АРХИТЕКТУРНИ СТИЛОВЕ

Част 2

Съдържание на лекцията

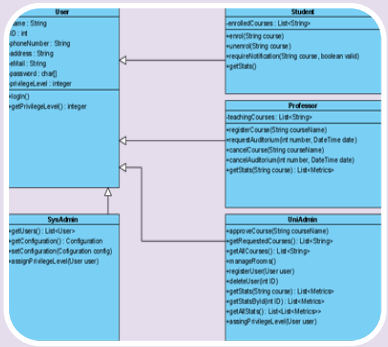
- Някои размисли върху софтуерната архитектура
- Определение на архитектурни стилове
- Различни стилове:
  - Implicit invocation/Message passing
  - Model-View-Controller
  - Обвивка (Wrapper)
  - Фасада
  - Архитектурни стилове в облака

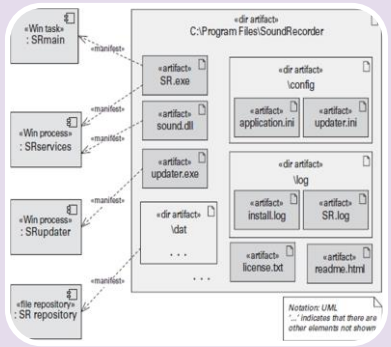
Софтуерна архитектура

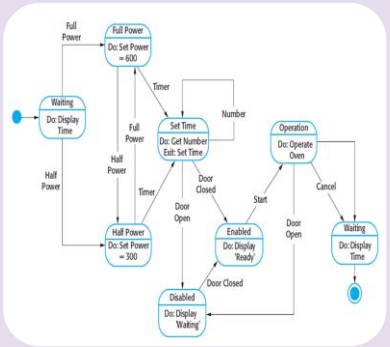
- Описанието на софтуерната архитектура надхвърля алгоритмите и структурите от данни, които съставляват приложенията.
- Софтуерната архитектура се отнася до:
  - Общата организация и глобалната структура на управление в системата;
  - Протоколите за комуникация, синхронизация и достъп до данни;
  - Разпределението на функционалноссттта по елементите на дизайна;
  - Композирането на тези елементи;
  - Физическо разпределение на тези елементи;
  - Скалируемостта и производителността;
  - Избор между различните алтернативи на дизайна.

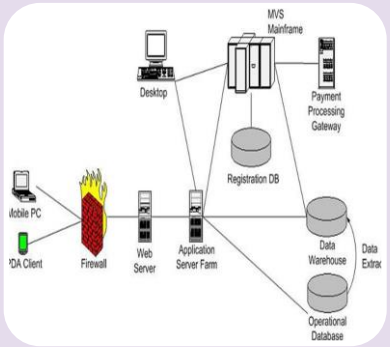
Източник: David Garlan and Mary Shaw. An Introduction to Software Architecture, Carnegie Mellon University

4+1 изгледа на софтуерната архитектура *(по Philippe Kruchten) 1/2*









Logical view

Code (Deve-lopment) view

Process view

Deploy ment (Physical) view

SCENARIOS

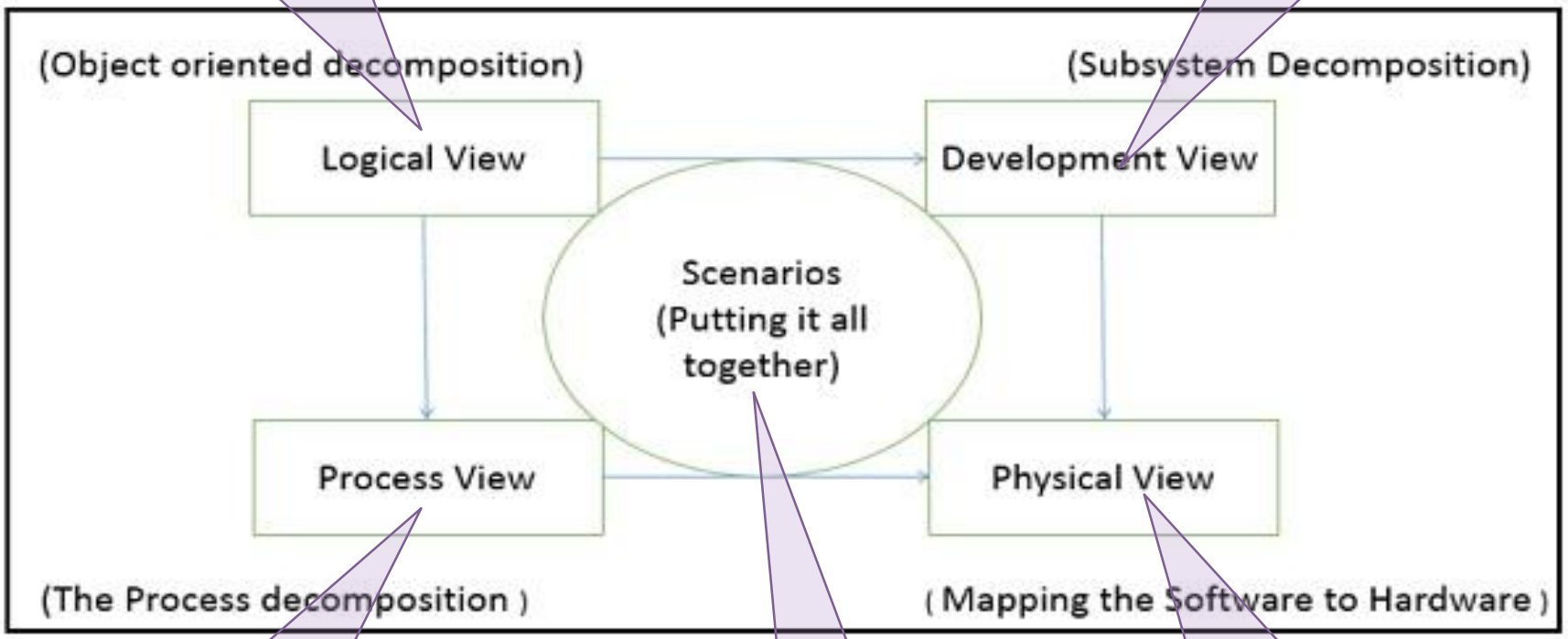
Sources: http://www.agilemodeling.com

Software Engineering by Ian Sommerville, 9th edition (2010), Addison-Wesley Pub Co;

4+1 изгледа на софтуерната архитектура *(по Philippe Kruchten) 2/2*

т. нар. концептуален изглед - описва обектния модел на дизайна

описва статичната организация или структура на кода в средата за разработка



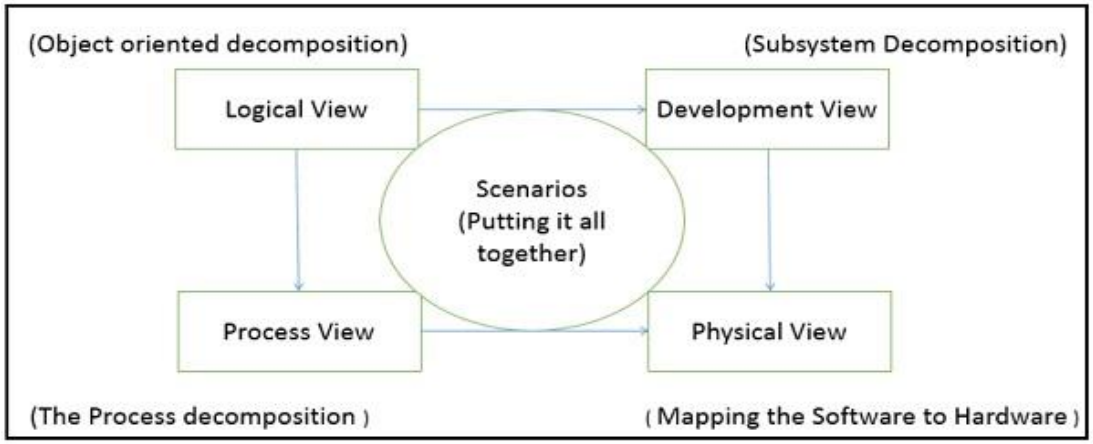
описва аспектите на конкурентност и синхронизация

UML случаи на употреба

описва разполагането (deployment) на софтуера върху хардуера

Логически изглед на СА

- Логическият изглед на софтуера има четири нива на абстракция:
  - ***Компоненти и конектори***
  - Техните ***интерфейси***
  - Архитектурни ***конфигурации*** - специфична топология на взаимосвързани компоненти и конектори
  - Архитектурни ***стилове*** – образци (patterns) за успешни и практически доказани архитектурни конфигурации



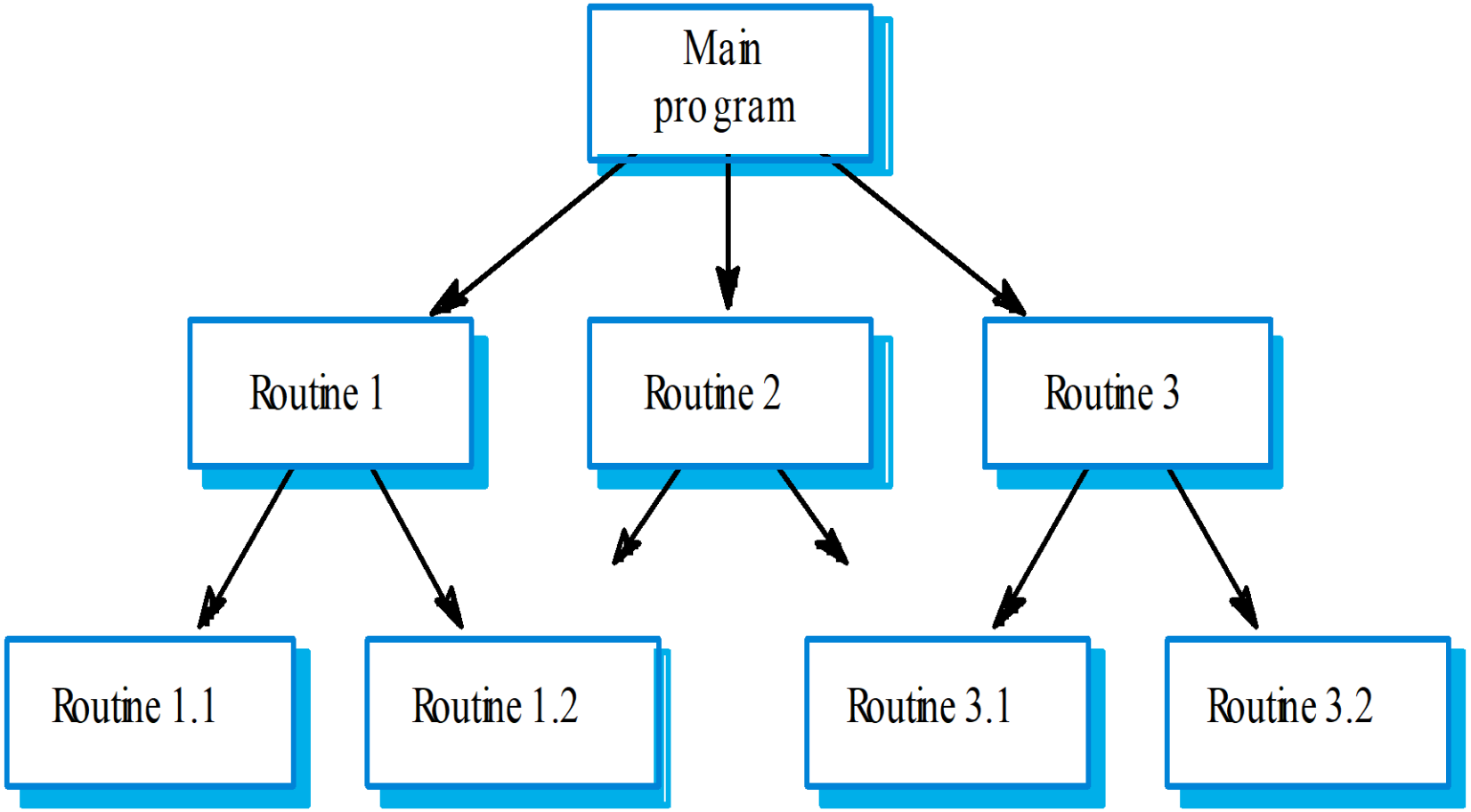
Архитектурен стил

- Дефиниция: *Архитектурният стил определя семейство от системи по отношение на модел (образец) на структурна организация*.
- Архитектурните стилове определят:
  - Речникът (vocabulary) от компоненти и конектори, които могат да се използват в екземпляри от този стил
  - Набор от ограничения (constraints) за това как те могат да бъдат комбинирани, като:
    - Топология на описанията (например, без цикли)
    - Семантика на изпълнение (например, процеси които се изпълняват конкурентно/паралелно)

Стил неявно извикване (Implicit invocation)

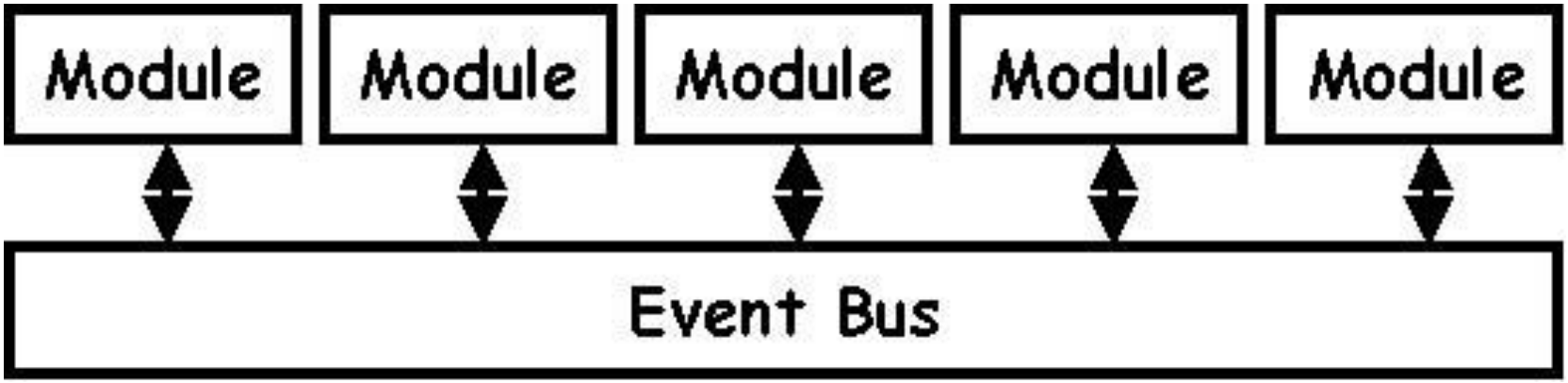
- Компонентите в системата си взаимодействат чрез изпращане на събития
- Събитията могат да съдържат не само управляващи съобщения (control messages), но също така и данни
- Други имена на този стил:
  - Publish-subscribe
  - Event-based стил
  - Message passing стил

Explicit invocation стил

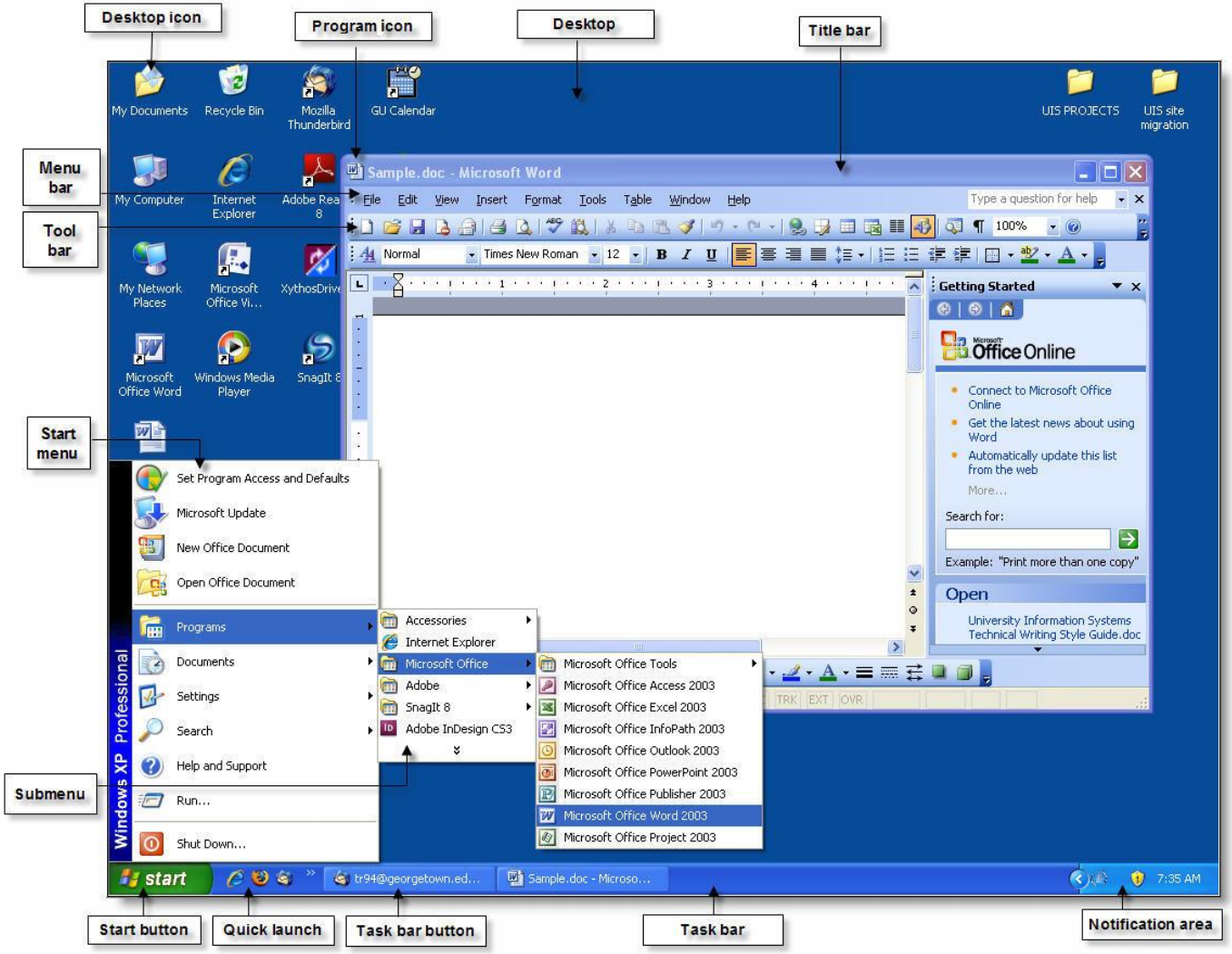


Неявно извикване (*Implicit* invocation)

- Компонентите на този стил работят едновременно и комуникират чрез получаване или излъчване на събития
- Компонентите са свързани през конектор, който представлява шина за събития (event bus)
- Всички компоненти си взаимодействат чрез шината



Пример за неявно извикване

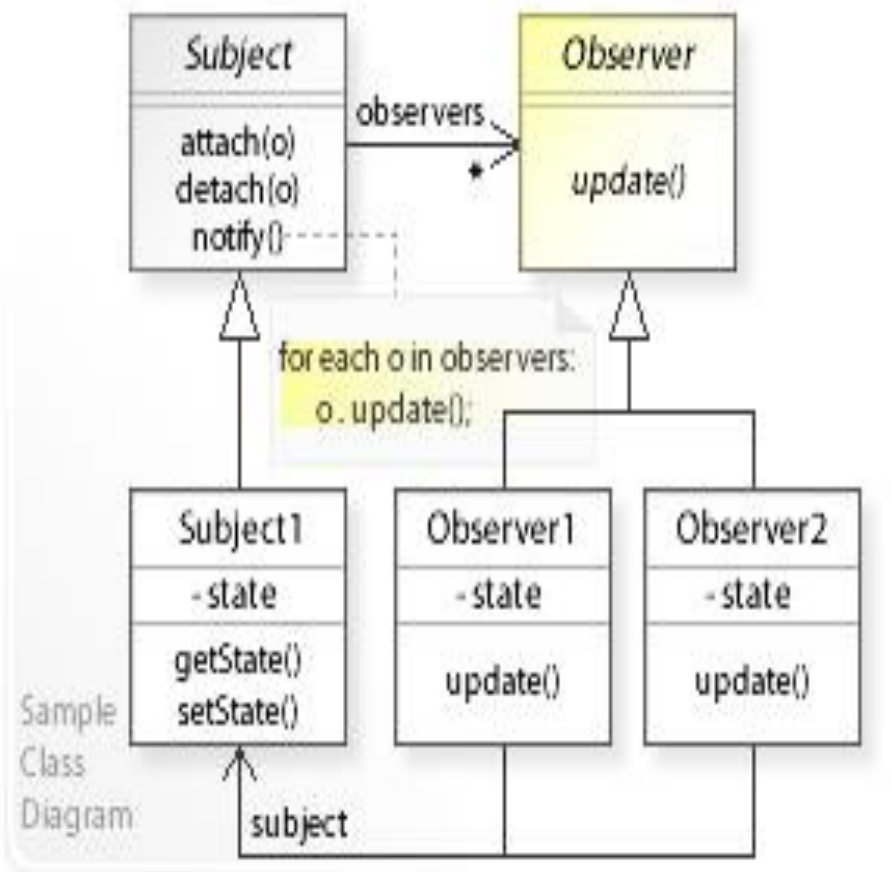


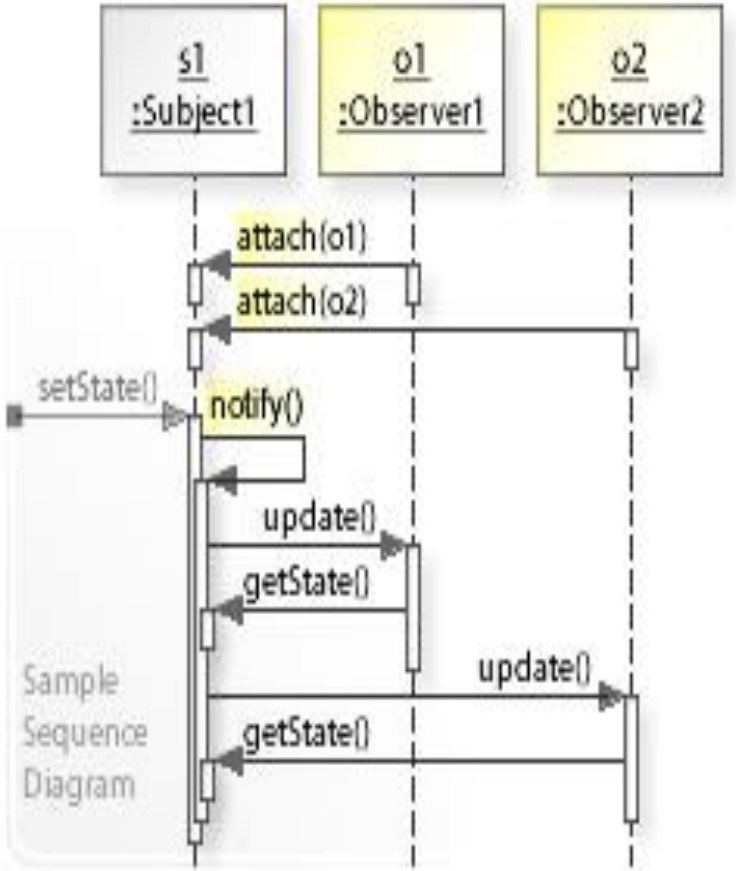
- Взаимодейст-вията на потребителите се предават към приложението като събития
- За всеки event – event handler

Системи с имплицитно (неявно) извикване

- Архитектурен стил с висока кохезия и слаба свързаност (loose coupling)
- Примери:
  - всички съвременни операционни системи (OS)
  - интегрирани среди за разработка (IDE)
  - системи за управление на бази данни (DBMS)
- Гарланд и Шоу: „Идеята зад неявното извикване е, че:
  - Вместо да извиква директно процедура, даден компонент може да анонсира (или излъчи) едно или повече събития.
  - Други компоненти в системата могат да регистрират интерес към събитие чрез асоцииране на своя процедура със събитието.
  - Когато събитието е обявено, системата сама извиква всички процедури, които са били регистрирани за събитието.
  - По този начин събитие „имплицитно“ причинява извикване на процедури в други модули.“

Свързан шаблон за проектиране - *Observer*





Приложимост

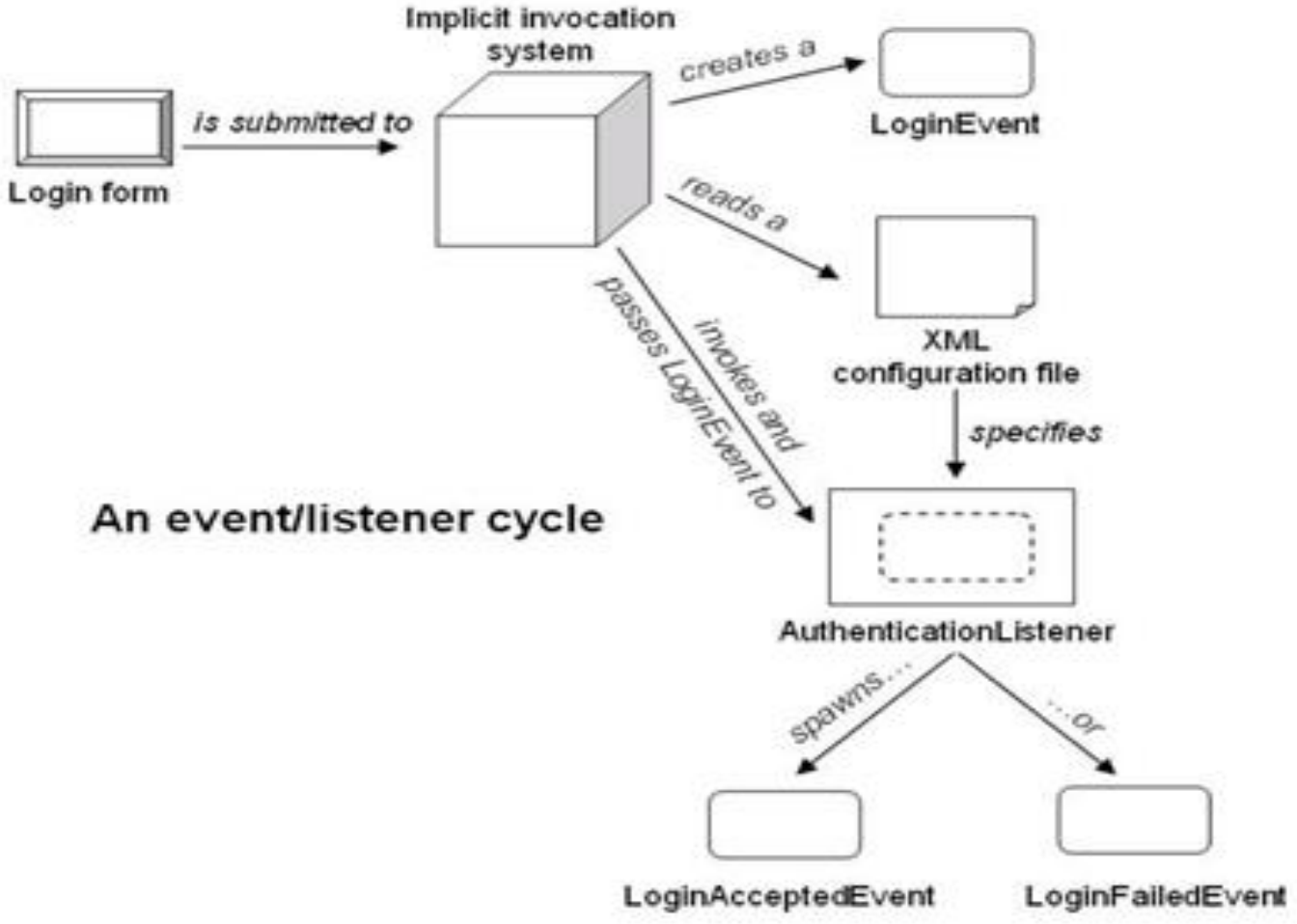
Използвайте шаблона Наблюдател (*Observer*) във всяка от следните ситуации:

- Когато една абстракция има два аспекта, единият зависи от другия. Капсулирането на тези аспекти в отделни обекти ви позволява да ги променяте и използвате повторно.
- Когато промяна на един обект изисква промяна на други, а вие не знаете колко обекти трябва да бъдат променени.
- Когато един обект трябва да може да уведомява други обекти, без да прави предположения кои са тези обекти. С други думи, когато не искаме тези обекти да бъдат силно свързани.

Пример

Benjamin Edwards. An Introduction to Implicit Invocation Architectures, https://pdfs.semanticscholar.org/4907/3d810c7f98244e9bbea79d852eff8840d034.pdf

AuthenticationListener's tryLogin() method, passing to it the event. Based on information in the event, the tryLogin() method will seek to authenticate the user. If the authentication succeeds, a new LoginAcceptedEvent is triggered. If authentication fails, a new LoginFailedEvent is triggered. The cycle then continues, with any listeners of the new event being notified.



Предимства на implicit invocation стила

- Слаба свързаност (Louse coupling)
  - Компонентите могат да бъдат много разнородни
  - Компонентите се подменят или използват много лесно
- Висока кохезия – компонентите са по-прости, по-независими и оттук – преизползваеми
- Голяма ефективност за разпределените системи - събитията са независими и могат да се предават по мрежата
- Сигурност - събитията лесно се проследяват и регистрират

> Maximized cohesion (simple components) and minimized coupling (fewer connectors) are hallmarks of a flexible, maintainable architecture.
>
> *Benjamin Edwards*

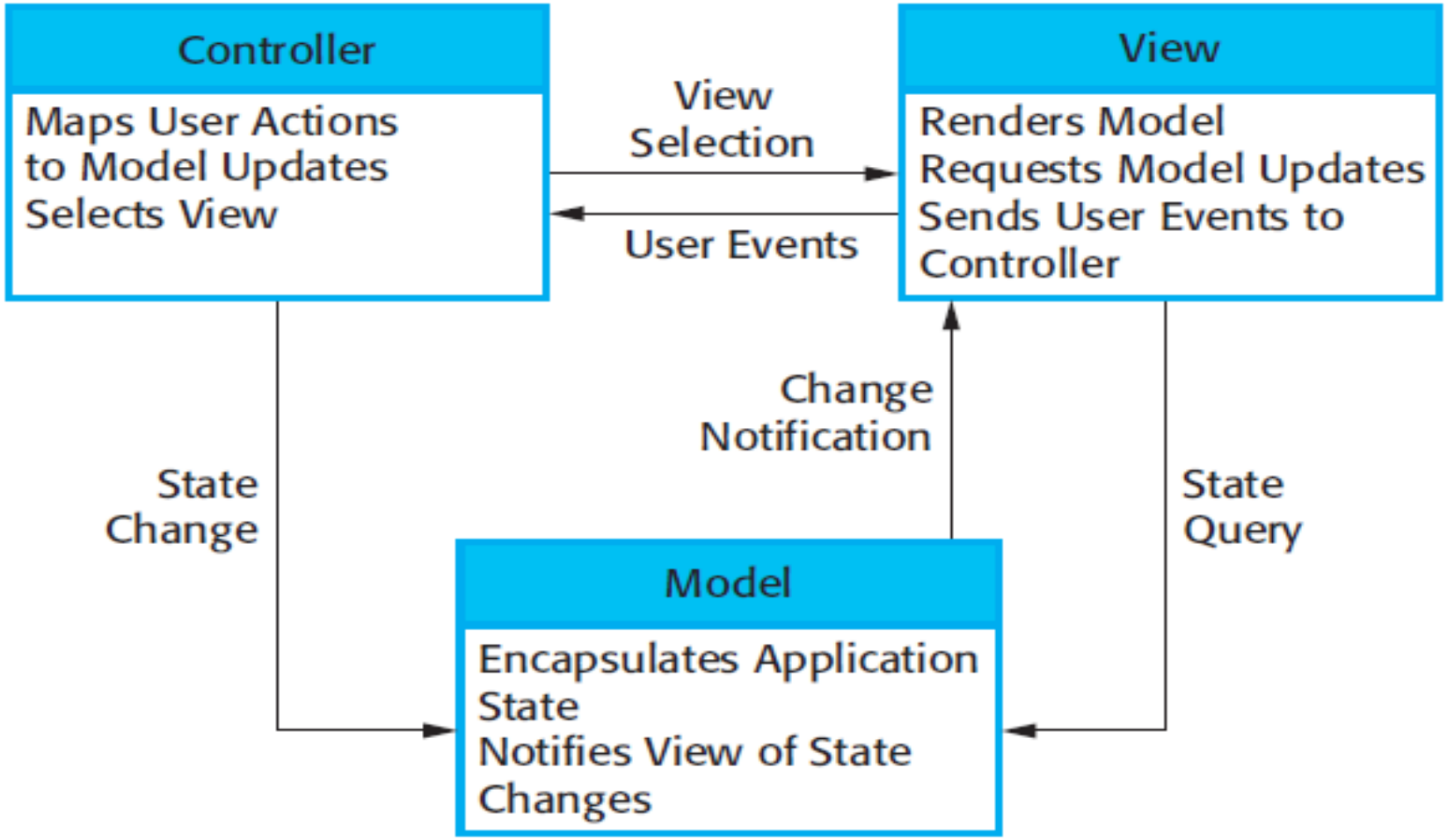
Недостатъци на implicit invocation стила

- Неясна структура на системата
  - Последователността на изпълнението на компоненти е трудно да се контролира
  - Трудно отстраняване на грешки
- Не е сигурно дали има компонент, който да реагира на дадено събитие
- Големите количества данни са трудни за пренасяне от събития
- Проблем с надеждността - неизправността на шината за събития ще компрометира цялата система

Стил Модел-Изглед-Контролер (Model-View-Controller, MVC)

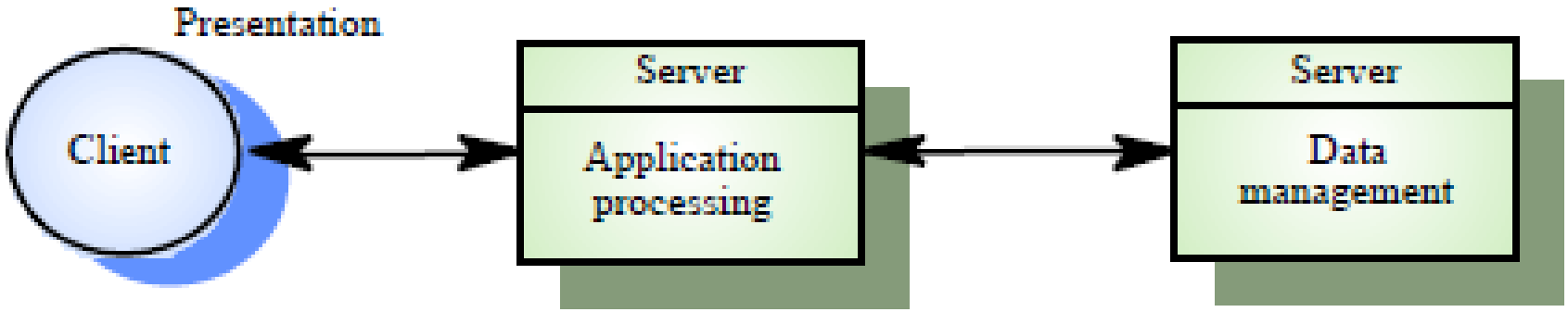
- Позволява независимост между данни, представяне на данни и потребителския контрол
- Моделният компонент (Model) представлява знанието. Той управлява поведението и данните на домейна на приложението, изпраща информация за неговото състояние (към изгледа) и отговаря на инструкции за промяна на състоянието (обикновено от контролера)
- Изгледът (View) има задължението да управлява представянето на информация пред потребителите
- Контролерът (Controller) управлява взаимодействието с потребителя (например кликвания на мишката, натискане на клавиш и т.н.) и информира модела или изгледа, за да предприеме подходящи действия

MVC стил



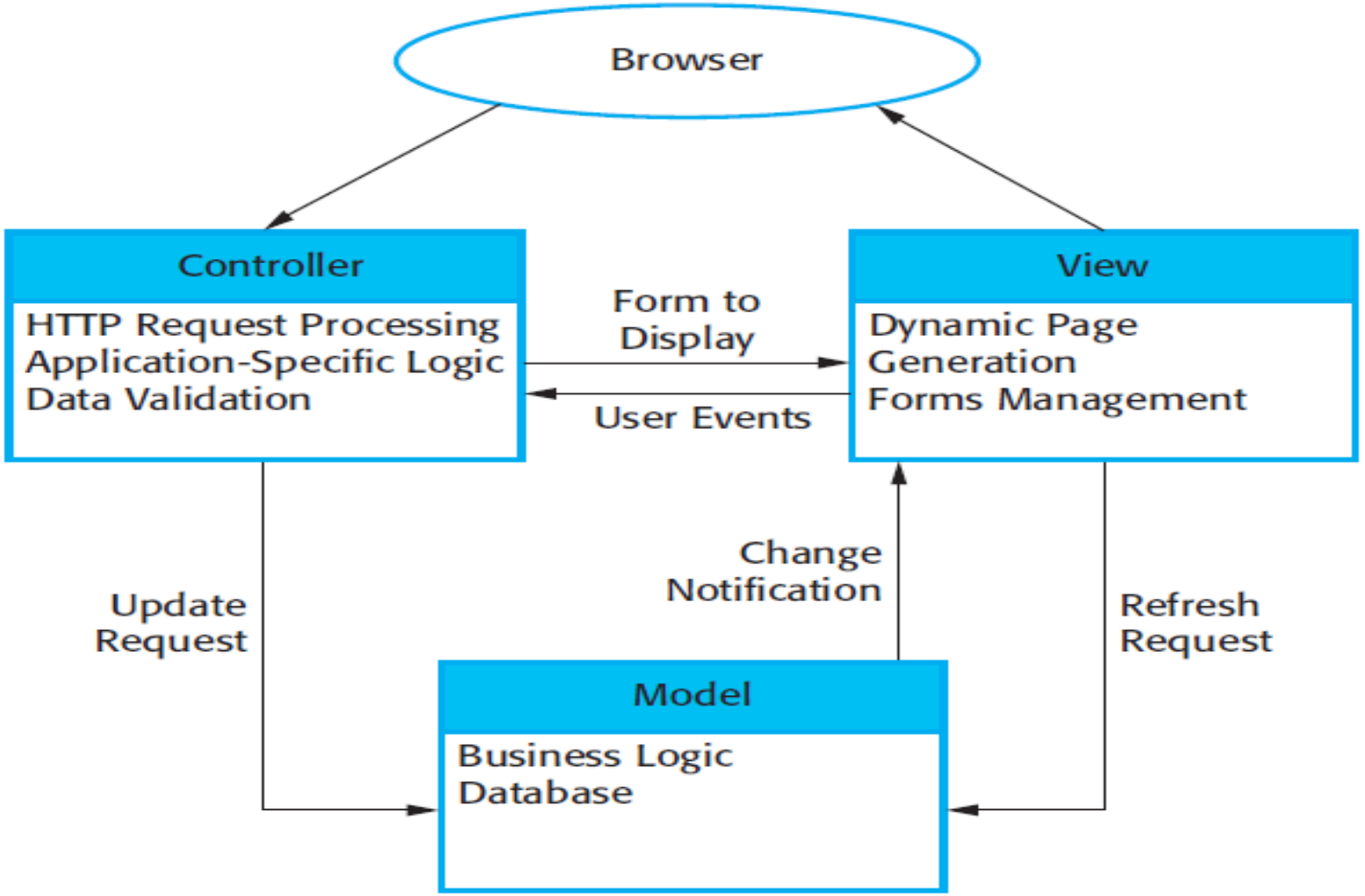
Software Engineering by Ian Sommerville, 9th edition (2010), Addison-Wesley Pub Co;

Трислоен модел клиент-сървър



- По-добра производителност
- По-добра сигурност
- Какво още бихте добавили?
- …

MVC – пример в Уеб приложения



Software Engineering by Ian Sommerville, 9th edition (2010), Addison-Wesley Pub Co;

Предимства на MVC

- Голяма гъвкавост
  - Лесен за поддръжка и прилагане на бъдещи подобрения
  - Ясно разделяне между логиката на представяне и бизнес логиката
- По-лесна актуализация – напр. при поддръжка на нови типове потребители
- Изгледът е отделен и в повечето системи претърпява много промени
- Разработването на приложението става бързо – много разработчици могат лесно да си сътрудничат и да работят заедно
- По-лесно за отстраняване на грешки – в приложението имаме няколко нива

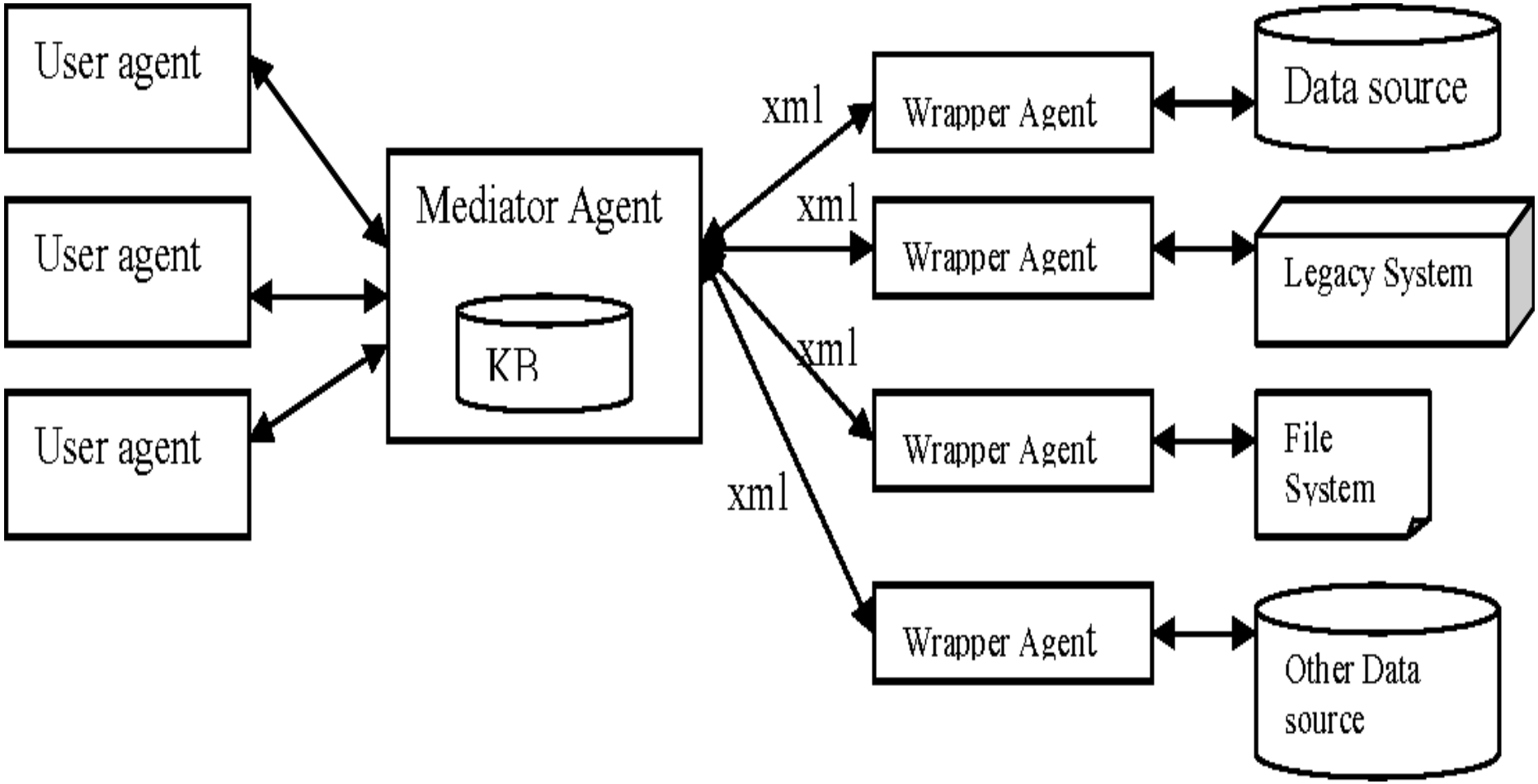
Недостатъци на MVC

- Дори ако моделът на данни е прост, този стил може да внесе сложност и да изисква много допълнителен код
- Не е подходящ за малки приложения – архитектурата е относително сложна
- Проблем с производителността при чести актуализации в модела
- Трябва да има строги правила относно методите

Стил Обвивка (Wrapper)

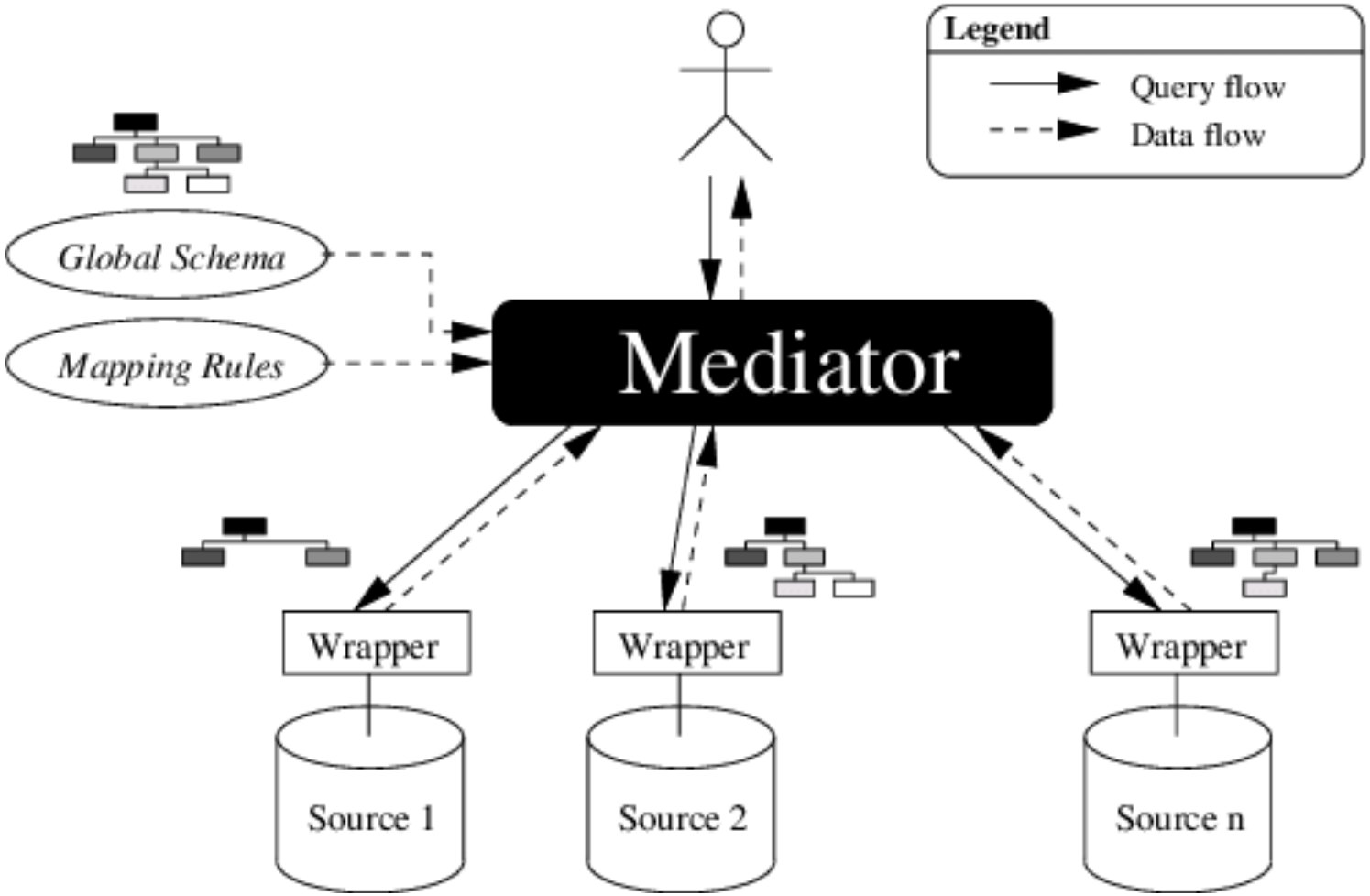
- Въвежда допълнителен компонент, който действа като посредник между два взаимодействащи компонента
- Да приемем, че имаме два компонента: клиент (A) и сървър (B). Очевидно A зависи от B. Тази зависимост трябва да бъде сведена до минимум
- Съществуват различни техники по отношение на характера на зависимостта. Те могат да имат различни имена в литературата, като ние ги наричаме стил на *обвивка* (*wrapper* style)

Wrapper архитектура 1/2



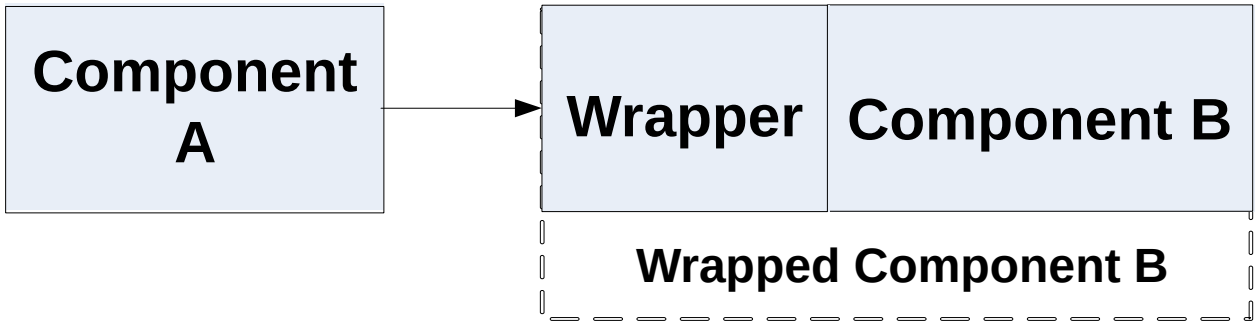
https://www.semanticscholar.org/paper/Data-Integration-using-Agent-based-Mediator-Wrapper-Shi/63867b0dbfde12a0ec4f803d5ce569e2fbacea59/figure/5

Wrapper архитектура 2/2



https://www.researchgate.net/figure/Typical-Mediation-Architecture_fig1_220737859

Wrapper

- В зависимост от неговата семантика може да служи за различни цели
  - Адаптер
  - Фасада
  - Name server
  - Посредник
- 

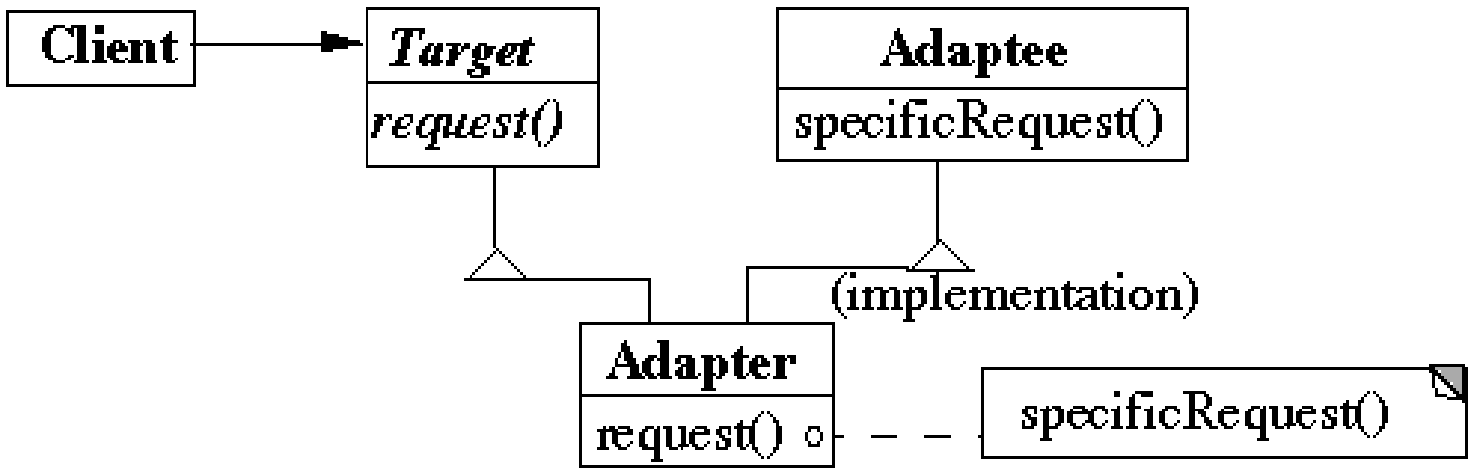
Адаптер (Adapter)

- **Цел -** преобразува интерфейса на даден клас в друг интерфейс, какъвто клиентът очаква. Адаптерът позволява на двата класа да работят заедно, което иначе не е възможно, защото са с несъвместими интерфейси.

- **Познат още като -** Wrapper

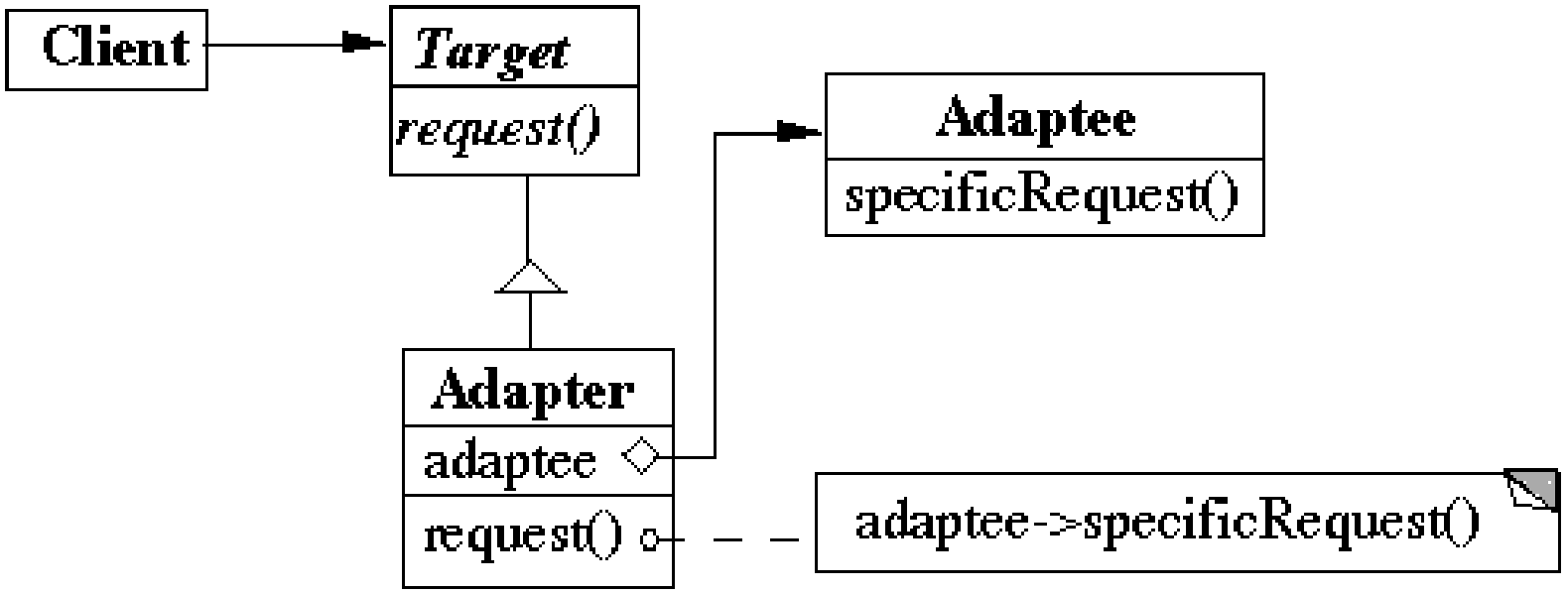
Участници и структура (Class Adapter)

- **Target** (Shape) - дефинира домейн-специфичен интерфейс за ползване от клиента (Client).
- **Client** (DrawingEditor) – кооперира с обекти, ориентирани към интерфейса Target.
- **Adaptee** (TextView) – дефинира съществуващ интерфейс, изискващ адаптиране.
- **Adapter** (TextShape) – адаптира интерфейса Adaptee към този на Target.



Участници и структура (Object Adapter + Delegation)

- **Участници –** същите както на предходния слайд.
- **Комуникация –** клиентите извикват операция на екземпляр на Adapter. На свой ред, адаптерът извиква операции на Adaptee, които изпълняват заявката.



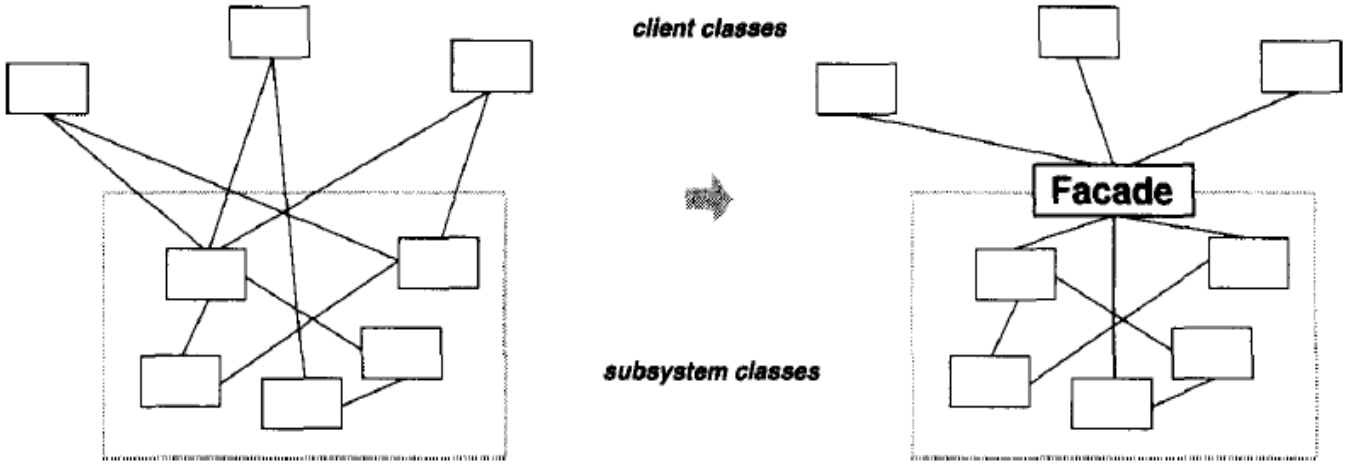
Приложимост

Използваме шаблона Adapter, когато:

- искаме да използваме съществуващ клас и интерфейсът му не съвпада с този, който ни е нужен.
- искаме да създадем клас за многократна употреба, който си сътрудничи с несвързани или непредвидени класове, т.е. с класове, които не е задължително да имат съвместими интерфейси.
- (*само за обектен адаптер*) трябва да използваме няколко съществуващи подкласа, но е непрактично да адаптираме техните интерфейси чрез наследяване от всеки един. Адаптер-обектът може да адаптира интерфейса на родителския клас.

Шаблон Фасада (Façade,Facade)

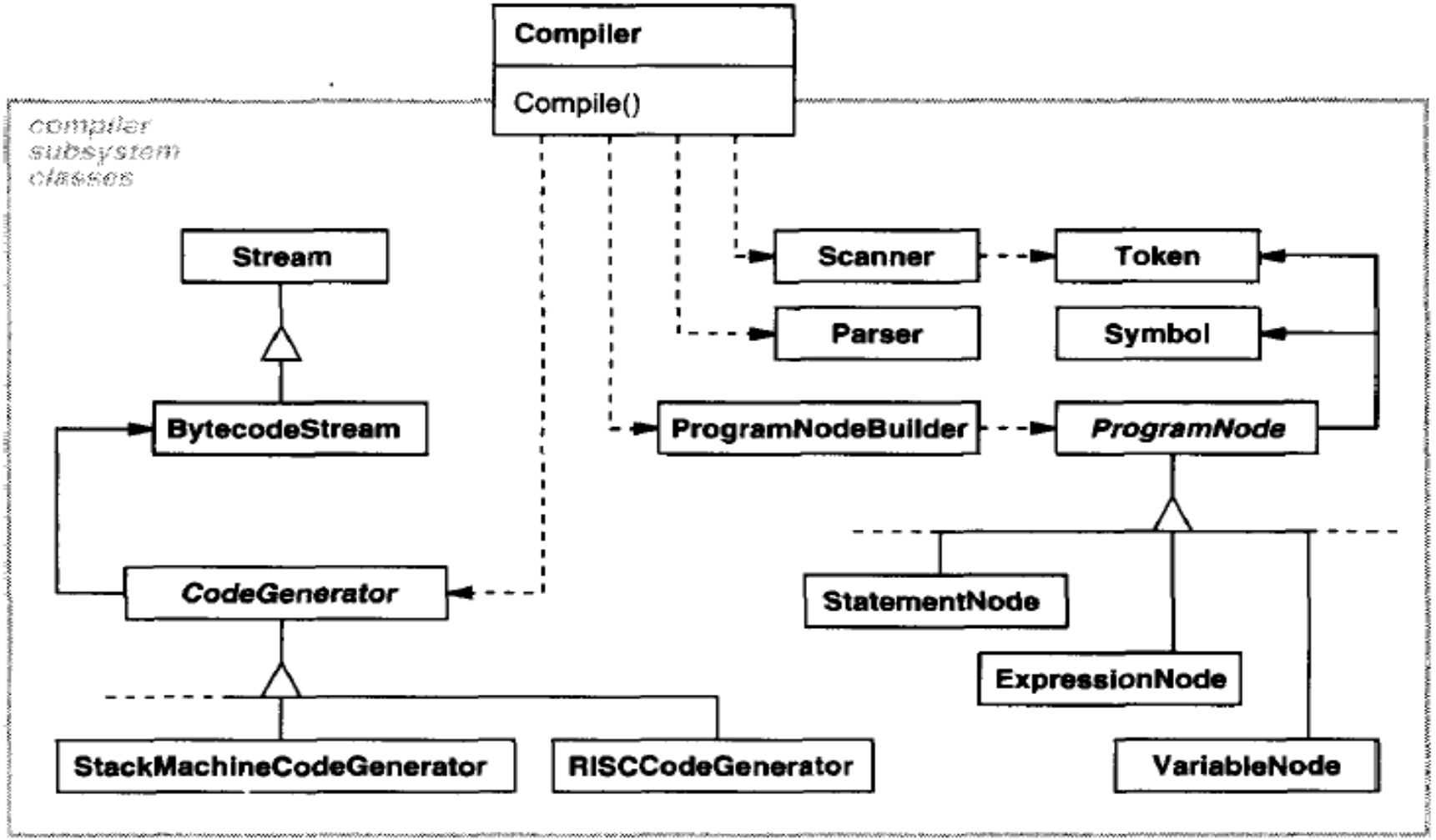
- **Цел -** осигурява единен интерфейс към набор от интерфейси в подсистема - определя интерфейс от по-високо ниво с цел по-лесно използване на подсистемата.
- **Мотивация –** цел на дизайна е да се сведе до минимум комуникацията и зависимости между подсистеми. Един от начините за постигане на това е да се въведе обект-фасада, с единен, опростен интерфейс на по-общите свойства на подсистемата.



Мотивация – Compiler Interface

- Разглеждаме приложения с достъп до подсистема-компилатор. Тя съдържа класове като Scanner, Parser, ProgramNode, BytecodeStream, и ProgramNodeBuilder, които имплементират компилатора.
- Някои специализирани приложения трябва да достъпват тези класове директно. Но за клиентите на компилатора детайли като разбор на входния текст и генериране на код не са важни - те просто искат да компилират код.
- За да предостави интерфейс от по-високо ниво, който предпазва клиентите от тези класове, компилаторната подсистема включва Compiler клас, имащ единен интерфейс към функционалността на подсистемата и по този начин действащ като фасада: тя предлага на клиентите един опростен интерфейс към компилатора като обединява заедно класове от компилатор, без да ги скрива напълно.

Решението


Compiler
Compile()
compiler subsystem classes
Stream
Scanner
Token
Parser
Symbol
BytecodeStream
ProgramNodeBuilder
ProgramNode
CodeGenerator
StatementNode
ExpressionNode
StackMachineCodeGenerator
RISCCodeGenerator
VariableNode

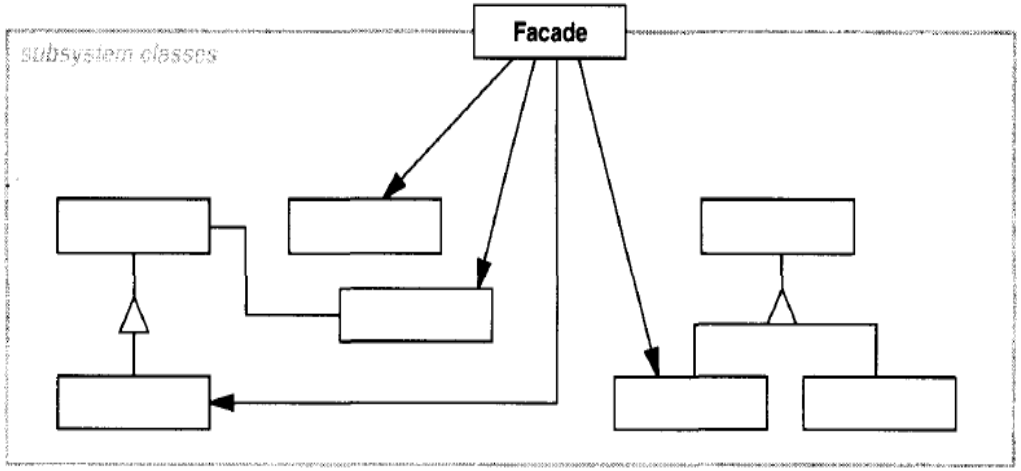
Приложимост

Използваме шаблона Фасада (Façade), ако:

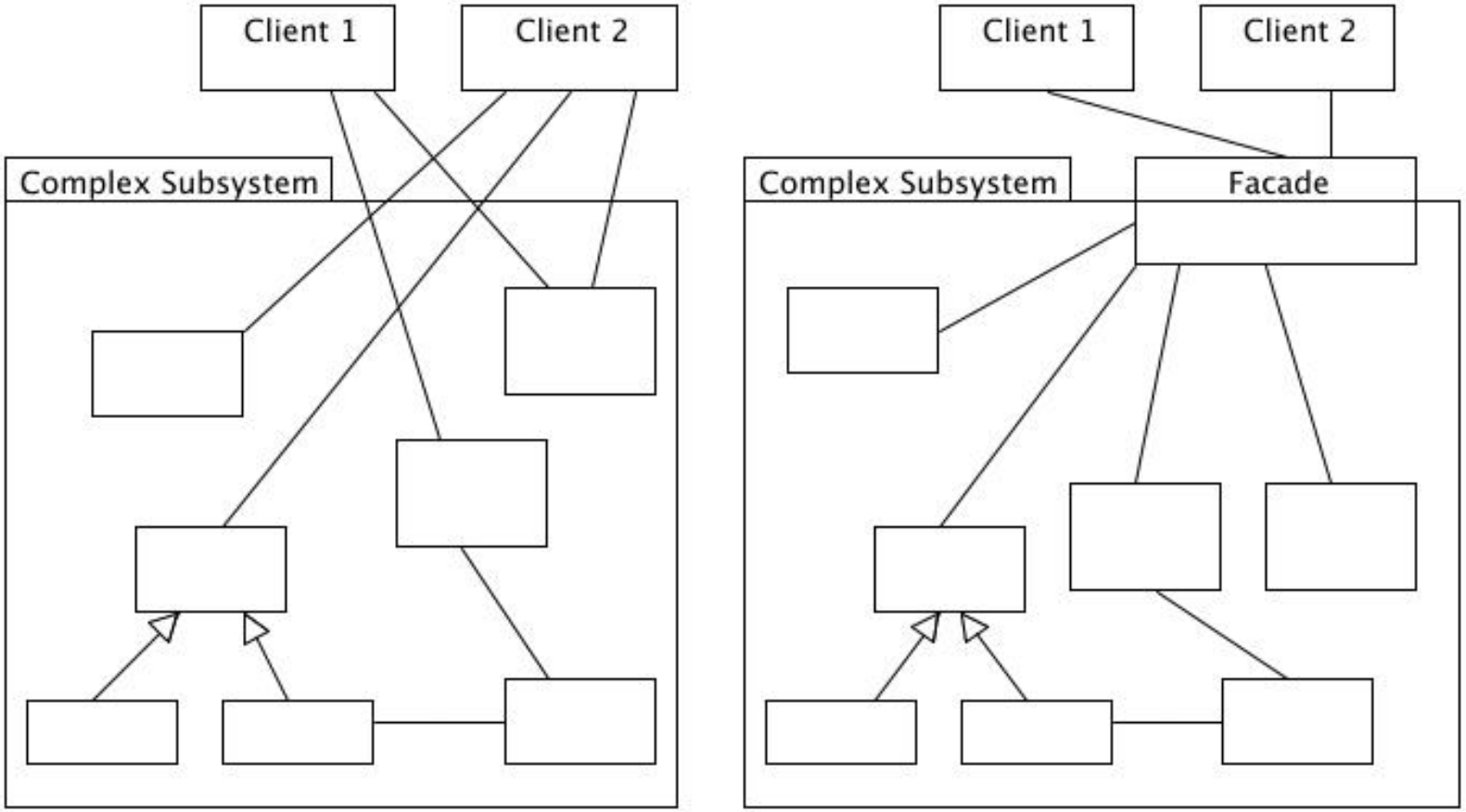
- имаме нужда от прост интерфейс до комплексна под-система. Повечето шаблони при прилагането им водят до *повече и по-малки класове*. Това прави подсистемата лесна за *многократна употреба и за персонализация*, трудна за използване от клиенти, които не трябва да я персонализират. Една фасада може да осигури *прост изглед по подразбиране на подсистемата*, и то достатъчно добър за повечето клиенти. Само клиенти, които се нуждаят повече адаптивност, ще трябва да погледнат отвъд фасадата.
- има много зависимости между клиента и имплементационните класове на абстракцията. Фасадата отделя подсистемата от клиента и други подсистеми, и оттук - независимост и преносимост.
- искаме една входна точка за всяко подсистемно ниво; ако подсистемите са зависими, то те си комуникират само чрез техните фасади

Участници и структура

- Façade (Compiler)
  - - знае кои подсистемни класове ще изпълнят заявката;
  - - делегира клиентските заявки към тези класове.
- Подсистемни класове (Scanner, Parser, ProgramNode, etc.)
  - - имплементират системната функционалност;
  - - извършват работата, заявена през обекта Façade;
  - - нямат знания за фасадата, т.е. нямат референция към нея.



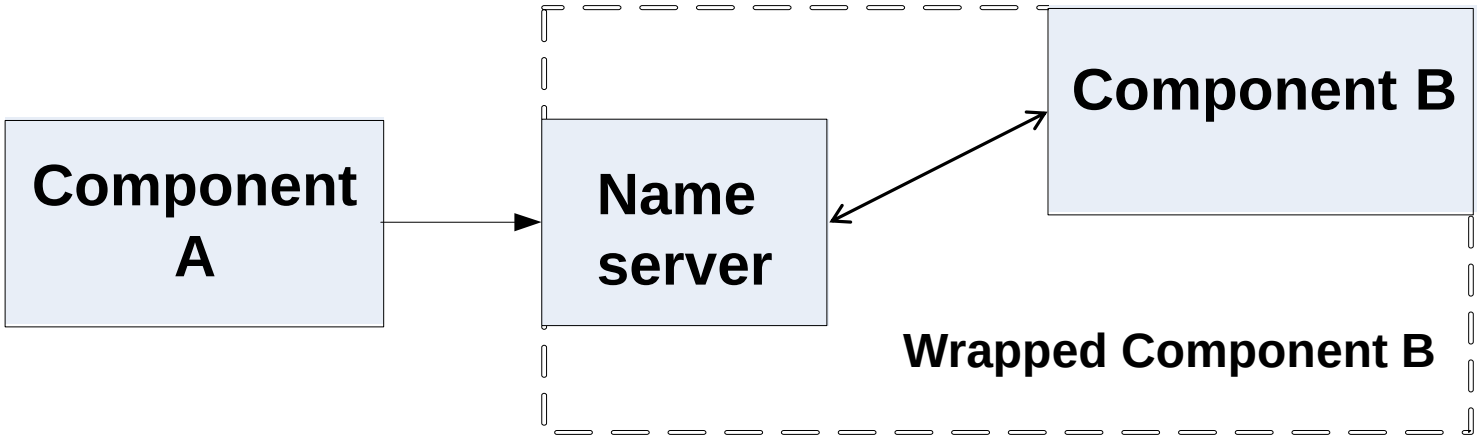
Фасада (Facade)



Source: http://best-practice-software-engineering.ifs.tuwien.ac.at/patterns/facade.html

Name server

- Когато клиентът (A) не знае за местоположението на сървъра (B), обвивката се нарича “name server”



Посредник

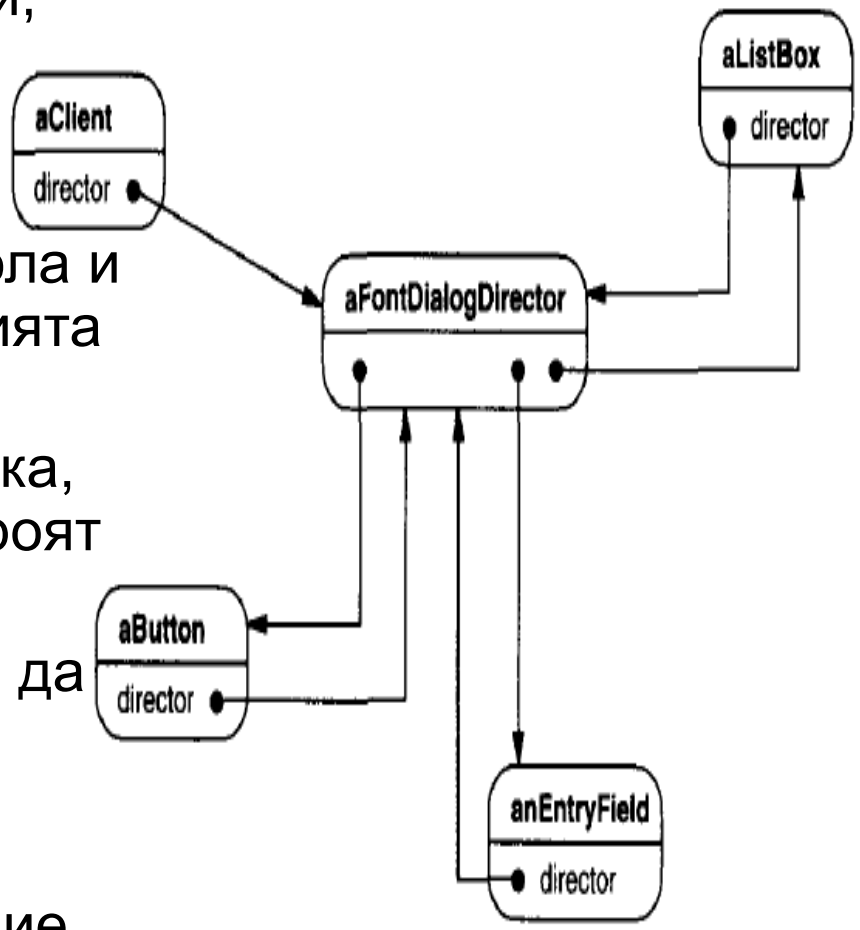
- Идея - дефинира обект, който капсулира как взаимодействат набор от обекти. Посредникът дава възможност за свободно свързване, като не позволява на обектите да се позовават изрично един на друг и позволява лесно да променяме тяхното взаимодействие.
- Мотивация - (1) разделянето на система на много обекти обикновено подобрява повторната употреба, но много взаимовръзки правят по-малко вероятно обектът да работи без останалите - системата действа така, сякаш е монолитна. (2) трудно се променя поведението на системата по някакъв съществен начин, тъй като поведението се разпределя между много обекти.

Пример за медиатор 1/2

- Диалогов прозорец представя колекция от бутони, менюта и полета за въвеждане, често със зависимости между тях (бутонът се деактивира, когато определено входно поле е празно; избирането на запис в списък променя съдържанието на поле за запис и т.н.).
- Така класовете трябва да бъдат персонализирани, за да отразяват специфичните за диалога зависимости.
- Индивидуалното им персонализиране чрез подкласове ще бъде тежко, тъй като участват много класове.

Пример за медиатор 2/2

- Можем да избегнем тези проблеми, като капсулираме колективното поведение в отделен обект - ***медиатор***.
- Медиаторът е отговорен за контрола и координирането на взаимодействията на група обекти.
- Обектите познават само посредника, като по този начин се намалява броят на взаимовръзките.
- Например FontDialogDirector може да бъде посредник между обектите в диалоговия прозорец. Обектът FontDialogDirector ги познава и координира тяхното взаимодействие като комуникационен хъб.



АРХИТЕКТУРНИ СТИЛОВЕ В ОБЛАК (CLOUD)

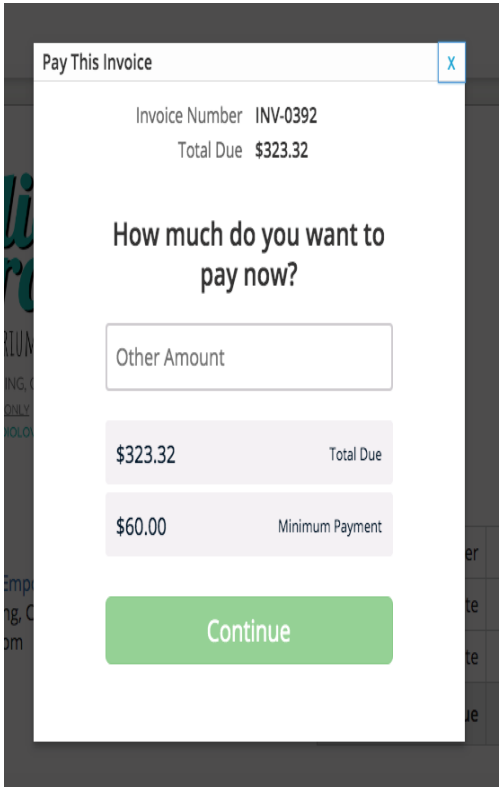
http://msdn.microsoft.com/en-us/library/dn568099.aspx

Какво е облачно изчисление (Cloud Computing)? 1/2

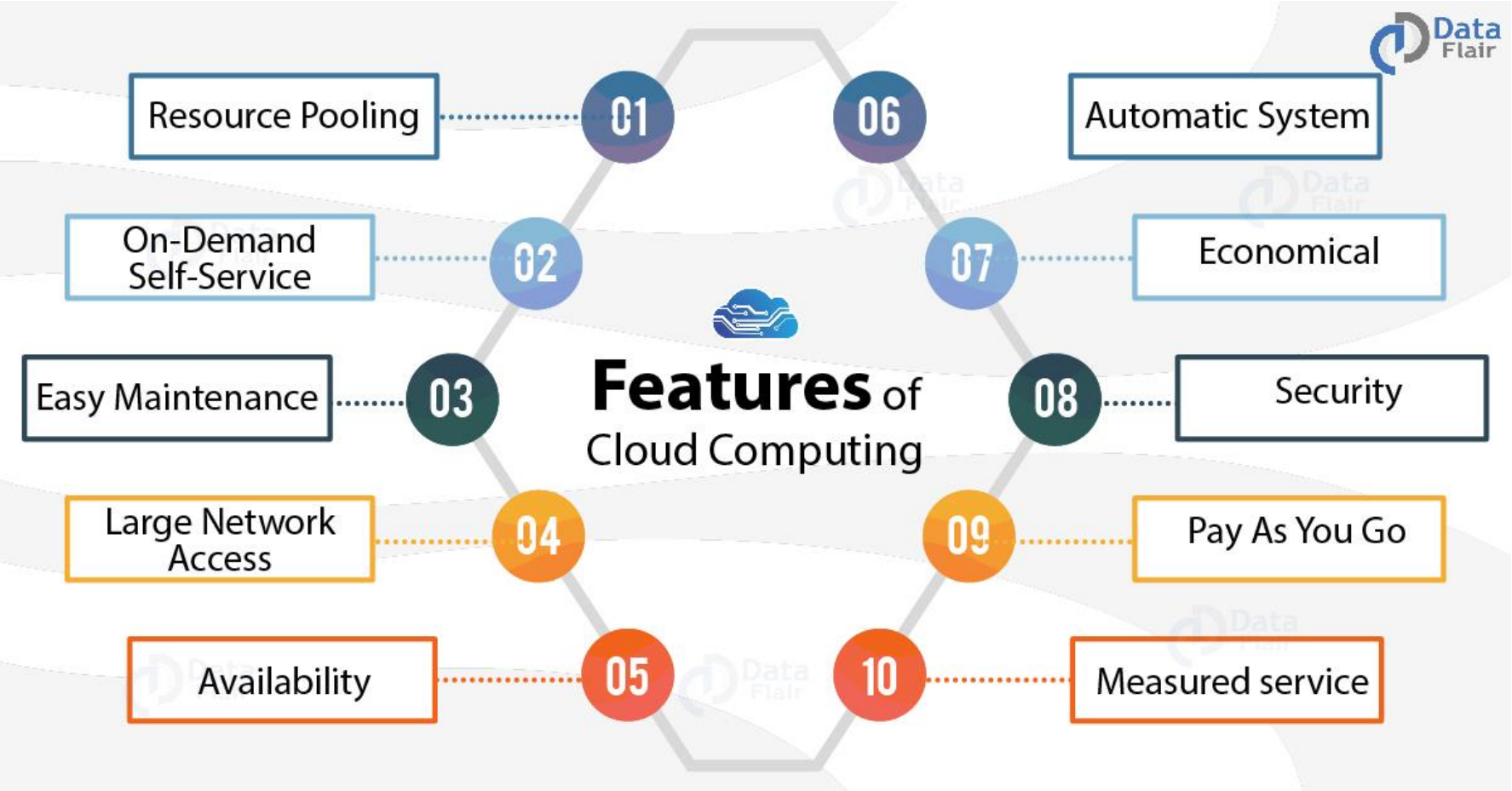
- Cloud Computing е общ термин, използван за описване на нов клас мрежови изчисления, който се осъществява през Интернет
  - Група от интегрирана и мрежова хардуерна, софтуерна и интернет инфраструктура (наречена платформа).
  - Използването на Интернет за комуникация и транспорт предоставя на клиентите хардуер, софтуер и мрежови услуги
- Платформите скриват сложността и детайлите на основната инфраструктура от потребителите и приложенията, като предоставят много прост графичен интерфейс или API (Applications Programming Interface).

Какво е облачно изчисление (Cloud Computing)? 2/2

- Облачните изчисления са общ термин, използван за обозначаване на интернет-базирани разработки и услуги
- Редица характеристики определят облачните данни, приложенията и инфраструктурата:
  - Отдалечно хостване: Услугите или данните се хостват в отдалечена инфраструктура.
  - Повсевместен (Ubiquitous) достъп: Услугите или данните са достъпни отвсякъде.
  - Commodified (to treat as a commodity): Резултатът е полезен изчислителен модел, подобен на този за традиционните комунални услуги, като предоставяне на населението на питейна вода, газ и електричество - *вие плащате за това, което бихте искали да използвате*!

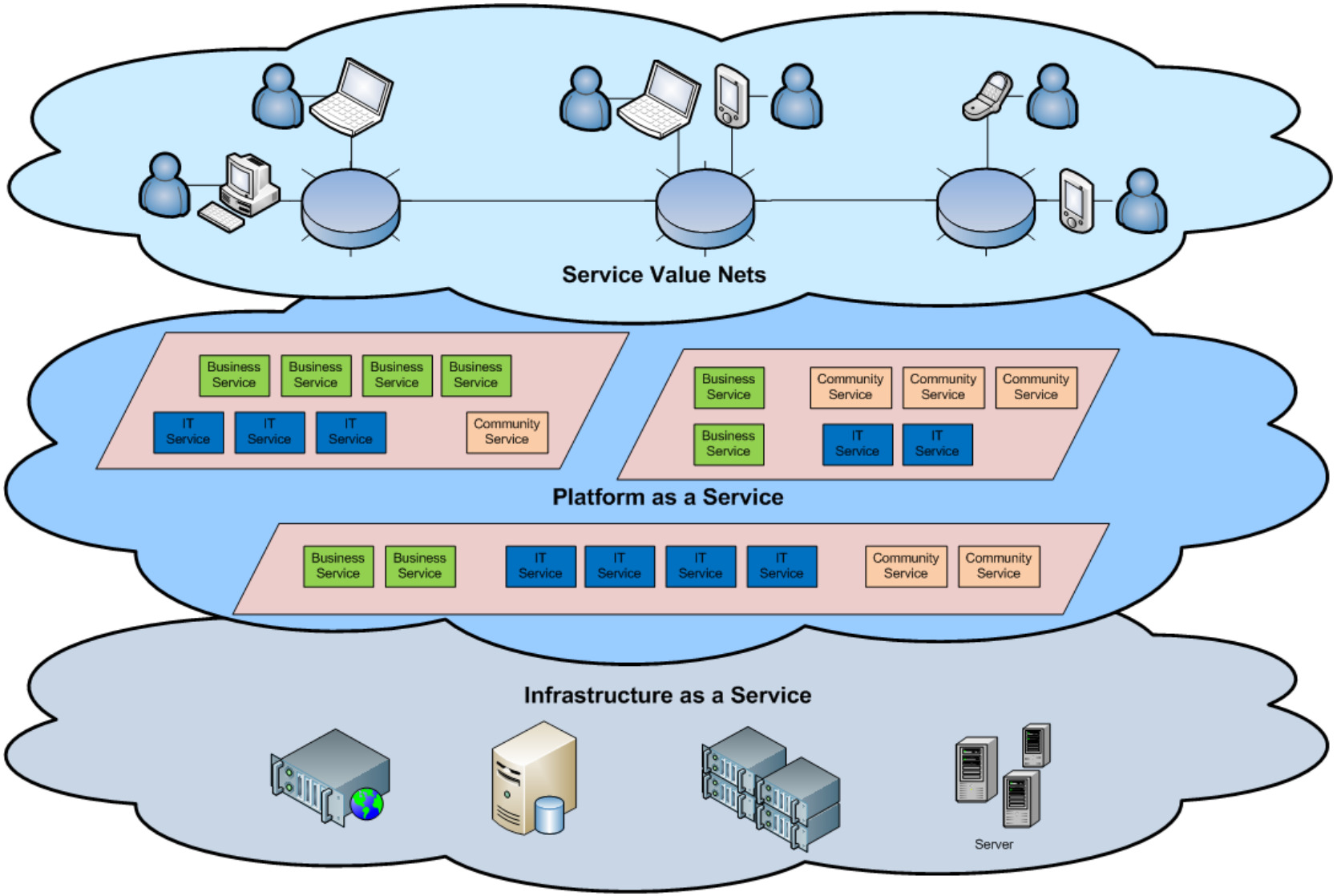


Характеристики



data-flair.training: Features of Cloud Computing – 10 Major Characteristics of Cloud Computing - DataFlair

Cloud Flavors?



SaaS
PaaS
IaaS

Software as a Service (SaaS)

- SaaS е методология за доставяне на софтуер, която предоставя лицензиран достъп на много наематели до софтуер и неговите функции от разстояние като Уеб-базирана услуга.
  - Обикновено се таксува въз основа на използване
  - Среда с много наематели (multi-tenant environment)
- Силно мащабируема архитектура
- Крайните потребители са потребителите на облака
  - Те използват приложения (имейл, календар и т.н.), които работят в облак
  - Не управляват наличната облачна инфраструктура

SaaS примери

facebook

Platform as a Service (PaaS)

- PaaS предоставя всички удобства, необходими за поддържане на пълния жизнен цикъл на изграждане и доставяне на уеб приложения и услуги изцяло през Интернет.
  - Обикновено приложенията се разработват с определена платформа
  - Multi-tenant среда
  - Високо мащабируема многостепенна архитектура

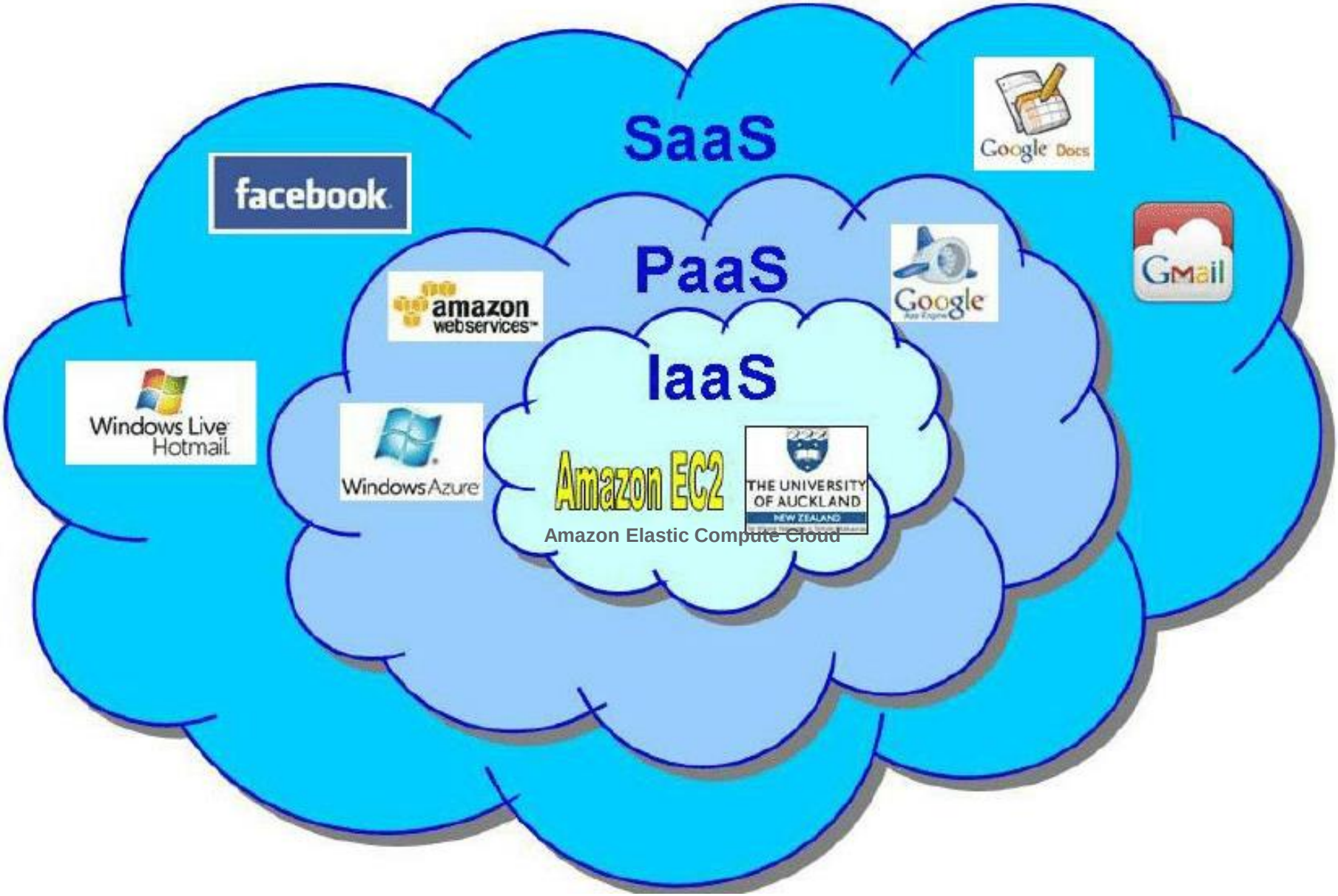
- Потребителите в облака са разработчици или системни администратори
  - Платформата предоставя услуги като: база данни, балансиране на натоварването, мащабируемост, среда за разработка
  - Потребителите внедряват своите приложения, използвайки възможности, предоставени от доставчика на облака
  - Потребителите не управляват облачната инфраструктура, но контролират внедрените приложения
  - Облачните доставчици предлагат интернет-базирана платформа за разработчици, които искат да създават услуги, но не искат да изграждат собствен облак

PaaS примери

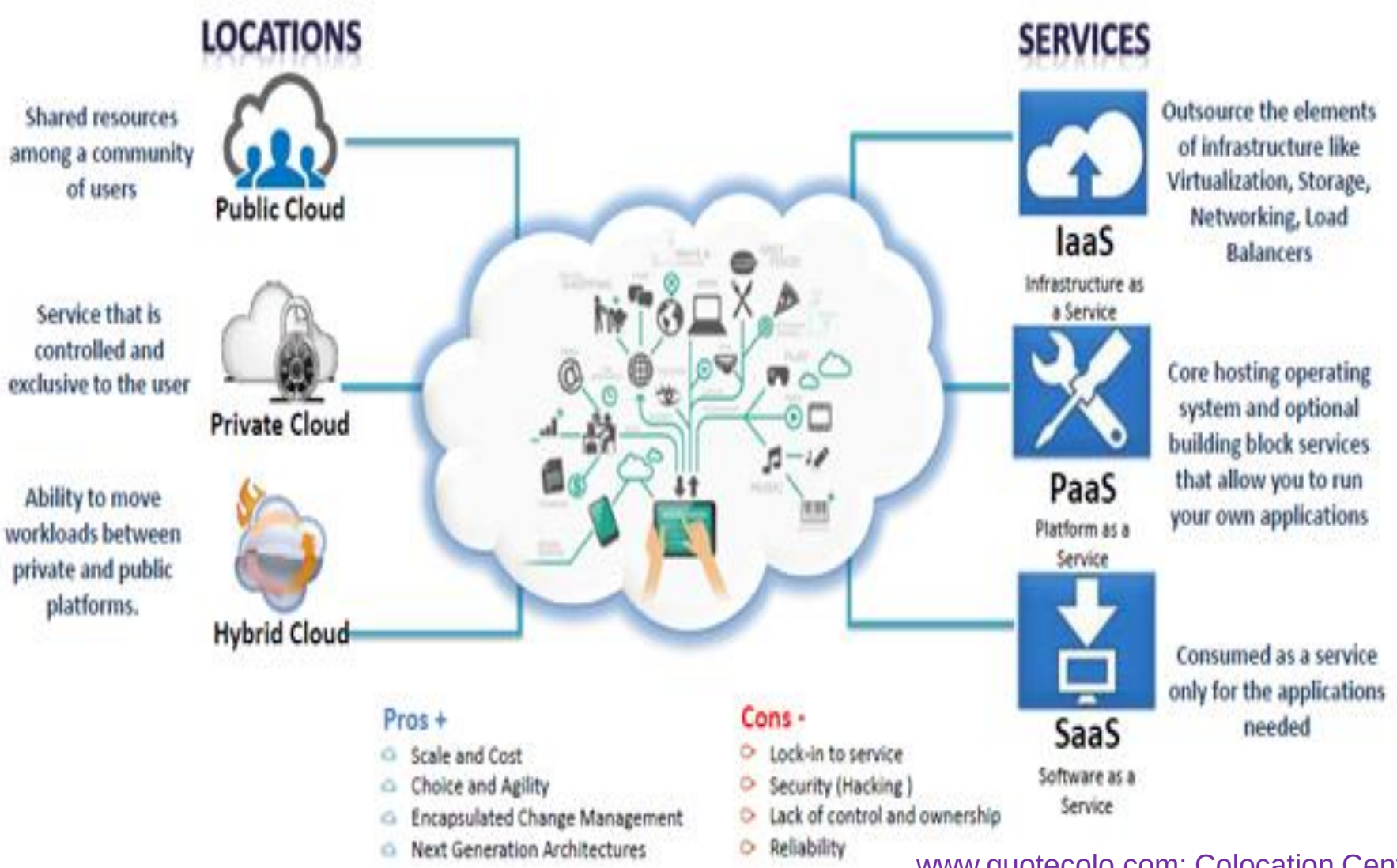
Infrastructure as a Service (IaaS)

- IaaS е доставката на технологична инфраструктура като услуга с цел мащабиране при поискване
  - Обикновено това е multi-tenant виртуализирана среда
  - Може да бъде съчетано с управлявани услуги за поддръжка на ОС и приложения
- Потребителите в облак са разработчици или системни администратори
  - Обработка, съхранение, мрежи и други ресурси, където потребителите внедряват и стартират софтуер
  - Потребителите имат контрол над операционните системи, съхранението, внедрените приложения ...
  - Потребителите в облака наемат хранилища, изчислителна мощност и поддръжка от доставчиците на облака (плащане според използването – pay-as-you-go)

IaaS примери


SaaS
PaaS
IaaS
facebook
Google Docs
Gmail
Google
amazon web services
Windows Live Hotmail
Windows Azure
Amazon EC2
THE UNIVERSITY OF AUCKLAND
NEW ZEALAND
Amazon Elastic Compute Cloud

What is Cloud Computing?



www.quotecolo.com: Colocation Centers Are Bucking the Cloud Data Center Trend - QuoteColo

Архитектурни стилове в облак

- Circuit breaker
- Queue
- Caching
- Sharding

Circuit breaker

- Изпълнението на дадена услуга може да блокира в разпределената среда
- Класическото решение е да се въведе интервал на изчакване (тайм-аут, timeout) за други услуги, които я извикват
- Това обаче може да доведе до ненужно потребление на ресурси в облачна среда
- Да приемем, че стотици потребители чакат за изпълнението на неуспешна услуга и всеки от тях изчаква timeout

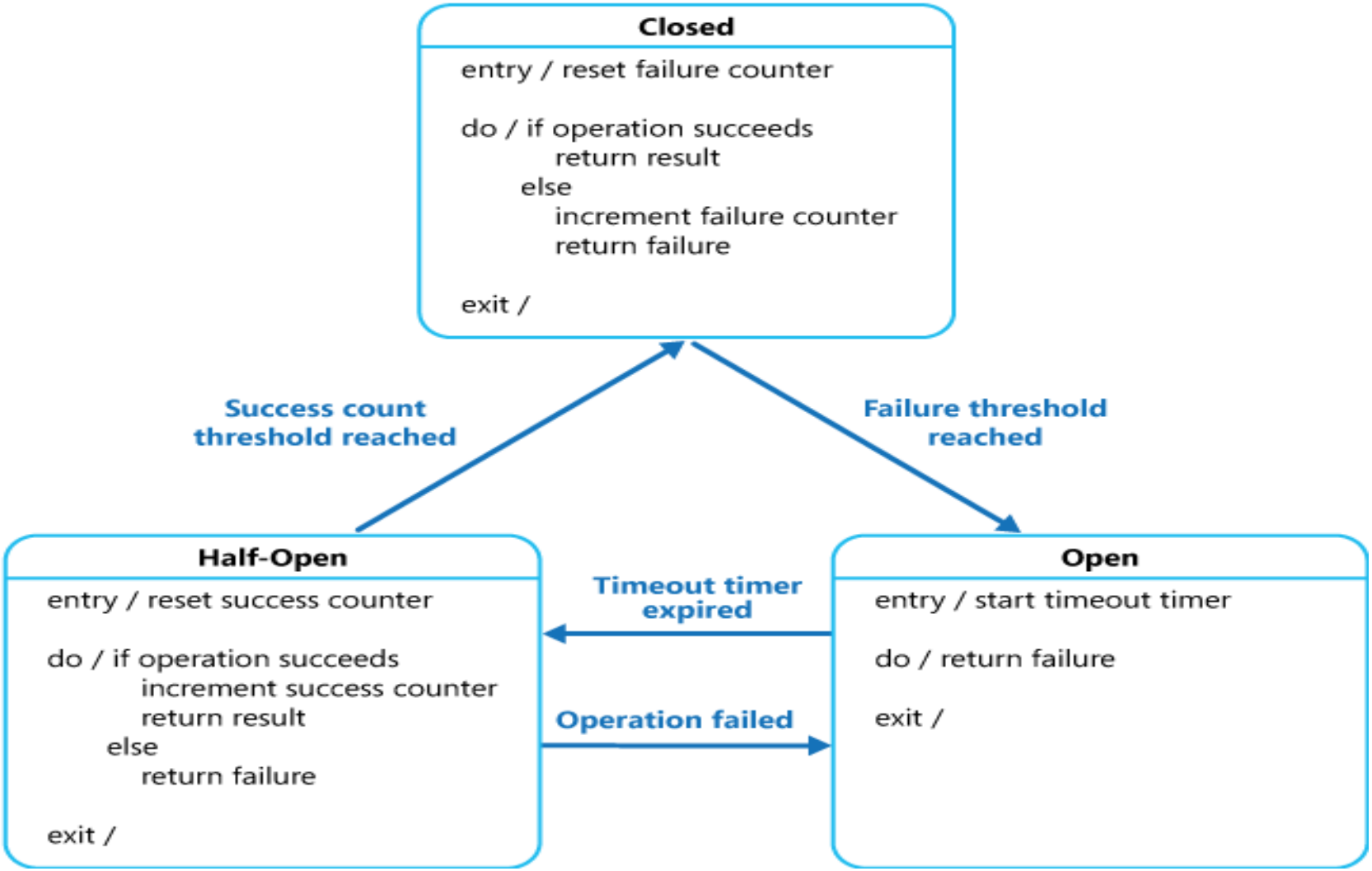
???


LIMBO SERVICES
WHAT GIVES, MAN? THIS LINE HASN'T MOVED IN HOURS!
CARTOONSTOCK .com
Search ID: jcon737

Circuit breaker шаблон

- Той не позволява на приложението да извърши операция, която е малко вероятно да успее
- Например извикване на услуга, за която вече е известно, че е неуспешна
- Действа като пълномощник (прокси) за операции, които могат да се провалят
- Следи броя на възникналите скорошни повреди и след това използва тази информация, за да реши дали да позволи операцията да продължи или просто незабавно да върне изключение

Circuit Breaker Pattern



Source: Cloud design patterns: http://msdn.microsoft.com/en-us/library/dn568099.aspx

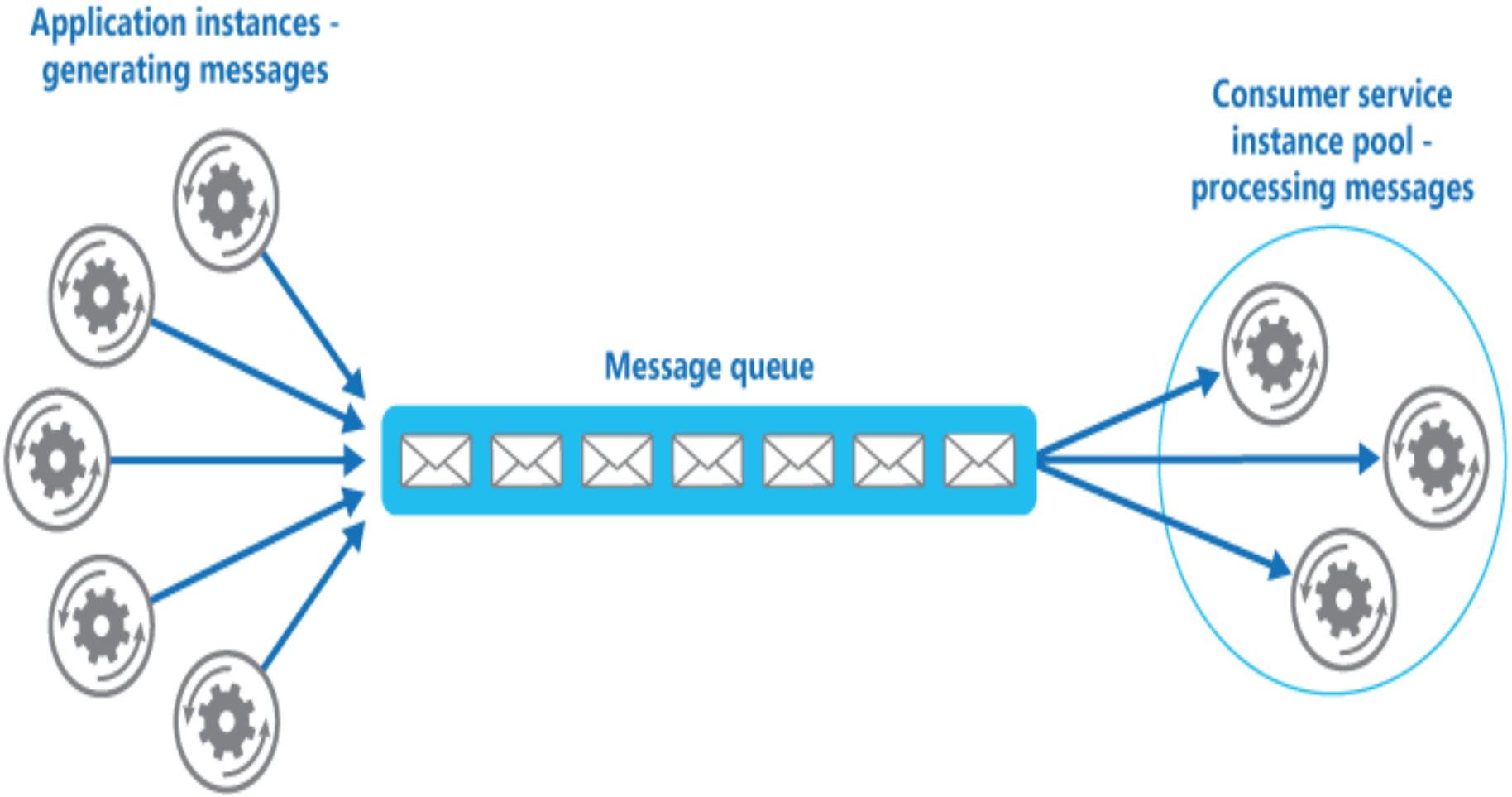
Circuit Breaker шаблон

- Спомага за увеличаване на надеждността (dependability) на системата
- Приложението със заявка или голям брой приложения няма да се налага да изчакват излишно за времеви интервал (timeout)
  - Когато се знае, че услугата е прекъсната
  - Мрежовата връзка временно е прекъсната
  - Известно е, че услугата е много заета и не може да отговори

Queue (опашка)

- Дефиниране на проблема
  - В облака може да имате услуги, които са залети от голям брой едновременни заявки от други услуги. В този случай те могат да бъдат претоварени или да изпитват пикови натоварвания.
  - Трябва да намерим решение за изглаждане на тежки товари, които могат да доведат до неуспех на услугата или изчакване на извикващата задача.
- Различни видове опашки
  - Стандартна опашка
  - Приоритетна опашка
  - Опашка с фиксирана дължина

Стандартна опашка (Standard queue)

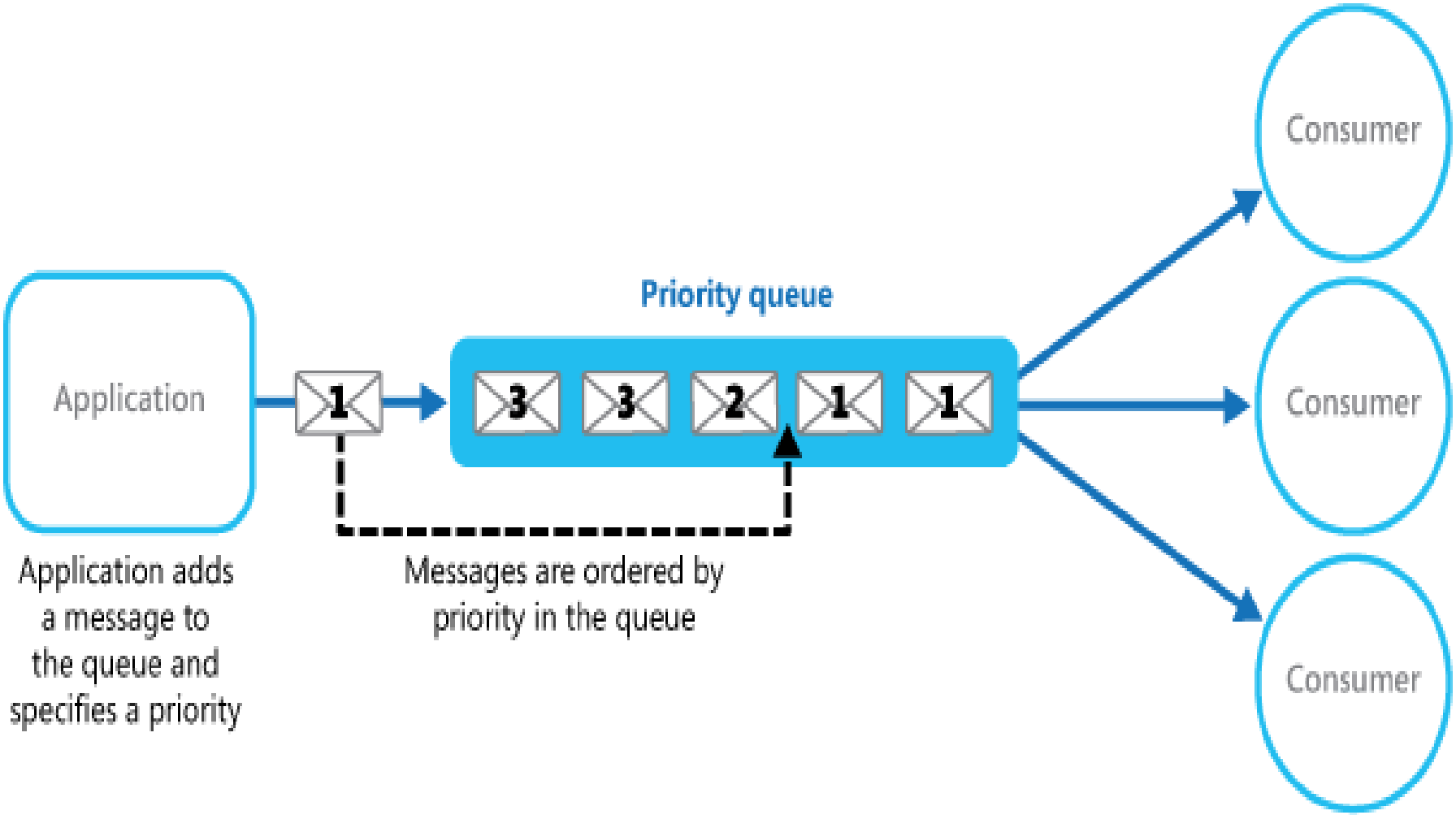


Source: Cloud design patterns: http://msdn.microsoft.com/en-us/library/dn568099.aspx

Standard queue

- Опашката действа като буфер, съхранявайки съобщенията, докато не бъдат извлечени от услугата.
- Услугата извлича съобщенията от опашката и ги обработва
- Минимизира рисковете за наличност чрез голям брой едновременни (конкурентни) заявки

Приоритетна опашка (Priority queue)



Source: Cloud design patterns: http://msdn.microsoft.com/en-us/library/dn568099.aspx

Priority queue

- Прилага политика за сортиране на входящите заявки според техния приоритет
- Приоритетът на заявката може да бъде определен от:
  - изпращащите приложения
    или от
  - самата опашка

Опашка с фиксирана дължина (Fixed length queue)

- Можем да проектираме опашката, за да изпратим изключение, когато бъде достигнато определено (изключително високо за услугата) количество съобщения

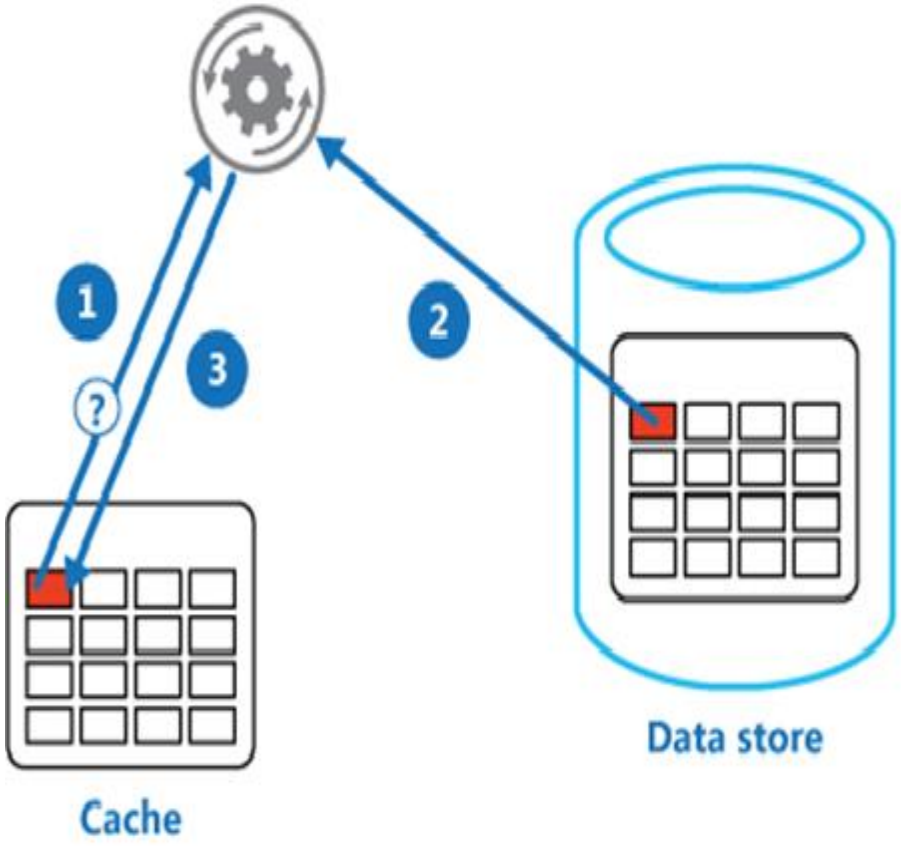
- В този случай заявяващата задача ще знае, че нейната заявка няма да бъде обработена и може да предприеме подходящи действия, без да чака изчакване (timeout)

Cache (кеш)

- Използва се за оптимизиране на многократния достъп до информация, съхранявана в хранилище на данни

- Дали кешираните данни винаги ще бъдат напълно съвместими (консистентни) с данните в хранилището на данни?

Cache



- **четене**

1. Търси елемент в кеша
2. Ако го няма, извлича го от хранилището
3. Съхранява копие в кеша

- **писане**

1. Прави модификация в хранилището на данни
2. Инвалидира съответния елемент в кеша.

Sharding стил

- Целта е да се увеличи мащабируемостта (scalability) при използване на големи количества данни
- Идеята е да се раздели хранилището на данни на набор от хоризонтални дялове, наречени парчета (shards)
- Мотивацията тук е да се увеличат:
  - Мястото за съхранение
  - Изчислителните ресурси
  - Мрежова пропусквателна способност (Network bandwidth)
  - Географското разделение

Sharding стил – особености

- Важно е решението как да се разделят данните между парчетата (shards)
  - Едно парче (shard) обикновено съдържа елементи, които попадат в определен диапазон, определен от един или повече атрибути на данните.
  - Атрибутите на данните формират ключа за разделяне (*shard key),* известен още като ключ за дял (*partition key*).
  - Shard key трябва да е статичен. Той не трябва да се основава на данни, които могат да се променят.
- Шардингът физически организира данните - когато приложението съхранява и извлича данни, sharding логиката насочва приложението към подходящия shard

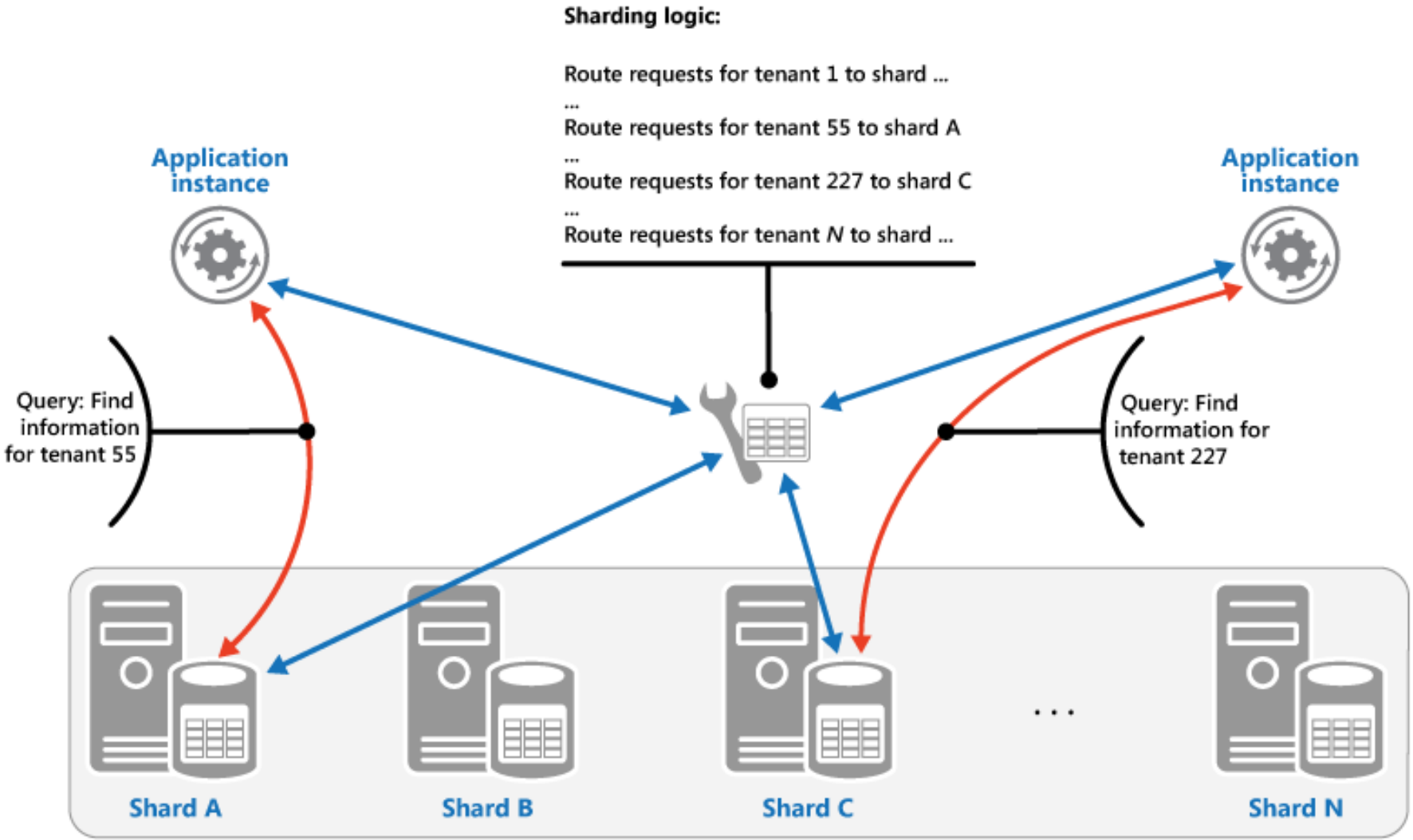
Sharding стил – възможни проблеми

- Проблеми с имплементацията на sharding логиката
  - Може да се реализира като част от кода за достъп до данни в приложението
  - Може да бъде имплементирана от системата за съхранение на данни, ако тя прозрачно поддържа sharding

Sharding стил – използване

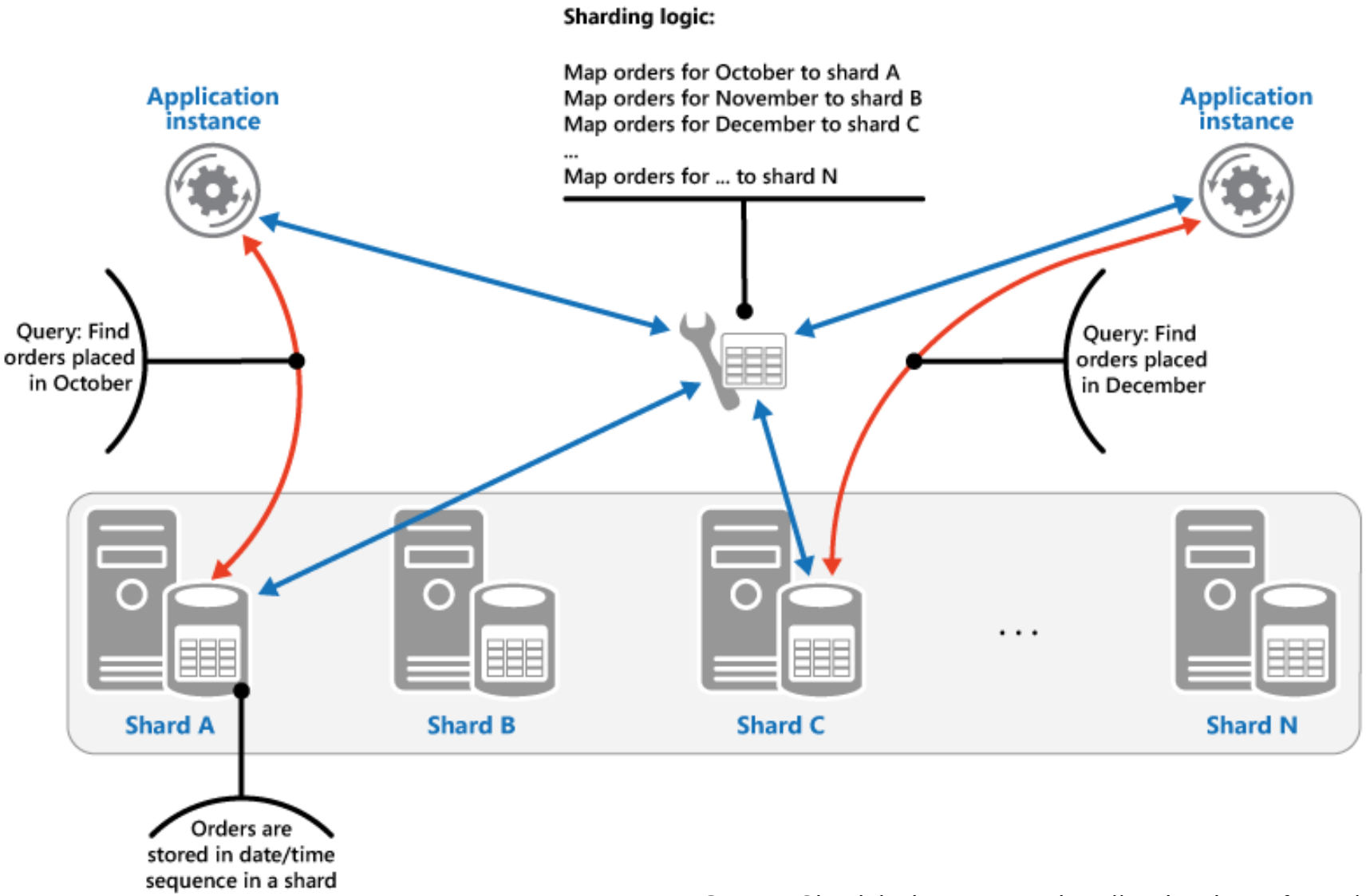
- Съществуват три основни стратегии за прилагане на логиката за sharding
  - Стратегия за търсене (Lookup)
  - Стратегия за обхват (Range)
  - Хеш (Hash) стратегия

Lookup sharding стратегия



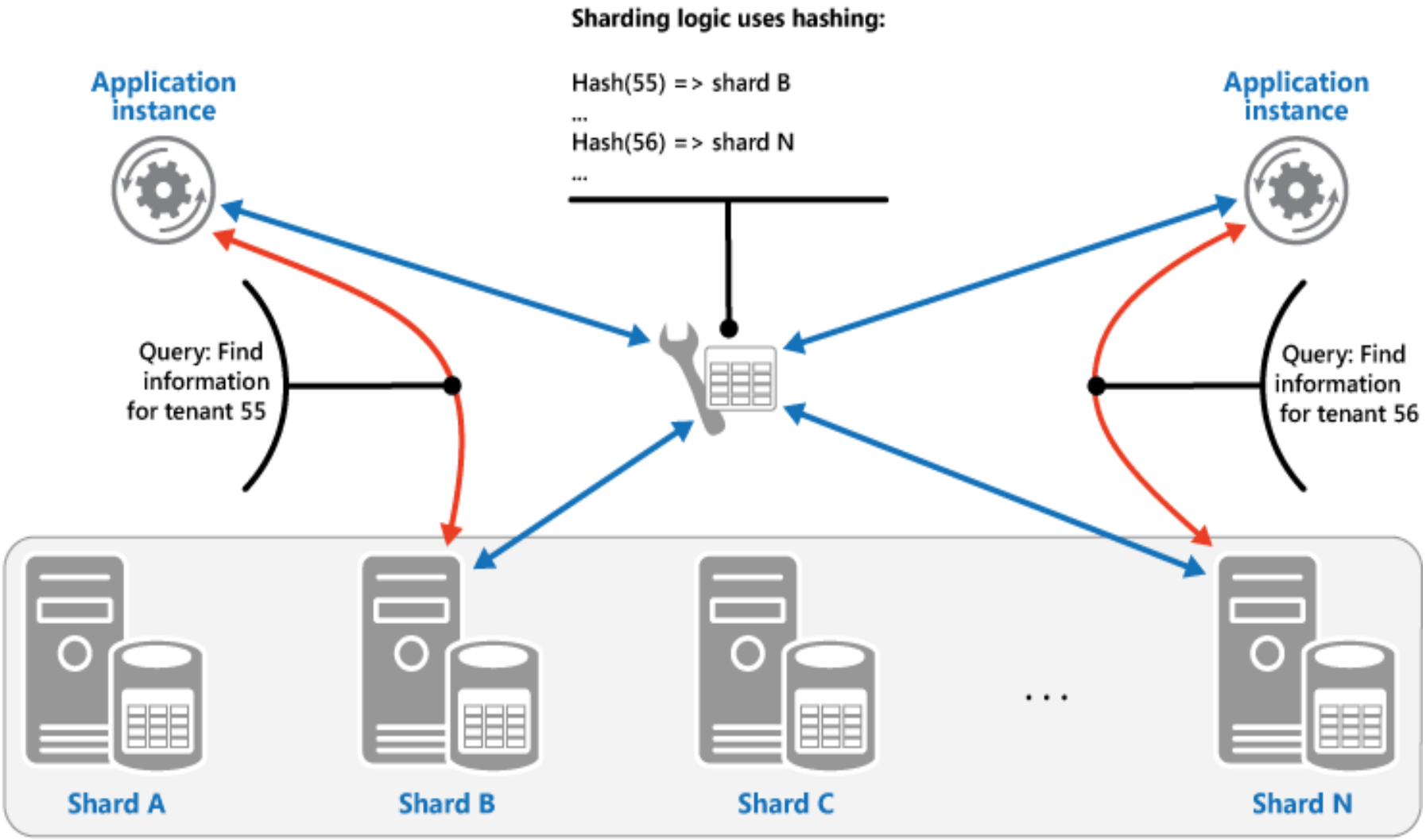
Source: Cloud design patterns: http://msdn.microsoft.com/en-us/library/dn568099.aspx

Range sharding стратегия



Source: Cloud design patterns: http://msdn.microsoft.com/en-us/library/dn568099.aspx

Hash sharding стратегия



Source: Cloud design patterns: http://msdn.microsoft.com/en-us/library/dn568099.aspx

Sharding – предимства

- По-добро управление на данните - предоставя абстракция на физическото местоположение на данните
- Контрол върху кои парчета (shards) съдържат дадени данни
- Повишена производителност на хранилището за данни

Sharding – недостатъци

- Проблеми с производителността на приложението:
  - Допълнителни разходи (Overhead) за определяне на местоположението на всяка информация
  - По-голями разходи, когато данните, които съответстват на една заявка, се разпределят между много части
- Shards може да съдържат небалансирано количество данни
- Понякога е изключително трудно да се проектира shard key, който да съответства на изискванията на всяка една възможна заявка за данни

Въпроси?