genau das, was sich jede Anwenderin wünscht: Eine neue Idee wird vom Team aufgenommen und ist in kurzer Zeit im Produkt zu finden.

---

1. Wir verwenden hier den Begriff »Refactoring« in einem sehr weiten Sinne. Alternative Begriffe wären: »Reorganisation« oder »soziotechnische Transformation«.
2. *https://hschwentner.io/domain-driven-refactorings/socio-technical/form-cross-functional-team-out-of-layer-team-members*.
3. *https://hschwentner.io/domain-driven-refactorings/socio-technical/form-second-team-out-of-partly-layer-team-and-first-team-members*.
4. *https://hschwentner.io/domain-driven-refactorings/socio-technical/form-second-team-out-of-layer-team-only*.
5. *https://hschwentner.io/domain-driven-refactorings/socio-technical/assign-context-to-existing-team*.

8.2.1 Arbeiten im Flow

Für dieses Arbeiten im Flow nennen Skelton und Pais zwei wichtige Voraussetzungen:

1. Man muss die **kognitive Last** für die Teammitglieder im Griff behalten, sodass sie sich auf ihre Aufgabe und deren zügige Umsetzung konzentrieren können. Ziel ist es, dass alle möglichst viel ihrer gedanklichen Energie darauf verwenden können, einen Wert für die Kundinnen zu schaffen – und ihre Energie nicht auf Nebenschauplätzen, wie Abstimmungen mit Abteilungen oder akzidentelle Komplexität (s. Abschnitt 1.3) aller Art verschwenden.
2. Es ist wichtig, den Teammitgliedern zur größtmöglichen **Autonomie** bei ihrer Arbeit zu verhelfen, indem man die Abhängigkeiten nach außen reduziert und lokale Entscheidungen ermöglicht. Bei hoher Autonomie können sie ihre Aufgaben in ihrem eigenen Rhythmus und zu dem Zeitpunkt, an dem sie Zeit haben, umsetzen, ohne durch Abhängigkeiten gebremst zu werden.

Zur Bewältigung der kognitiven Last und zur Schaffung von Autonomie müssen im Team-Topologies-Ansatz folgende Basisvoraussetzungen erfüllt werden:

- **Teamgröße**
  Ein Team sollte nicht mehr als fünf bis neun Personen umfassen, damit der Kommunikationsbedarf zwischen den Teammitgliedern nicht überhandnimmt. In großen Teams wird die kognitive Last durch mehr Kommunikation erhöht und auch dadurch, dass sie größere Stücke Software gemeinsam bearbeiten.
- Teams sollten **langfristig** zusammenarbeiten, um effiziente Arbeitsabläufe entwickeln und optimieren zu können. Arbeitet ein Team langfristig zusammen, so reduziert das die kognitive Last, weil man sich gut kennt und nicht immer wieder den Forming-Storming-Norming-Performing-Zyklus[6] durchlaufen muss – und hoffentlich auch regelmäßig die Schmerzpunkte in Retrospektiven ausräumt.
- Jedes Team sollte klare **Zuständigkeiten** und **Grenzen** für seine Arbeit haben, damit die Aufgaben und das Ziel für alle Teammitglieder verständlich und verfolgbar bleiben. Das schafft Autonomie und verringert die kognitive Last, weil weniger über unklare Randbedingungen nachgedacht werden muss und Entscheidungen schneller getroffen werden können.

8.2.2 Arten von Topologien

Aufbauend auf diesen Grundlagen beschreiben Skelton und Pais vier verschiedene Arten von Topologien:

- **Stream-aligned Team**
  Diese Teams schaffen den Wert für die Kundin entlang des Value Stream. Sie sind cross-funktional zusammengesetzt, damit sie mit diesem Kompetenzmix schnell

6. Auch »Tuckman's stages of group development« genannt und zuerst beschrieben in [Tuckman 1965].

signifikante Inkremente liefern können. Schnell geht es allerdings nur, wenn diese Teams arbeiten können, ohne auf andere Teams warten zu müssen.

Diese Topologie sollte in einem Unternehmen am häufigsten anzutreffen sein. Im Zusammenhang mit DDD würde man ein solches Team an einem Bounded Context (oder mehreren) exklusiv arbeiten lassen, da so die Zuständigkeiten und Grenzen sehr klar bestimmt werden können.

Im Programmkino-Beispiel hatten wir in Abbildung 2–9 eine Context Map entwickelt. Unser Zielbild für die Arbeit an den verschiedenen Kontexten ist, dass wir, wie von *Team Topologies* vorgeschlagen, Stream-aligned Teams für die Kontexte haben. Die Softwareentwicklung für unser Kinosystem hatte bisher eine Teamorganisation mit Frontend-, Backend- und Datenbank-Team. In einem teamübergreifenden Workshop diskutieren wir mit den Entwicklerinnen und Architektinnen über eine neue Teamzusammensetzung mit cross-funktionalen Teams. Wir können vier cross-funktionale Teams bilden und bitten die Teams, sich selbst Namen zu geben. Schließlich diskutieren wir, welches Team am besten zu welchem fachlichen Kontext passt. Abbildung 8–5 zeigt das Ergebnis.

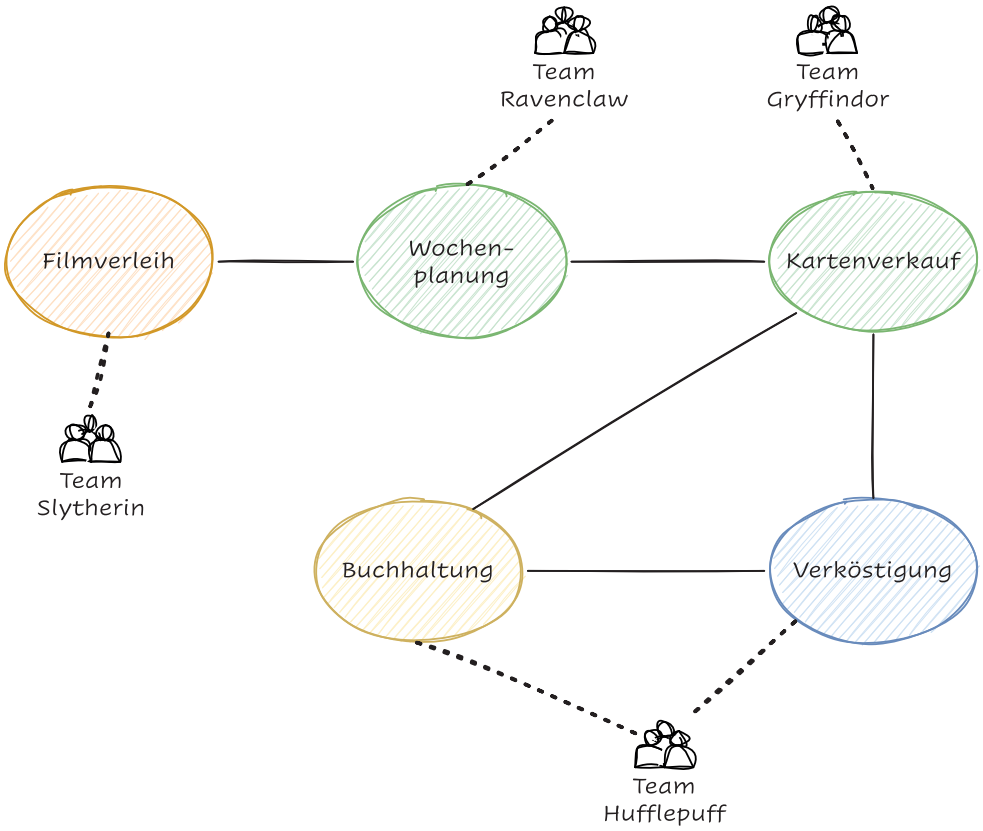

Abb. 8–5 *Stream-aligned Teams in der Context Map für das Kino*

Die beiden Kontexte »Kartenverkauf« und »Wochenplanung« haben wir in Abschnitt 2.2.3 als Core Domains identifiziert und in diesen beiden Kontexten wird es beim Umbau von CineSmall viel Arbeit zu erledigen geben. Deshalb sehen wir für diese beiden Kontexte jeweils ein eigenes Team vor. Die Supporting Domain »Verköstigung« und die Generic Domain »Buchhaltung« übergeben wir dem dritten Team. Dem externen Team für den »Filmverleih« geben wir ebenfalls einen Namen, auch wenn diese Entwicklerinnen nicht zu unserer Organisation gehören.

- **Platform Team**
  Ein solches Team stellt eine zugrunde liegende Plattform zur Verfügung, um die Stream-aligned Teams zu unterstützen. Entscheidend dabei ist, dass die Plattform Services anbietet, die die Komplexität der darunterliegenden Technologie oder Fachlichkeit vereinfacht und so die kognitive Last der Stream-aligned Teams verringern, um es ihnen zu ermöglichen, schnell zu arbeiten.

  Ziel eines Platform Team sollte sein, eine Plattform zu schaffen, die »gerade gut genug« ist. Man sagt auch, es solle die »Thinnest viable platform (TVP)« werden, also nicht nur eine dünne Plattform, sondern die dünnste Plattform, die möglich ist.

CineSmall ist aktuell noch ein Monolith und wir haben keine zugrunde liegende Plattform, die wir von einem eigenen Team bearbeiten lassen können. In Abschnitt 8.2.5 werden wir sehen, dass erst durch die Zerlegung von CineSmall ein Platform Team entstehen wird.

- **Complicated Subsystem Team**
  Diese Teams haben einen speziellen Auftrag für ein Teilsystem, das zu kompliziert ist, um von einem normalen Stream-aligned Team oder einem Platform Team bearbeitet zu werden.

  Wichtig ist, dass diese Teams bei Bedarf gebildet und nach kurzer Zeit wieder aufgelöst werden, denn auch hier ist das Ziel, die Stream-aligned Teams kognitiv zu entlasten. Diese Teams arbeiten an einem Teil eines Bounded Context, der später in den Verantwortungsbereich des Stream-aligned Team integriert wird, das für den Bounded Context zuständig ist.

Bei dem Bankbeispiel wird ein KI-Modul für die Bewertung der Kundenbonität beim Abschluss von Kreditverträgen eingeführt. Um dieses KI-Modul aufzusetzen, ist es sinnvoll, KI-Expertinnen für einen begrenzten Zeitraum in einem Team zusammenzuziehen, um diese Aufgabe zu lösen.

Enabling Team

Diese Teams unterstützen die Stream-aligned Teams sowie die Teams der beiden anderen Topologien beim Erlernen neuer Techniken und Technologien. Ziel ist es hier, dass die Stream-aligned Teams den Fokus auf ihre Aufgabe beibehalten, aber gleichzeitig ihre Effektivität steigern können.

Die Enabling Teams bestehen aus Expertinnen auf einem Gebiet oder einer Technologie. Sie haben Zeit, für die Stream-aligned Teams wichtige neue Themen auszuprobieren und an dort entstandenen Fragestellungen zu forschen.

Für den geplanten Umbau von CineSmall in eine Microservices-Architektur werden unsere Entwicklungsteams viel Know-how benötigen, das sie aktuell nicht haben. Moderne Technologien, wie Container und Messaging-Systeme, werden unsere Entwicklungsteams von entsprechenden Enabling Teams lernen.

Aus der Beschreibung der verschiedenen Team-Topologien wird bereits deutlich: Das gesamte System ist auf die Stream-aligned Teams ausgerichtet. Die drei anderen Teams agieren mit dem Ziel, die Stream-aligned Teams so zu unterstützen, dass diese mehr Autonomie und weniger kognitive Last erfahren. Im Endeffekt sind diese drei Teams Dienstleister für die Stream-aligned Teams und zumindest die Complicated Subsystem Teams sind Teams von kurzer Dauer.

Die Topologien werden mit einer eigenen Notation visualisiert, die in Abbildung 8–6 dargestellt ist.

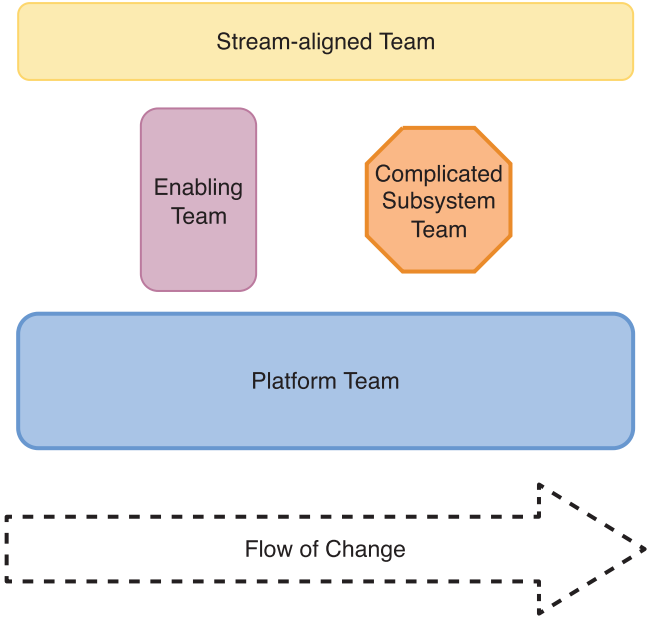

Abb. 8–6 *Grafische Darstellung der Arten von Topologien*

8.2.3 Arten von Zusammenarbeit

Der zweite Punkt, den der Team-Topologies-Ansatz beinhaltet, ist, wie diese Topologien zusammenarbeiten sollten. In der Beschreibung der einzelnen Teams oben wurde schon deutlich, dass die Teams zusammenarbeiten müssen, sonst könnte das Enabling Team die Stream-aligned Teams nicht unterstützen und die Complicated Subsystem und Platform Teams könnten den Stream-aligned Teams keine Arbeit abnehmen. Der Team-Topologies-Ansatz beschreibt drei grundsätzliche Arten der Zusammenarbeit:

- **Collaboration**
  Zwei Teams arbeiten eng zusammen, um schnell zu kommunizieren und Ideen auszutauschen. Diese Art der Zusammenarbeit führt zu viel Interaktion zwischen den Teams, bei der die Grenzen zwischen den Teams manchmal verschwimmen.
  Der Effekt ist, dass die kognitive Last steigt und die Autonomie der Teams sinkt. Deshalb ist Collaboration mit Vorsicht zu genießen und sollte nur kurzfristig stattfinden.

  Bei DDD findet sich diese Form der Zusammenarbeit unter dem Namen Partnership wieder (s. Abschnitt 2.3.2). Bei der Partnership können zwei Teams nur gemeinsam erfolgreich sein und ihre Grenzen verschwimmen über kurz oder lang.
- **X-as-a-service**
  Ein Team bietet eine Bibliothek, Komponente o. Ä. für ein oder mehrere andere Teams als Dienst an. Der Service soll ohne große Interaktion oder Input von dem anderen Team genutzt werden können.

  Auch diese Art von Zusammenarbeit kennt DDD. In Abschnitt 2.3.2 haben wir sie unter dem Namen Upstream-Downstream kennengelernt. Beim Context Mapping wird dadurch ein Machtverhältnis beschrieben, bei dem das Upstream-Team die Macht hat, die Schnittstelle festzulegen. Das Downstream-Team konsumiert diese Schnittstelle. Die X-as-a-service-Zusammenarbeit, die darauf abzielt, das Stream-aligned Team zu unterstützen und schneller arbeiten zu lassen, entspricht der Customer-Supplier-Version der Upstream-Downstream-Zusammenarbeit. Die eher diktatorische Interpretation der Upstream-Downstream-Zusammenarbeit von DDD, der Conformist, findet sich in den Team Topologies nicht wieder.

  Für die technische Umsetzung der X-as-a-service-Interaktion hat DDD auch einen Begriff: den Open-Host-Service ggf. mit Published Language.
- **Facilitating**
  Schließlich gibt es noch eine Art der Zusammenarbeit, die im DDD Context Mapping nicht beschrieben wird: das Facilitating. Diese Zusammenarbeit ist kurzfristig und findet statt, wenn ein Team ein anderes unterstützt und coacht. Typischerweise arbeiten Enabling Teams auf diese Weise mit den anderen Teams zusammen.

Die Interaktionsarten werden visualisiert, wie in Abbildung 8–7 dargestellt.

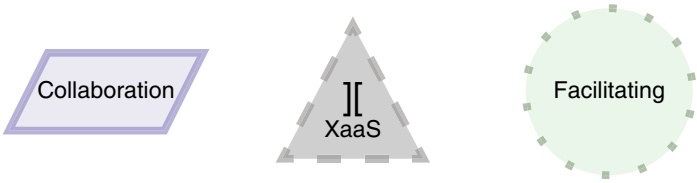

Abb. 8–7 *Grafische Darstellung der Interaktionsarten zwischen den Topologien*

Wie sehen die Topologien der Teams für unser Programmkino-Beispiel mit seinen Core Domains »Kartenverkauf«, »Wochenplanung«, seiner Supporting Domain »Verköstigung« und der Generic Domain »Buchhaltung« aus?

Wir haben die Context Map (s. Abb. 8–5) vorliegen, die angibt, wie wir die Teams zukünftig aufteilen wollen, und uns Wissen über die Softwareentwicklung angeeignet, wie sie aktuell funktioniert:

- Unsere geplanten Stream-aligned Teams arbeiten aktuell als ein Team an unserem Monolithen CineSmall, den wir zerlegen wollen.
- Es gibt ein externes Team Slytherin für den Filmverleih, das wir nicht in unsere Überlegungen mit einbeziehen werden.
- Unsere zukünftigen Stream-aligned Teams sollen moderne Technologien von einem Enabling Team lernen.
- Für die Generic Domain soll eine Standardsoftware eingekauft werden und die Standardsoftware soll über einen Anti-Corruption Layer mit CineSmall Daten austauschen.

Daraus entwickeln wir die Topologien in Abbildung 8–8.

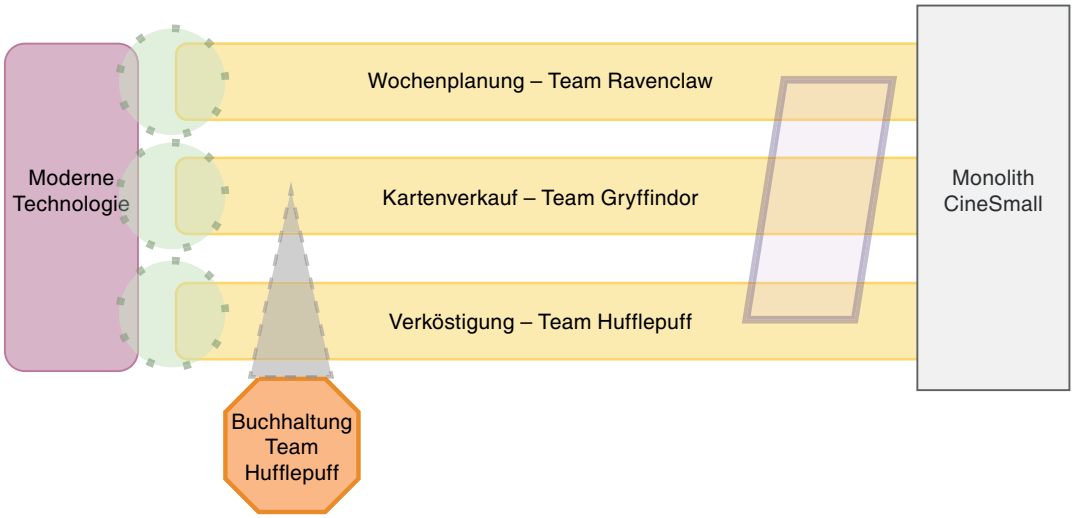

Abb. 8–8 *Topologien der Teams im Programmkino-Beispiel (ohne Zeitfaktor)*

Da Wochenplanung, Kartenverkauf und Verköstigung noch verwoben im Monolithen stecken, müssen unsere drei Teams, die diese Kontexte bearbeiten, miteinander kollaborieren. Die Zusammenarbeit wird eng sein und sie werden sich immer wieder in die Quere kommen.

Die Arbeit an dem Buchhaltungssystem, also seine Einbindung über eine Schnittstelle mit Anti-Corruption Layer in CineSmall, wird von einem Teil des Hufflepuff-Teams als Complicated Subsystem Team durchgeführt. Dabei entsteht eine X-as-a-Service-Zusammenarbeit mit dem Team Gryffindor, weil der Kartenverkauf Nachrichten an die Buchhaltung schicken will. Auch die Verköstigung wird eine X-as-a-Service-Zusammenarbeit benötigen, um Verkäufe zu verbuchen.

8.2.4 Teamentwicklung in der Zeit

In Abbildung 8–8 haben wir die Topologien zu einem bestimmten Zeitpunkt betrachtet. Aber man kann mit dem Team-Topologies-Ansatz auch die Zusammenarbeit und die Art der Teams in der Zeit visualisieren. In Abbildung 8–9 sind beispielhaft ein Platform Team, ein Complicated Subsystem Team, zwei Enabling Teams, drei Stream-aligned Teams und ihre Zusammenarbeit abgebildet. Die Zeit oder der »Flow of Change« läuft von links nach rechts.

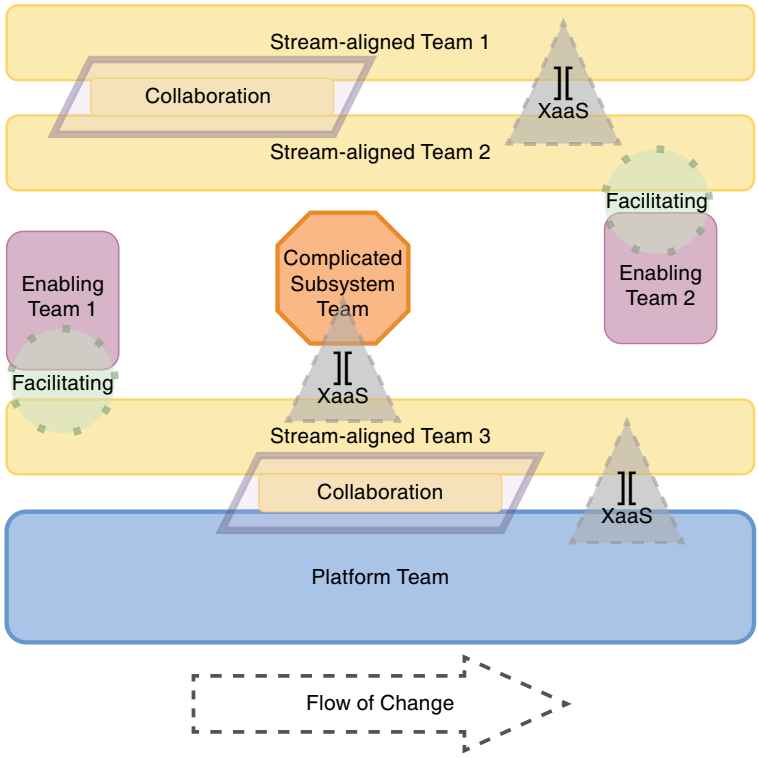

Abb. 8–9 *Topologien der Teams in der Zeit*

Das Enabling Team 1 hat am Anfang das Stream-aligned Team 3 unterstützt, während das Enabling Team 2 erst am Ende des hier dargestellten Zeitausschnitts dem Stream-aligned Team 2 geholfen hat. Stream-aligned Team 1 und Stream-aligned Team 2 haben einige Zeit eng zusammengearbeitet – sie haben kollaboriert. Das Stream-aligned Team 3 war einige Zeit eng mit dem Platform Team verbunden. In beiden Fällen gab es hinterher eine X-as-a-Service-Schnittstelle zwischen den beiden Teams. Möglicherweise haben sie während der Phase der Collaboration an diesen Schnittstellen gearbeitet. Schließlich sieht man in der Mitte noch ein Complicated Subsystem Team, das eine X-as-a-Service-Schnittstelle für das Stream-aligned Team 3 zur Verfügung gestellt hat.

Diese zeitliche Darstellung eignet sich für Organisationen, in denen der Monolith bereits zerlegt ist und es tatsächlich getrennt voneinander arbeitende Stream-aligned Teams gibt. Unser Kinobeispiel ist noch nicht so weit. Im nächsten Abschnitt werden wir sehen, wie sich die Organisationsentwicklung gemeinsam mit der Zerlegung des Monolithen in der Zeit sichtbar machen lässt.

8.2.5 Architecture Modernization Enabling Teams

Um den Umbau eines Monolithen in der Zeit betrachten zu können, haben Eduardo da Silva und Nick Tune den Ansatz *Architecture Modernization Enabling Teams* (s. [da Silva & Tune 2023]), kurz AMET, entwickelt. AMET zielen auf das gleiche Problem wie die Domain-Driven Transformation: Ein Unternehmen hat einen großen Monolithen, der durch seine Verwobenheit für die Teams, die an ihm arbeiten, nur schwer und langsam zu verändern ist. Wenn das Unternehmen schnellere Innovationen benötigt, dann wird die verworrene Architektur zu einem Engpass.

Die Autoren empfehlen in ihrem Artikel, dass man ein temporäres Team zur Verbesserung der Architektur einsetzen sollte, um die Modernisierung (oder Transformation, wie wir in diesem Buch sagen) voranzutreiben. Vier grobe Schritte werden vorgeschlagen, die wir hier mit dem Kinobeispiel in Anlehnung an die Grafiken aus [da Silva & Tune 2023] nachbilden:

1. **Business-Problem**
   Im Anfangszustand ist die Architektur ein Blocker für die Geschäftsstrategie (s. Abb. 8–10). Wir haben drei Teams, die kollaborieren müssen, weil sie am selben Stück Software arbeiten.

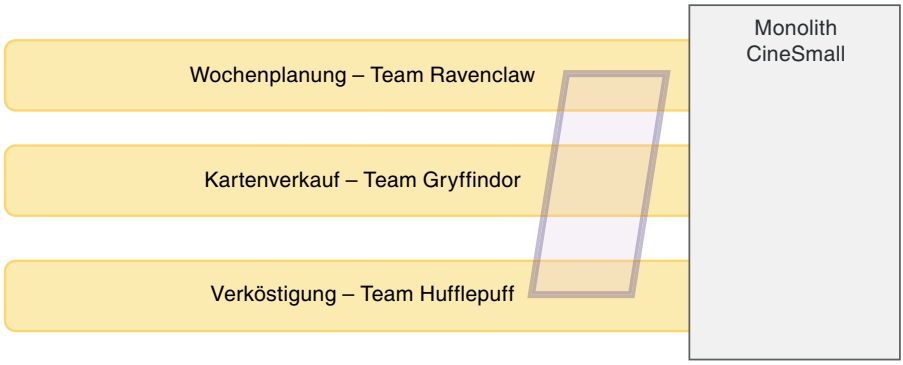

Abb. 8–10 *Anfangszustand mit Monolith*

2. **Anschub für die Modernisierung**
   Ein AMET wird gegründet, um die Initiative für die Architekturverbesserung zu starten. Dabei werden die passenden Menschen zusammengebracht und eine erste Idee zum Vorgehen etabliert (s. Abb. 8–11).

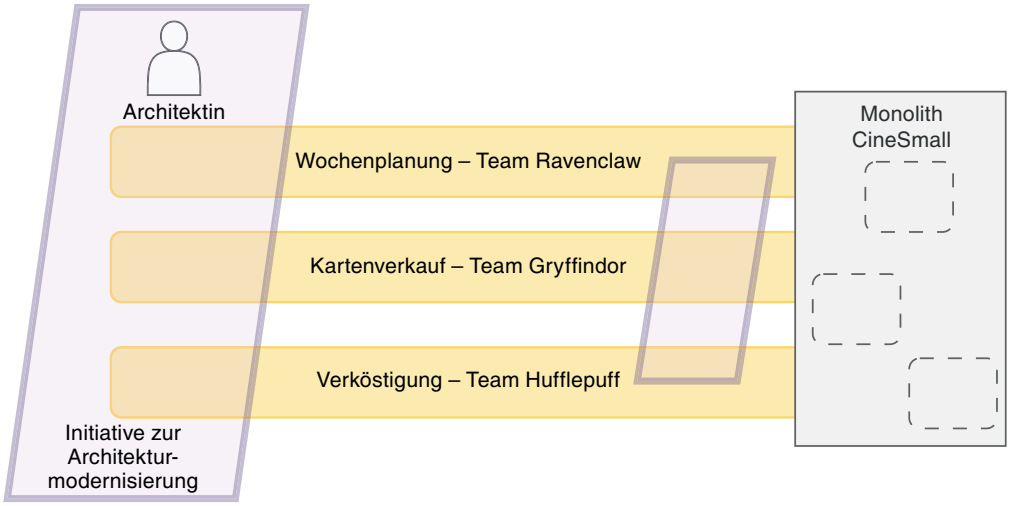

Abb. 8–11 *Initiative zur Architekturmodernisierung – Anschub*

3. **Durchführung der Modernisierung**
   Das AMET unterstützt die Teams dabei, die Dynamik der Architekturverbesserung aufrechtzuerhalten und eine bessere Architektur zu designen (s. Abb. 8–12).

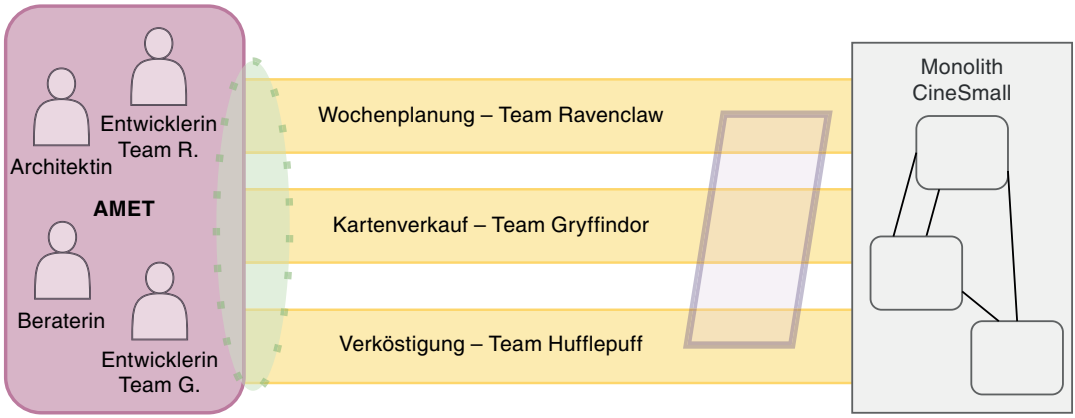

Abb. 8–12 *Initiative zur Architekturmodernisierung – Durchführung*

4. **Abschluss der Modernisierung**
   Die Architektur ist nicht länger ein Blocker für die Business-Strategie. Das Design wurde langfristig verbessert und die Reste des Monolithen werden von einem Platform Team gepflegt. Das AMET wird aufgelöst (s. Abb. 8–13).

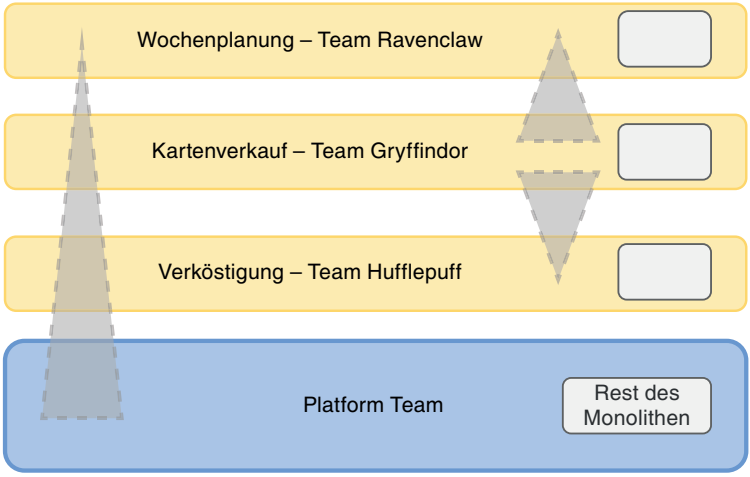

Abb. 8–13 *Initiative zur Architekturmodernisierung – Abschluss*

In dieser Abfolge von Schritten haben uns da Silva und Tune die perfekte Vorlage für die Domain-Driven Transformation geliefert. Die Domain-Driven Transformation, wie wir sie in Teil II des Buches vorgestellt haben und in Teil III weiterentwickeln werden, erläutert Ihnen, was Sie in Schritt 3, also der Durchführung der Modernisierung, im Detail tun müssen, damit die von da Silva und Tune beschriebene Modernisierung gelingt.

8.3 Zusammenfassung

In diesem Kapitel haben wir Ihnen Konzepte und Argumente an die Hand gegeben, wie Sie in Ihrer Organisation Diskussionen über die vorhandenen Teamstrukturen einleiten können. Dabei haben Sie den Zusammenhang zwischen Softwarearchitektur und Teamstruktur sowie die Team Topologies kennengelernt.

Im nächsten Teil dieses Buches wird es nun um die strategische Domain-Driven Transformation gehen. Die Frage, die sich immer wieder stellt, ist: Wie können wir unser Altsystem mit DDT so zerlegen, sodass es für unsere Entwicklungsteams wieder beherrschbar wird und gleichzeitig die Anwenderinnen besser unterstützt?

8.4 Wo stehen Sie?

Nun haben wir einiges über mögliche Teamorganisationen und ihre Verbesserung gelernt. Prüfen Sie Ihr Niveau im Bereich der Teamorganisation anhand der Aussagen in Tabelle 8–1! Wie stark treffen diese jeweils zu?

Nr.	Aussage	1	2	3	4	5
1	Unsere Teams sind cross-funktional zusammengesetzt.					
2	Unsere Organisation hat eine Vorstellung von kognitiver Last und Autonomie.					
3	Unsere Teams bestehen aus fünf bis neun Personen und sind auf langfristige Zusammenarbeit ausgelegt.					
4	Wir haben Stream-aligned Teams, die eine abgrenzbare Fachlichkeit, wie einen Bounded Context, umsetzen.					
5	Es gibt Teams, die eher in Richtung von Enabling, Complicated Subsystem oder Platform Teams gehen.					
6	Unsere Teams wissen, in welcher Kommunikationsbeziehung sie zueinander stehen.					
7	Unsere Enabling und Complicated Subsystem Teams haben schon die Erfahrung gemacht, dass ihre Unterstützung für die anderen Teams gar nicht dauerhaft erfolgen muss.					
8	Unsere Platform Teams haben ein Verständnis davon, dass ihre Schnittstellen für die auf sie aufbauenden Stream-aligned Teams möglichst einfach sein sollten.					
9	Unsere Teams sind nach UI/Client, Businesslogik/Backend und Datenbank organisiert.					
10	Unsere Teams müssen sich sehr viel absprechen, um Erweiterungen umzusetzen.					
11	Es gibt bei uns Teams, die aus mehr als 10 Mitgliedern bestehen.					
12	Die Teamzusammensetzung wird ständig verändert.					

Tab. 8–1 *Bewertungsbogen zum Thema Teamorganisation*

Zur Auswertung Ihrer Ergebnisse vgl. die Tabellen II–1 und II–2.