中国科学技术大学
University of Science and Technology of China

Software Architecture

SSE USTC Qing Ding
dingqing@ustc.edu.cn
http://staff.ustc.edu.cn/~dingqing

Conceptual Architecture

Outline

- Definition
- Designing Conceptual Architecture
- Behaviour Model
- Component Stereotypes
- Design Guidelines

Simplified Workflow

Exemplary, simplified step-by-step guide to start the conceptual architecture 示范性的、简化的逐步指导，以启动概念架构

1. Identify actors with the systems (e.g. external system, users) 系统参与者
2. Identify data flow 数据流
3. Identify functional requirements
4. Identify non functional requirements
5. Prioritise requirements 需求优先级
6. Define use-cases 用例
7. Define data items 数据项目

Model abstraction and granularity level

模型抽象和粒度级别

- Models can be created at varying levels of abstraction and details E.g., an object-oriented model of a car

  模型可以在不同的抽象和细节级别上创建，例如，汽车的面向对象模型
- At the lowest detail level: car has engine, wheels, brakes, etc.

  在最低的细节水平:汽车有引擎，车轮，刹车等。
- At the highest detail level: car objects with variables, interacting with other objects

  在最高的细节级别:带有变量的car对象，与其他对象交互
- OO programming model **abstracts** assembly, operating system, hardware, ...

  面向对象编程模型对程序集、操作系统和硬件进行抽象

Definition

- A software architecture is a collection of models of a software system at various abstraction and detail levels. The models describe:

软件体系结构是软件系统在各种抽象和细节级别上的模型集合

- the system as a whole system components component connections

系统作为一个整体系统组件，组件连接

组件如何交互以实现系统目的

- how component interact to fulfill the system purpose.

Analysis

需求分析的结果

The results of the requirements analysis:

1. Functional requirements
2. Non-functional requirements
   (a)Runtime qualities
   (b)Non-runtime qualities
3. Contextual requirements

Design

- Design is the process of creating models (recollect the 设计是创建模型(回想一下SA的定义)的过程，这是两种基本类型的架构模型 definition of SA) Two basic types of architectural models
- Structure and behaviour
- **Architectural structure** is a 静态 static model of a system (i.e. 架构结构是系统的静态模型(即系统如何划分为组件) how the system is divided into components)
- **Architectural behaviour** is a 动态 dynamic model of a system 架构行为是一个系统的动态模型（即组件如何相互交互以执行一些有用的工作) (i.e. how the components interact with each other to perform some useful work)

Architectural views

我们可以从不同的角度来研究一个系统

- We can examine a system from different points of view

  Different kinds of views

- **Conceptual**: components are set of responsibilities and

  概念:组件是一组职责，连接器是信息流

  connectors are flow of information

- **Execution**: components are execution units (processes) and

  执行:组件是执行单元(进程)，连接器是进程之间的消息

  connectors are messages between processes

- **Implementation**: components are libraries, source code,

  实现:组件是库、源代码、文件等，连接器是协议、api调用等

  files, etc and connectors are protocols, api calls, etc.

Definition

What is conceptual architecture?

Conceptual Architecture

- 关注域级别的职责
  Focuses on **domain-level responsibilities**
- 最初的架构设计
  Initial architectural design
- 对涉众需求的第一个响应
  First response to stakeholder needs
- 需求分析设计
  Design by analysing requirements
- Contains components and connectors (box-and-line)
- 静态
  → Provides an at a glance overview of the structure
  提供系统功能方面的结构概览
  of a system in terms of its functionality

Components in conceptual architecture

一组相关的域级职责

- Set of related domain-level responsibilities
- Initially, responsibilities arise from functional requirements

最初，职责来自于功能需求

内部迭代

- However, design is an iterative process

设计是一个迭代的过程

- Further iterations take into account non-functional requirements

进一步的迭代会考虑非功能性需求

Connectors in conceptual architecture

- Connectors indicate that connected components exchange information
  连接器指示被连接的组件交换信息
- 箭头表示信息流
  Arrows indicate information flow
- 标签描述了信息的种类
  Labels describe kind of information
- In some cases two-directional connectors: labels should be put near arrows
  在某些情况下，双向连接器:标签应该放在箭头附近
- Conceptual components often have no direct counterpart in the final software
  概念上的组件通常在最终的软件中没有直接的对等物
- 不需要关心物理位置
  ... no need to care about physical location

Simple example

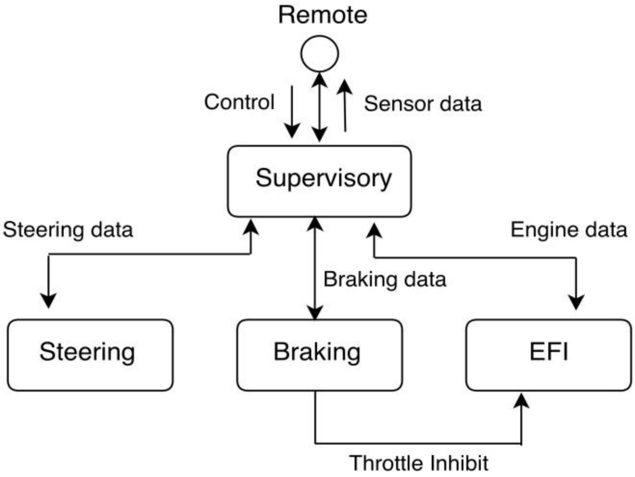


Figure: Model-car control system from Software Architecture Primer

Simple example

- Supervisory
  - Receive commands and decode them
  - Send command data to real-time components
  - Send selected data to remote interface

监督
· 接收命令并解码
· 将命令数据发送给实时组件
· 将选择的数据发送到远程接口

- Braking
  - Apply braking force in percentage amount
  - Generate throttle inhibit signal

制动
· 以百分比为单位施制动力
· 产生节流抑制信号

- Steering
  - Set steering gear to selected angle

- EFI
  - Control spark and injection timing
  - Control amount of fuel injected

EFI
· 控制火花和喷射时间
· 控制喷油量

Designing Conceptual Architecture

How to approach conceptual architecture design?

Design process

根据需求创建初始的概念架构

1. Create the initial conceptual architecture from requirements

专注于功能来精化

2. Elaborate focusing on functionality

关注质量属性(非功能需求、上下文需求)来精化

3. Elaborate focusing on quality attributes (non-functional requirements, contextual requirements)

4. Iterate over 2 and 3

Initial steps

- Identify key concepts 识别关键概念
  - •符合要求
  - •在叙事中
  - ▸ Within requirements
  - ▸ Within narratives

将关键概念划分为类别

- Assign key concepts to categories
  - ▸ Data, Function, Stakeholder, System, Abstract Concept

用户

The first step

- Underline the key concepts in the requirements (ask yourself does this concept relates to the functionality?)
  强调在需求中的关键概念(问问自己这个概念是否与功能相关?)
- Copy key concepts onto a sheet of paper (consider each one to see if it is a viable component)
  将关键概念复制到一张纸上(考虑每一个概念，看看它是否是一个可行的组成部分)
- Draw the components and add connectors (add arrows and labels)
  绘制组件并添加连接器(添加箭头和标签)

Example Application - Functional requirements

该系统是一个导航工具。系统可以计算出以下最优路径

- **UR1:** The system is a navigation tool. The system can calculate the following optimal routes.
- ▶ **UR1.1:** Shortest path
- ▶ **UR1.2:** Fastest route
- ▶ **UR1.3:** Most economical (lowest CO2 emissions)
- ▶ **UR1.4:** Cheapest
- ▶ **UR1.5:** Pleasant
- ⋆ **UR1.5.1:** Quietest
- ⋆ **UR1.5.2:** Most sightseeing spots
- ⋆ **UR1.5.3:** Along rivers and through parks

Example Application - Functional requirements

- **UR2:** The places are stored in a database
- **UR3:** The connections between the places are stored in the database
- **UR4:** The connections need to contain the transportation mode and provider
- **UR5:** The administrator can add/remove new places and connections
- **UR6:** The user can search for places

Example Application - Functional requirements

- **UR7:** The system needs to interact with external services
  - ▶ **UR7.1:** The system needs to buy tickets for the tramway
- **UR8:** The system needs to draw the route on a interactive map
- **UR9:** The system needs to provide user management
  - ▶ **UR9.1:** Register new users
  - ▶ **UR9.2:** Log-in / Log-out
  - ▶ **UR9.3:** Store user preferences
  - ▶ **UR9.4:** Store credentials for purchase
- …

The second step

- Assign every possible concept from the requirements to a category:

  从需求中分配每一个可能的概念到一个类别中

- **Data**: information that is stored, processed, etc. → Not directly a component but you might need components for data management

  数据:被存储、处理等的信息，不直接是组件，但可能需要组件进行数据管理

- **Function**: Something done by something → typically components

  功能:由某物完成的某事→典型的组件

- **Stakeholder**: users, organizations → never components

  利益相关者:用户、组织→不是组件

The second step

- Assign every possible concept in a system narrative to a category:
  将一个系统描述中的每一个可能的概念分配到一个类别
- **System**: external systems → sometimes you need an interface component
  系统:外部系统→有时你需要一个接口组件
- **Hardware** → physical components
  硬件→物理组件
- **Abstract concept**: explanation of something → rarely components With the goal to identify all components...
  抽象概念:解释某物→很少成分，目的是识别所有组件

Key Concepts

- navigation tool

Key Concepts

- navigation tool **Abstract concept**
- calculate

Key Concepts

- navigation tool **Abstract concept**
- calculate **Function**
- optimal route

Key Concepts

- navigation tool **Abstract concept**
- calculate **Function**
- optimal route **Abstract concept**
- shortest path

Key Concepts

- navigation tool **Abstract concept**
- calculate **Function**
- optimal route **Abstract concept**
- shortest path **Abstract concept**
- fastest route

Key Concepts

- navigation tool **Abstract concept**
- calculate **Function**
- optimal route **Abstract concept**
- shortest path **Abstract concept**
- fastest route **Abstract concept**
- most economical

Key Concepts

- navigation tool **Abstract concept**
- calculate **Function**
- optimal route **Abstract concept**
- shortest path **Abstract concept**
- fastest route **Abstract concept**
- most economical **Abstract concept**
- cheapest

Key Concepts

- navigation tool **Abstract concept**
- calculate **Function**
- optimal route **Abstract concept**
- shortest path **Abstract concept**
- fastest route **Abstract concept**
- most economical **Abstract concept**
- cheapest **Abstract concept**
- pleasant

Key Concepts

- navigation tool **Abstract concept**
- calculate **Function**
- optimal route **Abstract concept**
- shortest path **Abstract concept**
- fastest route **Abstract concept**
- most economical **Abstract concept**
- cheapest **Abstract concept**
- pleasant **Abstract concept**

Key Concepts

- places **Data**
- database **Data**
- connections **Data**
- transportation mode **Data**
- administrator **Stakeholder**
- add/remove (places) **Function**
- user **Stakeholder**
- search **Function**

Key Concepts

external services **System**

buy (ticket) **Function**

ticket **Data**

tramway **Data**

draw **Function**

route **Data**

interactive map **Abstract concept**

user management **Function**

register **Function**

users **Data**

log-in / log-out **Function**

user preference **Data**

credentials **Data**

purchase **Function**

Categorisation

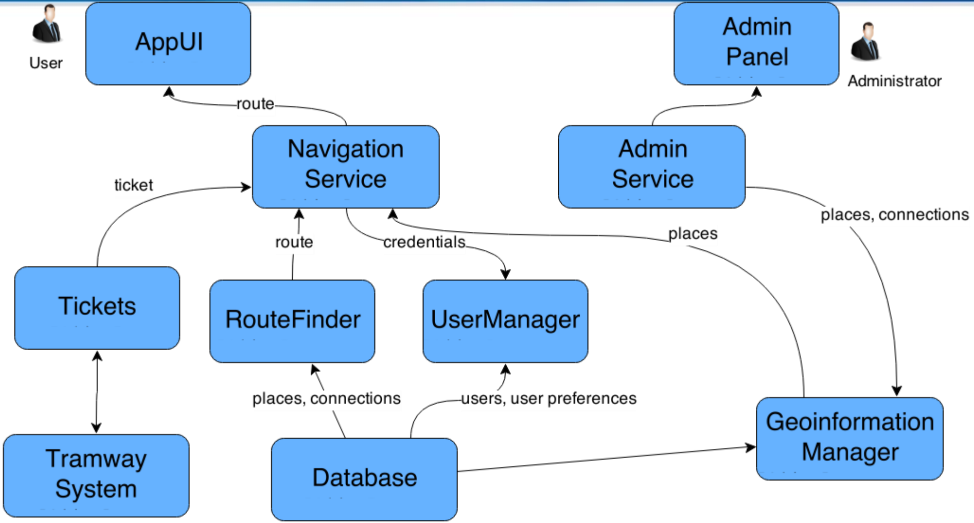
Data	Function	Stakeholder	System	Abs.concept
places	calculate	administrator	external services	navigation tool
database	add/remove (places)	user		optimal route
connections	search			shortest path
transp. mode	buy (ticket)			fastest route
tramway	draw			most economical
route	user management			cheapest
users	register			pleasant
user preference	log-in / log-out			interactive map
credentials	purchase			
ticket	download			
	register			
	log in			

Conceptual Architecture: Best Practice

- Start with stakeholders and external systems
- Think about the data, data exchange and data storage
- Think about active components, that trigger events
- Combine the functions into groups of related functionality
- Optionally, use narratives as guide how to group the components
- → draw a first iteration
- Validate the first iteration

- 从涉众和外部系统开始
- 想想数据、数据交换和数据存储
- 想想那些触发事件的活动组件
- 将功能组合到相关功能组中
- 可以选择使用说明来指导如何对组件进行分组
- 绘制第一次迭代
- 验证第一次迭代

Conceptual Architecture: Iteration 1



Component responsibilities

- AppUI
  - ShowPlaces
  - DisplayMap
  - DrawRoute
  - DisplayTicket
- AdminPanel
  - ListPlaces
  - ListConnections
- NavigationService
  - StartCalculation
  - BuyTicket
  - RouteComputed
  - TicketPurchased
- AdminService
  - AddPlace
  - RemovePlace

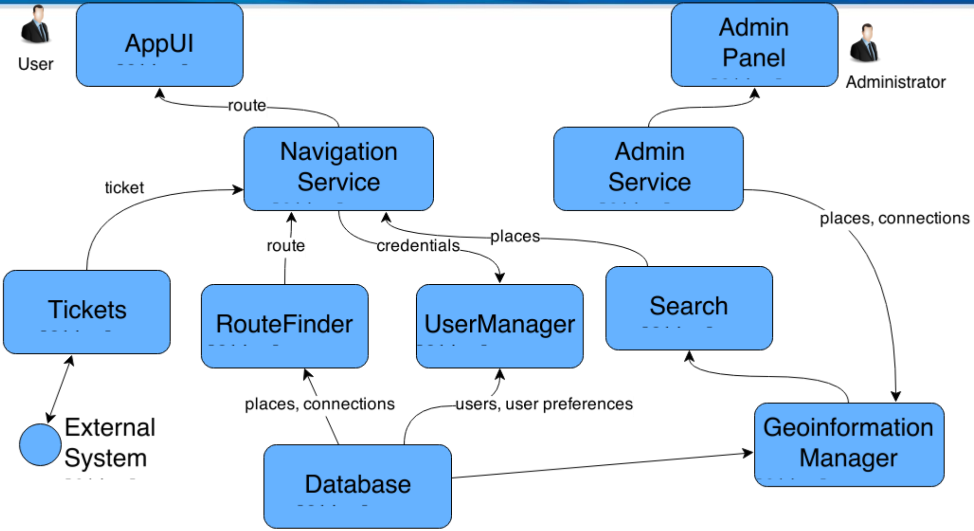
Component responsibilities

- RouteFinder
  - ComputeRoute
- Tickets
  - ListPrices
  - StartPurchase
- UserManager
  - RegisterUser
  - LoginUser
  - GetSeasonalTickets
- GeoinformationManager
  - AddPlace
  - RemovePlace
  - SearchPlace

Iterations

- Iterate over functional and non-functional requirements:
  在功能性和非功能性需求上进行迭代
- 功能似乎没有问题
  Functional seems to be OK
- 非功能性：性能作为质量属性
  Non-functional: performance as a quality attribute
- → Factor out the search functionality (e.g. search for places, ...) Also, design for change
  排除搜索功能(例如搜索地点，...)另外，为变化而设计
- → Abstract external systems through interface (e.g. new public transportation system)
  通过接口的抽象外部系统(例如新的公共交通系统)

Conceptual Architecture: Iteration 2



Component responsibilities

- Search
  - SearchPlaces
- PublicTransportAPI
  - ListPrices
  - BuyTicket

Behaviour Model

The dynamic aspect of conceptual architecture?

Behavioural exploration

到目前为止，只有结构架构

- Until now only structural architecture

我们需要考虑系统的行为

- We need to take into account behaviour of the system

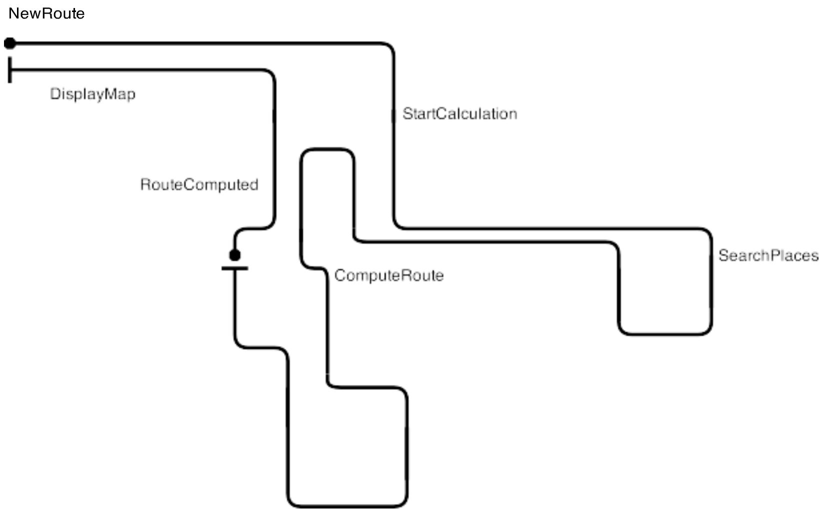
通常通过使用说明来完成

- Typically accomplished through usage narratives

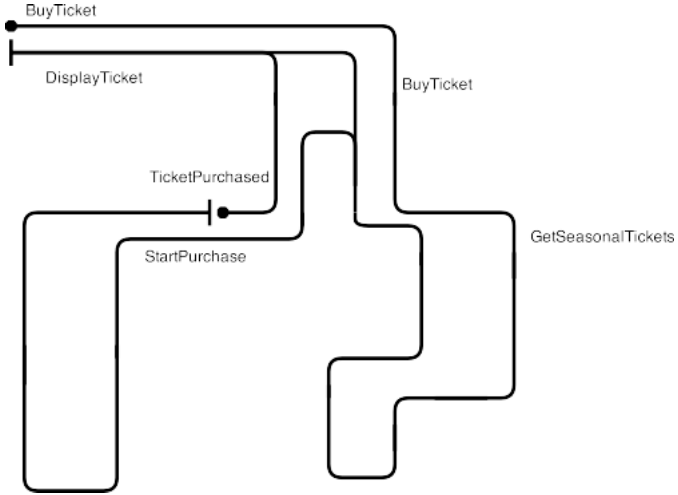
我们可以采用两个或三个使用场景并绘制用例图

- We can take two/three usage scenarios and draw use-case maps

Behavioural exploration



Behavioural exploration



Component Stereotypes

概念架构中的组件和连接器类型

Types of components and connectors in conceptual architecture

Component stereotypes

通过原型标记向组件添加语义

- Adding semantics to components through stereotypes Tagging

表示组件:表示某些属性的组件

- Presentation component: components to indicate certain properties

与用户交互

- interactions with users

持久存储:持久数据或来自外部系统的数据

- Persistent storage: persistent data or data from external systems

实时组件:“快速 ” 处理请求的组件

- Real-time components: components that handle requests “quickly”

Component stereotypes

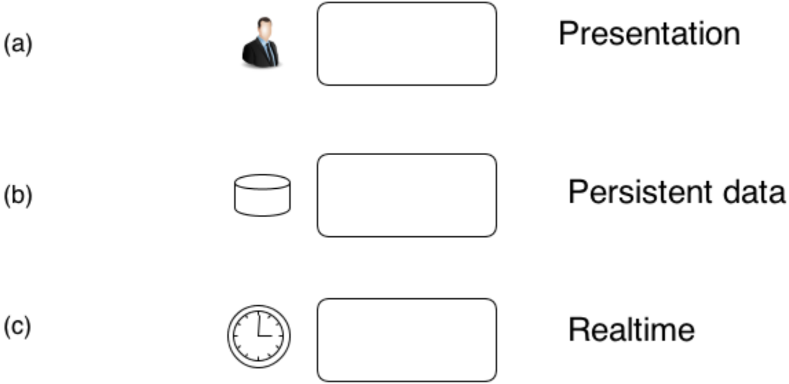
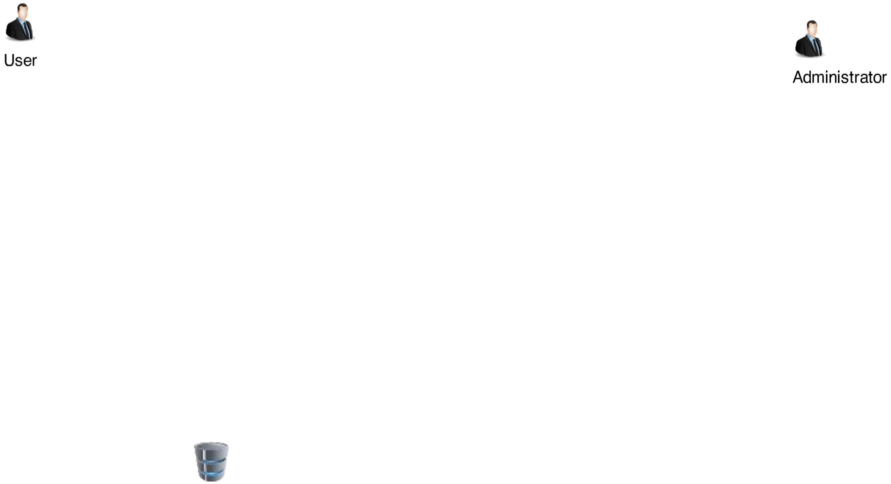


Figure: Source: Software Architecture Primer

Component stereotypes



Data Models

- 数据模型捕获域内信息的本质
- 如果组件之间交换的数据很复杂，我们可能需要创建数据模型
- 在我们的例子中:数据集和计算
- 在这两种情况下，我们都需要定义一种格式，例如UML

- Data models capture the nature of information in the domain
- If data exchanged between components is complex we might need to create data models
- In our case: datasets and calculations
- In both cases we need to define a format, e.g. UML

Script of Conceptual Architecture

Steps of developing the conceptual architecture

1. Clarify components & responsibilities
   - ▶ → e.g. by use of use-case maps
2. Clarify connectors
   - ▶ → e.g. by labelling information flow
3. Evaluate behaviour (against expected scenarios)
4. Identify stereotypes & structures
5. Define the data structure & models Simplify
6. architecture

开发概念架构的步骤

- 阐明组件,责任
  如，通过使用用例映射
- 阐明连接器
  如，通过标记信息流
- 评估行为(针对预期场景)
- 识别的原型,结构
- 定义数据结构，模型简化
- 体系结构

Design Guidelines

Practical aspects of conceptual architecture?

Conceptual arch. design rules

- **Guidelines for defining components**
- Granularity Cohesion Coupling
  - 定义组件的指导原则
  - 粒度内聚耦合

Conceptual arch. design rules

- 组件的粒度
- 分配给单个组件的功能数量
- 取决于组件的上下文

- **Granularity of components**
- Amount of functionality assigned to a single component
- Depends on the context of the component

Conceptual arch. design rules

- Avoid **blobs**: components that have too much responsibilities
- Recognized by: one component has all the responsibility, others have single responsibilities
- Reason 1: too much details in responsibilities
- Reason 2: too lazy too divide responsibilities
- Put more effort in dividing responsibilities

Conceptual arch. design rules

- 避免指令集群:两个组件都具有单个职责，但都是相互连接的
- 可以识别为:一组组件之间有太多的连接器
- 原因:责任实际上是相关的
- 将组件组合成单个组件

- Avoid **command clusters**: a couple of components all with a single responsibility but all interconnected
- Recognized by: too many connectors between a set of components
- Reason: responsibilities are in fact related
- Combine components into a single component

Cohesion

- 内聚性:组件的职责相互关联的程度
- 如果关系高，那么凝聚力也高
- 高内聚被认为是好的，因为它使架构更容易理解
- 将有助于维护系统
- 有助于设计可重用的组件

- Cohesion: how well are the responsibilities of a component related to each other
- If the relationship is high, then the cohesion is high as well
- High cohesion is considered to be good, as it makes the architecture easier to understand
- Will help to maintain the system
- Will help to design components that will be reused

Coupling

- Coupling: degree to which software components **depend** on each other
- The degree is influenced on a couple of aspects (more than in OO languages)
- E.g. how generic is the protocol?

· 耦合:软件组件相互依赖的程度
· 程度受几个方面的影响(比OO语言的影响更大)
· 例如:协议有多通用?

Degree of Coupling

- **Loose coupling** vs. **tight coupling**
- These are the both extremes
- In existing systems this is often a continuum
- Components might be tightly in one aspect and loosely coupled in another

松散耦合vs.紧密耦合
· 这是两个极端
· 在现有系统中，这通常是连续的
· 组件可能在一个方面紧密耦合而在另一个方面松散耦合

Aspects of Coupling

Level	Loose Coupling	Tight Coupling
Physical coupling	Physic intermediary	Direct physical link required
Communication style	Asynchronous	Synchronous
Type system	Data-centric, self-contained messages	OO-Style, complex object trees
Control of process logic	Distributed logic components	Central control
Platform dependencies	OS- and programming language independent	Strong OS and programming language dependencies

Consequences of Tight Coupling

- If one software component changes, the other (dependant) components need to be changed as well
- API/protocol specific to the needed requirements

  Often enforced by cross-cutting concerns
- Typically it is easier (less effort) to create tightly coupled components

- 如果一个软件组件改变了，那么其他(依赖的)组件也需要改变
- 特定于所需需求的API/协议，通常由横切关注点强制执行
- 通常，创建紧密耦合的组件更容易(工作量更小)

Consequences of Loose Coupling

- Components can be easily exchanged
- Components will be better suited to be reused
- Components are required to be more generic
- E.g. use standard protocols instead of custom ones
- Will typically require more effort and better design/planning

- 组件可以很容易地交换
- 组件将更适合被重用
- 组件需要更加通用
- 例如，使用标准协议而不是自定义协议
- 通常需要更多的努力和更好的设计/规划

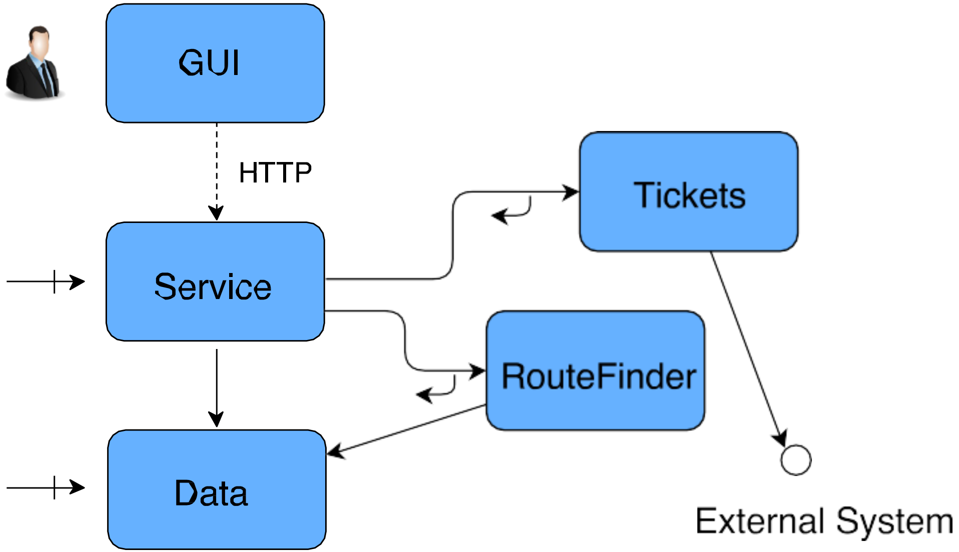
关于质量属性的耦合类型

Type of Coupling in Regard to Quality Attributes

- 耦合对质量属性有影响
- 松耦合通常会提高可维护性、可进化性和可重用性
- 紧密耦合通常会提高可实现的性能(和可跟踪性)
- 有些属性不清楚，例如可测试性
- 因此，稍微偏好松耦合

- Coupling has an **impact on quality attributes**
- Loose coupling will typically improve maintainability, evolvability, reusability
- Tight coupling will typically improve the achievable performance (and traceability)
- For some attributes it is not clear, e.g. testability
- ⇒ Therefore slight preference to loose coupling

Examples of Coupling



Examples of Coupling

- **Tight coupling** between Service and the RouteFinder
- Due to callback (both need to know each other, there might even be a temporal coupling here)
- **Loose coupling** between Tickets and External System
- The external system does not need to know the Ticket component
- The external system might use a standard protocol
- ⇒ Easy to exchange the external system

- 服务和RouteFinder之间的紧密耦合
- 由于回调(两者都需要了解对方，这里甚至可能存在暂时耦合)
- 票证与外部系统之间的松散耦合
- 外部系统不需要知道票据组件
- 外部系统可能使用标准协议
- 易于交换外部系统