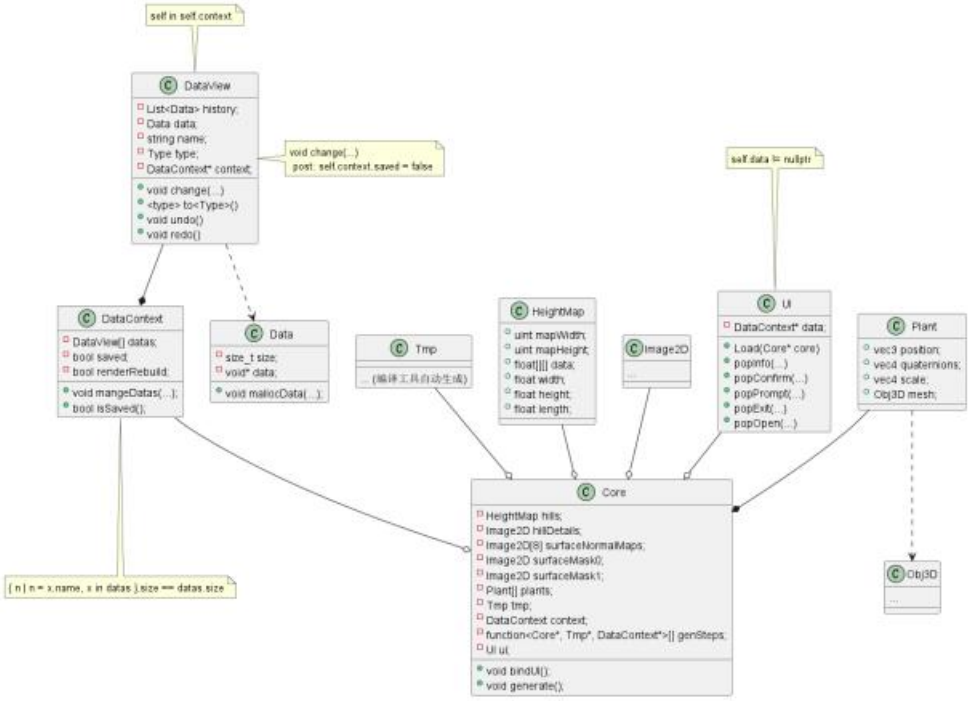
软件体系结构设计说明(SAD)

该软件架构文档模板，参考了国标“13-软件（结构）设计说明（SDD）、SEI ”Views and Beyond” 架构文档模板和 ISO/IEC/IEEE 42010:2011 架构描述模板，并根据实践经验对模板文档结构进行了适当裁剪和调整。

目录



1 引言

1.1 标识

World Generator Lit 0.0.1

1.2 系统概述

系统名称：World Generator Lit 地形生成器
软件用途：自动生成三维地形
系统概述：World Generator Lit 是一个基于 PCG 的过程化开源地形生成器，属于游戏开发过程中的应用 工具。在游戏开发或影视特效制作的过程中，往往有地形生成费时、不能重复使用等问题， 大大增加了游戏或影视制作相关人员的工作量。本项目旨在通过程序自动生成地形，以此 简化相关从业人员在开发制作过程中地形的创建及编辑的工作。 生成器通过使用一系列 PCG 算法，如：柏林噪声算法、二插值、分形噪声算法，实现游戏中 地形及植被的生成，包括但不限于纹理的生成，地形生成之后可以通过插件或脚本导入到 unity3D、unreal engine 等 3 维游戏引擎中，成为高复用、跨平台的面向细节的游戏地形。
开发人：许浩扬、刘亚琦、李怡霏、张雪琛、李嘉昊。
开发历史：3 月 3 日——3 月 20 日　可行性分析
　　3 月 21 日——4 月 20 日　需求分析
　　4 月 20 日——5 月 31 日　详细设计
计划当前运行现场：详细设计及代码编写。

1.3 文档概述

本文档为 World Generator Lit 地形生成器的软件体系结构设计说明（SAD），不具有保密性，可供用户及其他开发者浏览。

1.4 基线

1、完成可视化模块开发
2、完成基础算法库开发
3、完成基础模块开发
4、完成软件主体开发

2 引用文件

可行性研究报告.doc
项目说明.doc
软件需求规格说明(SRS).doc

3 主要设计决策

根据用户需求，软件分为以下几个模块：
A.渲染显示模块：显示数据可视化和 3D 预览，用户通过数据可视化对生成结果的具体数据（三视图、高度图、植被分布、水域分布等）进行可视化预览。3D 预览视图将生成结果通过比较接近 DCC 软件或游戏引擎内渲染效果的形式渲染出来，并且有简易的 3D 导航功能。**关键点：良好的可视化设计、极致逼近最终游戏画面或影视画面的渲染效果。**
B.用户输入模块：PCG 生成模块内的每个生成代理都需要将自己需要的用户输入注册到用户输入模块，用户输入模块会统一将所有代理需要的用户输入按照合适的形式绘制到 UI 界面，并建立由核心管理模块为中心的监听联系，在用户改变输入参数时唤醒代理执行生成过程。**关键点：即时响应用户修改、合理的各种类型的参数输入组件。**
C.PCG 生成模块：管理一系列生成代理，每个生成代理在唤醒后根据用户输入生成结果，并推送到其他模块。如果生成中用户修改了响应代理需要的输入，代理将立即回退进度重新生成。代理需要按需将输入注册到用户输入模块和数据管理模块。
关键点：实时生成、回退代价最小、优化新能。
D.数据管理模块：管理生成代理需要的输入数据，负责数据的保存、读取以及生成结果的导出。关键点：可保存用户输入数据、保证导出结果的正确性。
E.核心管理模块：管理其他模块的运作，在其中承担数据和消息流动的中介。
关键点：保证数据和消息在各个模块间正确流动。

4 体系结构设计

本章应分条描述 CSCI 体系结构设计。如果设计的部分或全部依赖于系统状态或方式，则应指出这种依赖性。如果设计信息在多条中出现，则可只描述一次，而在其他条引用。应给出或引用为理解这些设计所需的设计约定。

4.1 体系结构及视图

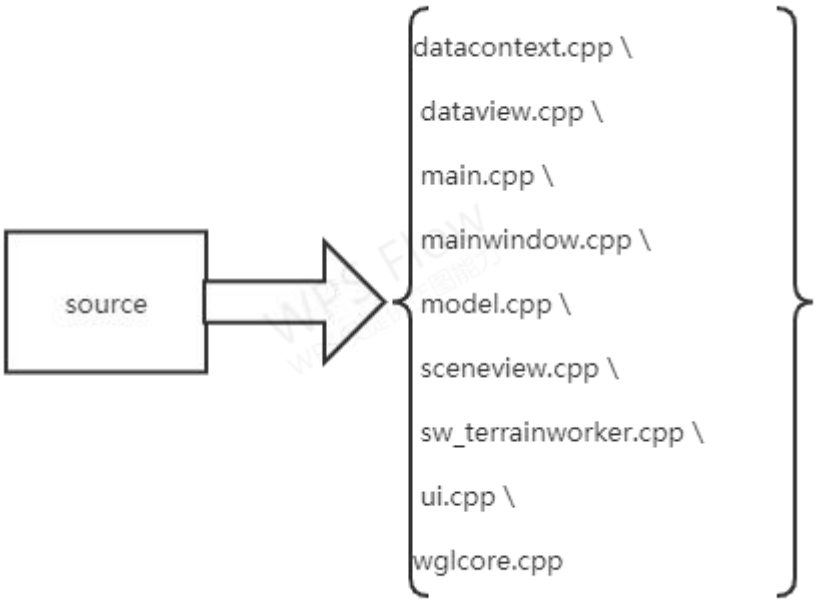
4.1.1 程序(模块)划分

名称	标识符	功能	源文件名	子模块
独立外部工具	ExternalTool	软件主体，拥有	详见子模块	渲染显示模块、

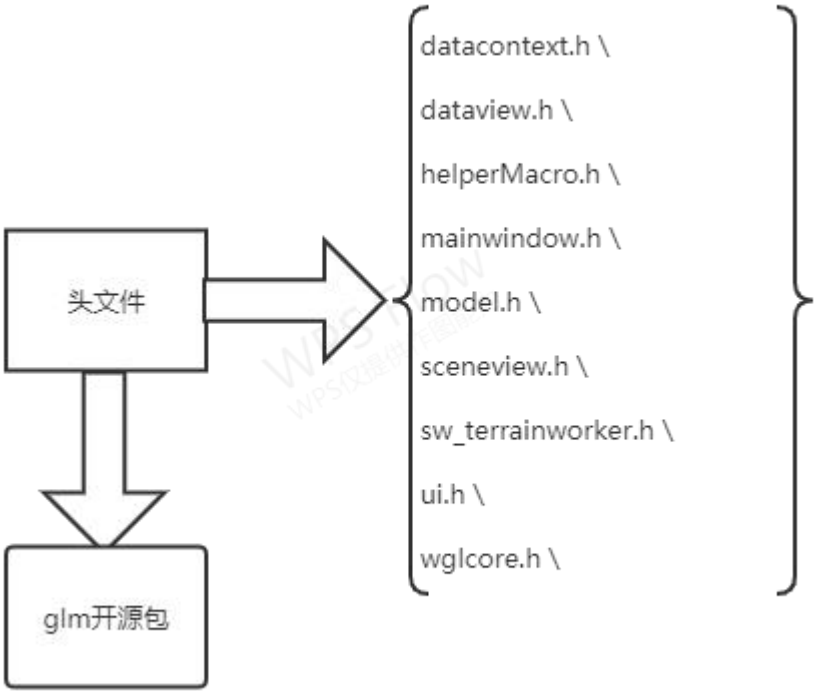
（程序）		完整用户界面的地形 PCG 工具		用户输入模块、PCG 生成模块、数据管理模块、核心管理模块
引擎内对接插件（程序）	Extensions	运行在各种 DCC 软件或游戏引擎内的插件，负责将生成结果平滑导入		
渲染显示模块	SceneView	根据需要可视化数据或渲染生成结果	sceneview.h sceneview.cpp	略
用户输入模块	UI, MainWindow	提供用户输入组件和发送用户修改事件	ui.h ui.cpp mainwindow.h mianwindow.cpp	略
PCG 生成模块（生成代理）	StepWorker 及其子类	负责具体 PCG 生成工作、生成结果推送和配置用户输入接口	wglcore.h sw_xxxx.h sw_xxxx.cpp	略
数据管理模块	DataContext, Job	管理用户输入和生成结果中需要导出的部分，保存用户输入和导出生成结果	datacontext.h datacontext.cpp job.h job.cpp	略
核心管理模块	WGLCore	管理所有 StepWorker，负责生成代理和用户输入之间的消息推送	wglcore.h wglcore.cpp	略

4.1.2 程序(模块)层次结构关系

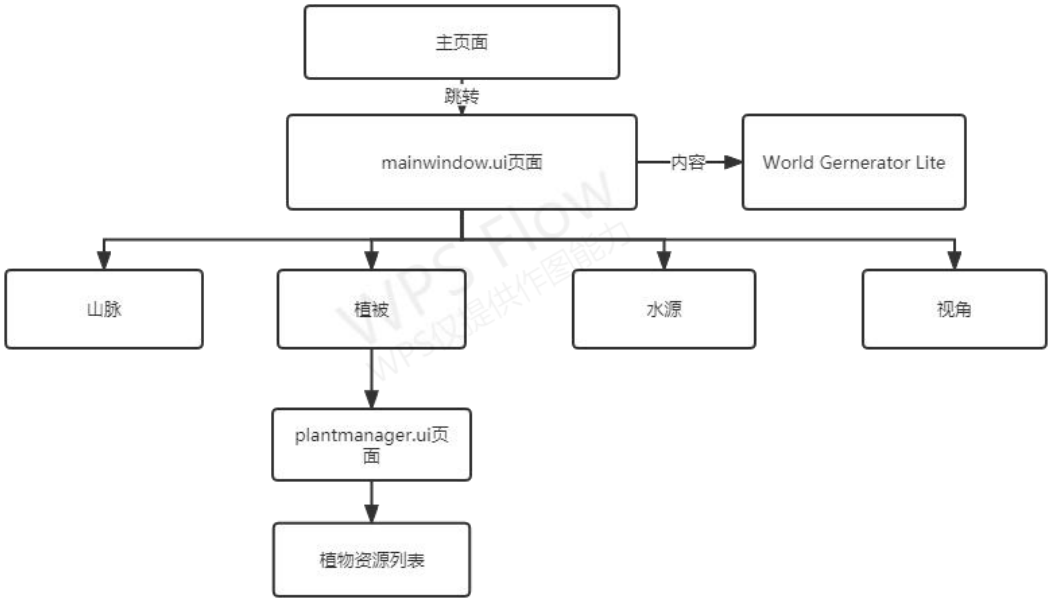
用一系列图表列出本 CSCI 内的每个程序(包括每个模块和子程序)之间的层次结构与调用关系。
模块的子程序包括



程序模块引用的头文件：



我们的视图界面



4.2 全局数据结构说明

4.2.1 常量

dataview.h：

```
#define DATA_VIEW_STRING_LENGTH 1024
#define DATA_VIEW_ITEM_LENGTH 256
#define DATA_VIEW_ITEM_COUNT 64
#define FOR_DATA(type, v) reinterpret_cast<const type*>(v)
```

helperMacro.h：

```
#define DBPRINT(name, format) printf("[debug] " #name " = " #format " [at '%s' -> '%s(...)': line %d]\n", name, __FILE__, __FUNCTION__, __LINE__)
#define DBAREA(...) do { __VA_ARGS__ } while (0)
#else
#define DBPRINT(name, format) do {} while (0)
#define DBAREA(...) do {} while (0)
#endif

#define VK(...) if (__VA_ARGS__ != VK_SUCCESS) { perror("Failed at " #__VA_ARGS__); exit(-1); }
```

model.h：

```
#define IndexSize sizeof(uint32_t)
#define IndexRestart UINT32_MAX
#define IndexType VK_INDEX_TYPE_UINT32
#define VertexSize sizeof(Vertex)
#define InstanceSize sizeof(Transform)
```

sceneview.h：

```
#define MAX_HILL_VSIZE (800 * 800 * 4)
#define MAX_PLANT_VSIZE (4096 * 4)
#define MAX_HILL_I_SIZE (800 * 800 * 5 + 1)
#define MAX_PLANT_I_SIZE (4096 * 5 + 1)
#define VDFT nullptr
#define TODO nullptr
#define PLANT_TYPE_NUM 8
#define AREA_TYPE_NUM 8
#define MAX_IMAGE_SIZE (1024 * 1024)
#define TEXEL_SIZE (sizeof(uint8_t) * 4)
#define MAX_TMP_BUFFER_SIZE ((800 * 800 * 5 + 1) * VertexSize)
#define MAX_TMP_IMAGE_SIZE (1024 * 1024 * sizeof(uint8_t) * 4 * 2)
#define MAX_PLANT_COUNT (1024 * 1024)
```

4.2.2 变量

dataview.h：

```
struct Type {
    enum class Name { Int, Float, Double, Vec2, Vec3, Vec4, String, Char, Enum };
    Name type;
    uint64_t size;

    Type(Name tname);
    Type(const Type& t);
}; // 数据类型
```

model.h：

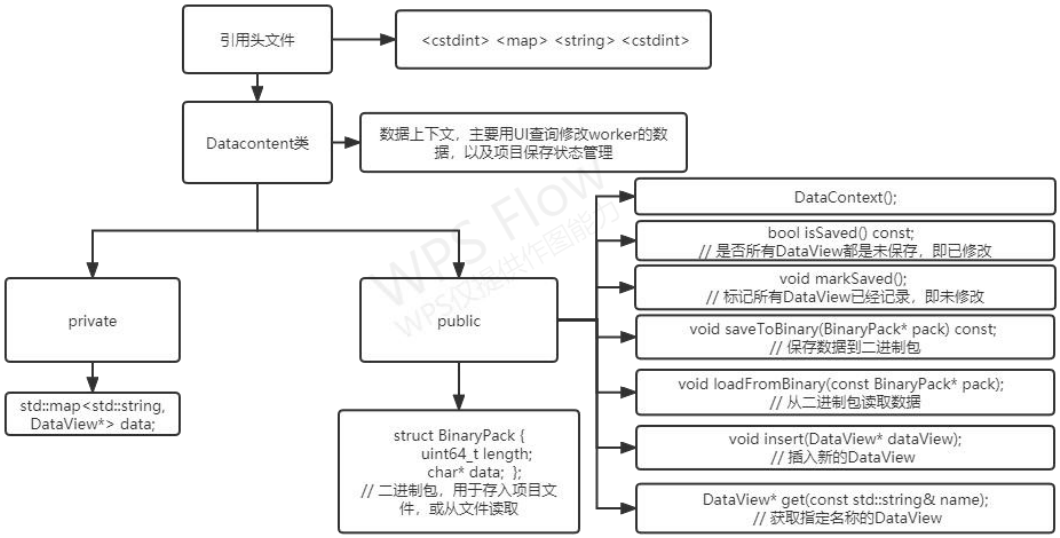
```
struct Vertex {
    glm::vec3 position;
    glm::vec3 normal;
    glm::vec4 tangent;
    glm::vec2 uv = glm::vec2(0.0f, 0.0f);
    glm::vec4 data0 = glm::vec4(0.0f);
    glm::vec4 data1 = glm::vec4(0.0f);
}; // 模型顶点
struct Transform {
    glm::vec3 position;
    glm::vec3 rotation;
    glm::vec3 scale;
}; // 物体变换
```

sceneview.h：

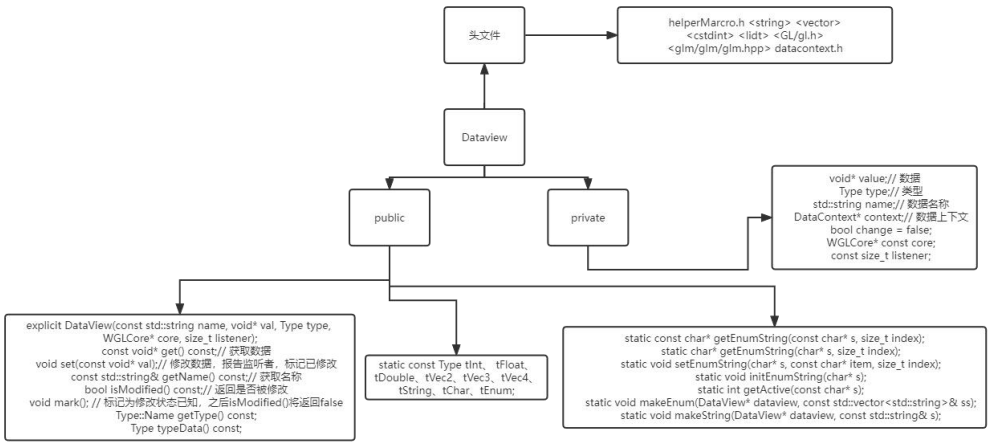
```
struct SimplePBR {
    float roughness;     // 粗糙度
    float specular;      // 高光强度
    float metallic;      // 金属度
    float contrast;      // 反射率对比度
};

struct RenderDescription {
    SimplePBR tallArborMaterial;        // 高大乔木材质
    SimplePBR arborMaterial;            // 乔木材质
    SimplePBR largeShrubMaterial;       // 大型灌木材质
    SimplePBR shrubMaterial;            // 灌木材质
    SimplePBR flowerMaterial;           // 开花植物材质
    SimplePBR grassMaterial;            // 草材质
    SimplePBR creepingMaterial;         // 匍匐植物材质
    SimplePBR stoneMaterial;            // 石头材质
    float fog;                          // 雾浓度
    float fogAttenuation;               // 雾衰减
    glm::vec3 sunColor;                 // 日光颜色
    float sunForce;                     // 日光强度
    float skyForce;                     // 天光强度
};

struct Uniform {
    alignas(4) float time;
    alignas(4) float skyForce;
    alignas(8) glm::vec2 fog;
    alignas(16) glm::vec4 resolution;
    alignas(16) glm::vec4 sun;
    alignas(16) glm::mat4 mvp;
    alignas(16) glm::vec4 bprRSMC;
};
```

4.2.3 数据结构、

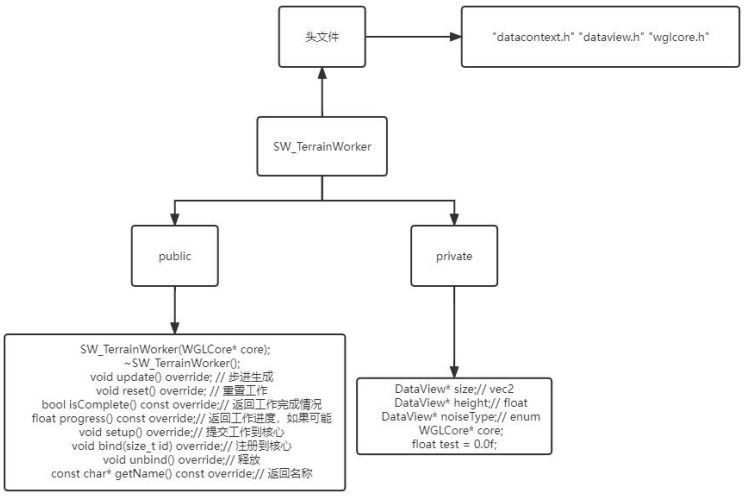
Datacontent 类：



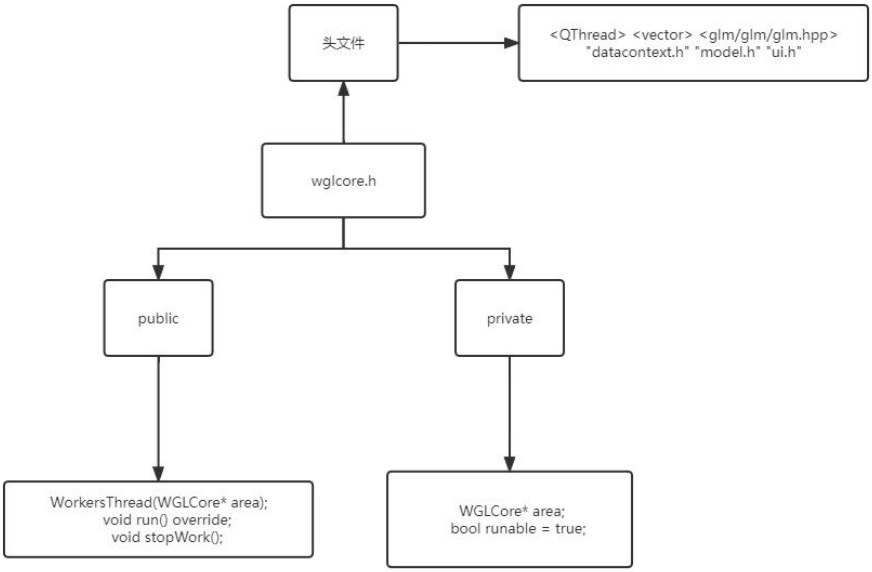
Dataview 类：



SW_TerrainWorker



wglcore.h




```
class Render: public QVulkanWindowRenderer {
    // 基于Vulkan的渲染器

public:
    // 对接窗体系统的重载函数
    Render(QVulkanWindow *w, const bool topView);

    void initResources() override;
    void initSwapChainResources() override;
    void releaseSwapChainResources() override;
    void releaseResources() override;
    void logicalDeviceLost() override;
    void physicalDeviceLost() override;

    void startNextFrame() override;
```

```
public:
    // 面向WGL的接口

    void pushHill(const Vertexs& vs, const Indexs& index);                  // 设置地形数据

    void setPlant(const Vertexs& vs, const Indexs& index, uint32_t id);     // 设置指定id的植被数据

    void setHillTexture(const QImage& image);                               // 设置地形纹理

    void setPlantTexture(const QImage& image, uint32_t id);                 // 设置指定植被的纹理

    void setShaderDescription(const RenderDescription* dsecription);        // 设置渲染相关配置
```

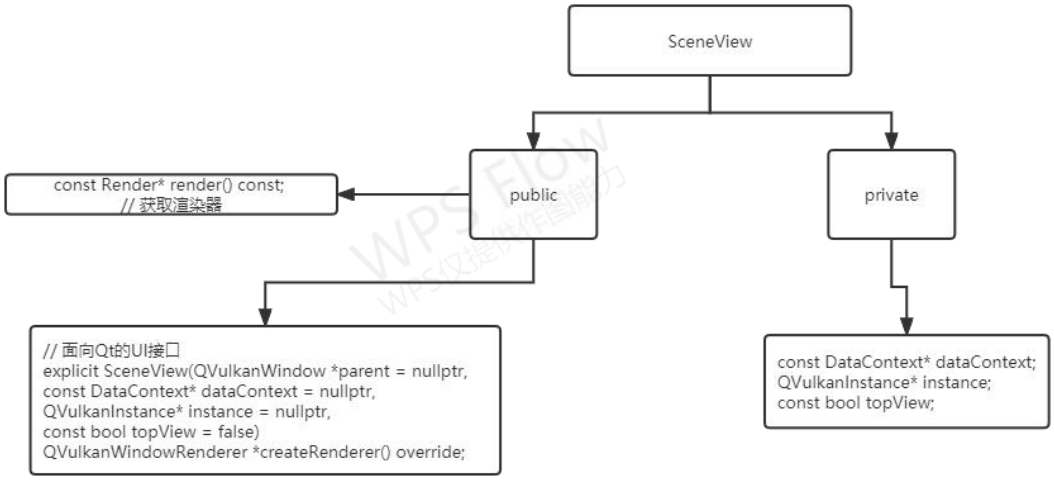
```
private:
    void createBuffer(uint32_t size, void*& data, VkBuffer& buffer, VkBufferUsageFlags usage,
                      VkDeviceMemory& memory, VkMemoryRequirements& memoryRequirement, bool local);
    void createImage(VkExtent2D extent, uint32_t mipsLevel, VkFormat format, void*& data, VkImage& image, VkImageUsageFlags usage,
                     VkDeviceMemory& memory, VkMemoryRequirements& memoryRequirement, bool local);
    void fillBuffer(uint32_t size, const void* data, VkBuffer buffer, uint32_t offset);
    void fillBufferInit();
    void fillImage(uint32_t size, VkExtent2D extent, uint32_t mipsLevel, const void* data, VkImage image, VkOffset2D offset);
    void fillImageInit();
    void buildPipeline(uint32_t width, uint32_t height);
    void destroyPipeline();
    void loadShader();
    void createBasicRenderPass();
    void createSampler();
    void createImageView(VkImage image, VkImageAspectFlags aspect, VkFormat format, uint32_t mipsLevel, VkImageView& view);
    void genMipmaps(VkImage image, VkExtent2D size, uint32_t mipsLevel);
```

```
private:
    const bool topView;
    QVulkanWindow* window;
    QVulkanDeviceFunctions* devFuncs;
    QVulkanFunctions* functions;
    float green = 0;

    VkBuffer hillVertexBuffer;
    uint32_t hillVsCount = 0;
    VkBuffer plantsVertexBuffers[PLANT_TYPE_NUM];
    uint32_t plantsVsCount[PLANT_TYPE_NUM] = { 0 };
    // Vertexs buffers and sizes;

    VkBuffer hillIndexBuffer;
    uint32_t hillIndexCount;
    VkBuffer plantsIndexBuffers[PLANT_TYPE_NUM];
    uint32_t plantIndexCounts[PLANT_TYPE_NUM];
```

```
VkDeviceMemory hillMemory;
VkMemoryRequirements hillMemoryRequirement;
void* hillVData;
VkDeviceMemory plantsMemories[PLANT_TYPE_NUM];
VkMemoryRequirements plantsMemoriesRequirements[PLANT_TYPE_NUM];
void* plantsVData[PLANT_TYPE_NUM];
// Device memories and memory mappings;

VkDeviceMemory hillIndexMemory;
VkMemoryRequirements hillIndexMemoryRequirement;
void* hillIndexData;
VkDeviceMemory plantsIndexMemories[PLANT_TYPE_NUM];
VkMemoryRequirements plantsIndexMemoryRequirements[PLANT_TYPE_NUM];
void* plantsIndexDatas[PLANT_TYPE_NUM];

VkPipeline sceneViewPipeline = VK_NULL_HANDLE;
VkPipelineLayout sceneViewPipelineLayout = VK_NULL_HANDLE;
// Graphics pipelines;


VkRenderPass basicPass;

VkShaderModule standardVertexShader;
VkShaderModule standardFragmentShader;
// Shaders;

Vertexs monkey;
Indexs monkeyIndex;

VkBuffer uniform;
VkDeviceMemory uniformMemory;
VkMemoryRequirements unigormMemoryRequirement;
void* uniformData;
VkDescriptorSetLayout uniformSetLayout;
VkDescriptorPool uniformPool;
VkDescriptorSet* uniformSet = nullptr;
uint32_t uniformSize = 0;
```

```
VkImage hillTextures[AREA_TYPE_NUM];
VkImage hillType0;
VkImage hillType1;
VkImage plantType0;
VkImage plantType1;
VkImage plantsTextures[PLANT_TYPE_NUM];
VkImage testImage;
VkImageView testImageView;
VkSampler standSampler;
VkDeviceMemory hillTMemory[AREA_TYPE_NUM];
VkDeviceMemory hillType0Memory;
VkDeviceMemory hillType1Memory;
VkDeviceMemory plantType0Memory;
VkDeviceMemory plantType1Memory;
VkDeviceMemory plantsTexturesMemory[PLANT_TYPE_NUM];
VkDeviceMemory testIMemory;
VkMemoryRequirements hillTMemoryRequirements[AREA_TYPE_NUM];
VkMemoryRequirements hillType0MemoryRequirement;
VkMemoryRequirements hillType1MemoryRequirement;
VkMemoryRequirements plantType0MemoryRequirement;
VkMemoryRequirements plantType1MemoryRequirement;
VkMemoryRequirements plantsTexturesMemoryRequirements[PLANT_TYPE_NUM];
VkMemoryRequirements testIMemoryRequirement;

    void* hillTDatas[AREA_TYPE_NUM];
    void* hillType0Data;
    void* hillType1Data;
    void* plantType0Data;
    void* plantType1Data;
    void* plantsTexturesDatas[PLANT_TYPE_NUM];
    void* testImageData;

    VkBuffer tmpBuffer;
    VkDeviceMemory tmpBufferMemory;
    VkMemoryRequirements tmpBufferRequirement;
    VkBuffer tmpImage;
    VkDeviceMemory tmpImageMemory;
    VkMemoryRequirements tmpImageRequirement;

    VkBuffer testInstanceBuffer;
    VkDeviceMemory testInstanceMemory;
    VkMemoryRequirements testInstanceMemoryRequirement;
    void* testInstanceData;
    uint32_t testInstanceCount;

};
```

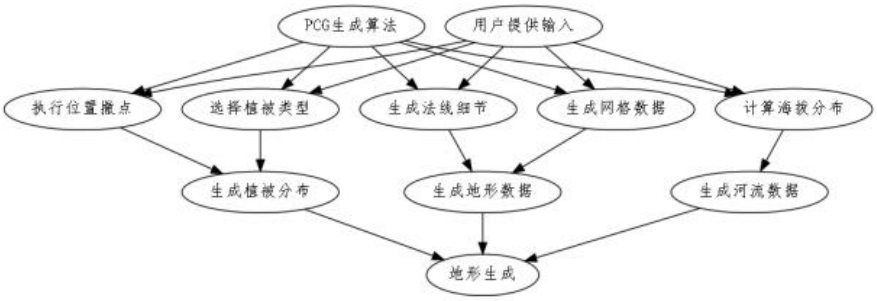
```
class VulkanWindow : public QVulkanWindow {
    // 渲染视图组件

public:
    QVulkanWindowRenderer *createRenderer() override;
};
```



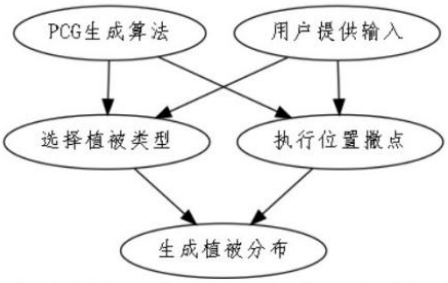
4.3 执行概念

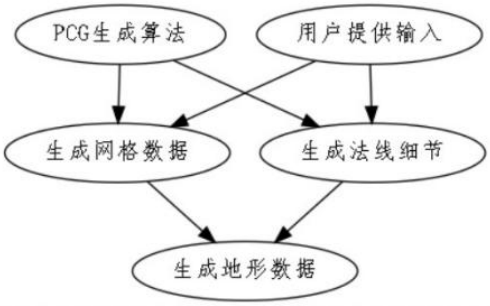
处理流程和数据流程：

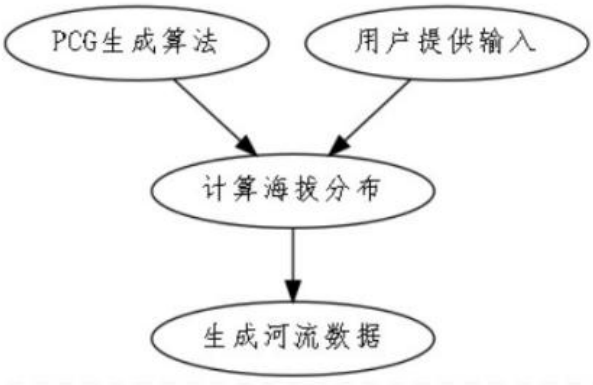
软件系统总体功能：


PCG生成算法
用户提供输入
执行位置撒点
选择植被类型
生成法线细节
生成网格数据
计算海拔分布
生成植被分布
生成地形数据
生成河流数据
地形生成

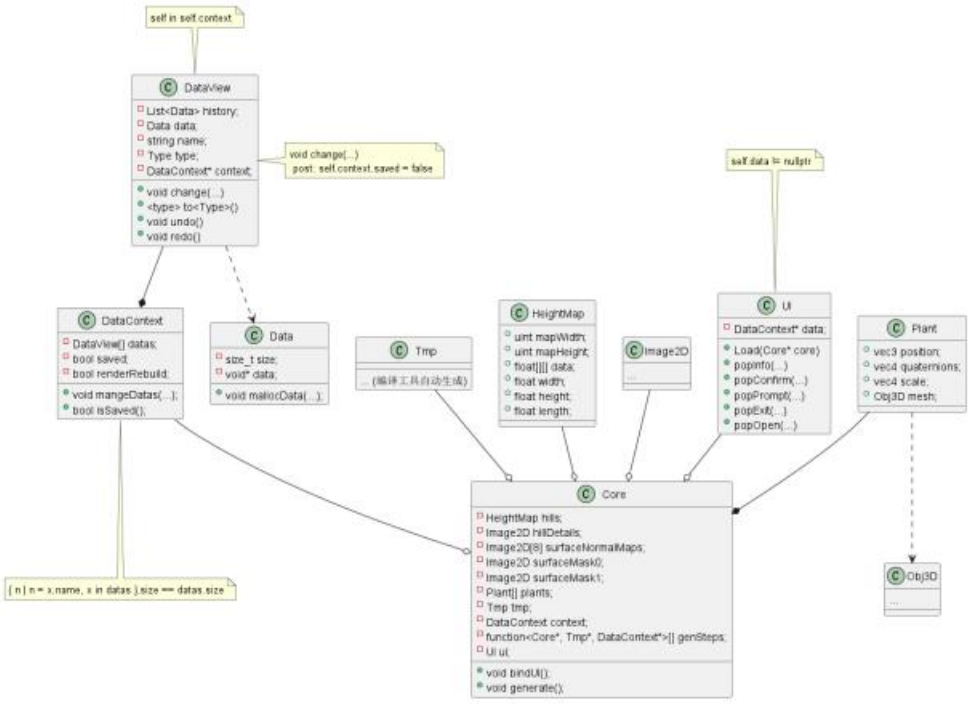
软件子系统功能：


PCG生成算法
用户提供输入
选择植被类型
执行位置撒点
生成植被分布


PCG生成算法
用户提供输入
生成网格数据
生成法线细节
生成地形数据


PCG生成算法
用户提供输入
计算海拔分布
生成河流数据

4.5 接口图



5 需求与架构之间的映射

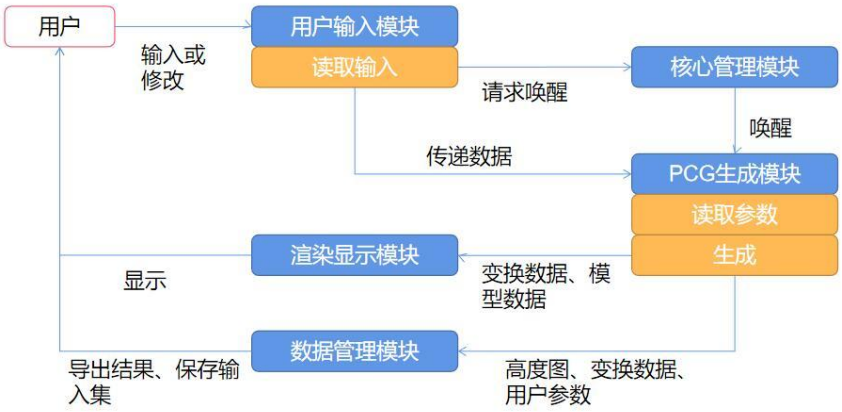
5.1 核心功能

1、地形生成：根据用户的输入生成一系列随机的三维地形，并将结果图形渲染输出。
2、地貌生成：根据参数生成一些列符合地形的地貌，如河流、池塘等，附加在所生成的地形之上。
3、植被生成：根据用户的输入生成一系列植被，与地形图一同输出。

5.2 辅助功能

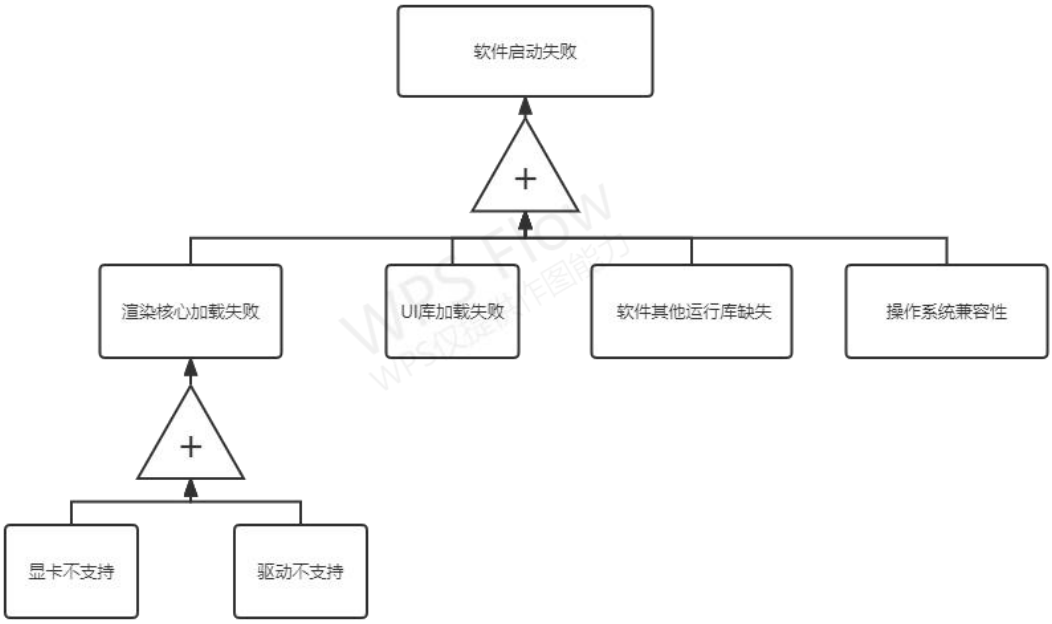
1、渲染显示：显示数据可视化和 3D 预览，用户通过数据可视化对生成结果的具体数据（三视图、高度图、植被分布、水域分布等）进行可视化预览。3D 预览视图将生成结果通过比较接近 DCC 软件或游戏引擎内渲染效果的形式渲染出来，并且有简易的 3D 导航功能。、
2、数据管理：管理生成需要的输入数据，保存输出的数据。
3、模型导出：以需要的格式导出所生成的地形模型。

5.3 软件体系总体功能/对象结构



6 附录

6.1 故障树



故障树变为割集树的过程方法：

1.分配割集树的顶部节点，以匹配故障树顶部的逻辑门。

2、自上而下展开割集树，如下所示：

展开或门节点以具有两个子节点，每个或门子节点对应一个子节点，因为不存在与门所以每个元素均为最小割。

展开与门节点，使子组合节点同时列出与门子节点，该项目不考虑此情况。

通过将合成节点传播到其子节点来展开该节点，但要展开该节点中列出的其中一个门

3、继续，直到所有叶节点都是基本事件或基本事件的合成节点

结论：原故障树对应割集为

6.2 软件体系结构专题学习报告

许浩扬

我主要阅读了 Fault Tree Handbook 故障树手册，所有者为美国核管理委员会（NUREG-0492)

在本书中，所关注的是某些正式的过程和模型。这些模型应该可以按照人类常规思维那样进行分类。人类常用的思维方式有归纳法和演绎法两种。这里有必要对两种方式各自的特点进行讨论。

第一种是归纳法，归纳法是从单独的个案中推导出通用性结论的方法。以一个特定的系统为例，他假设在启动条件上有一个特定的故障，并试图确定该故障或条件对系统运行的影响，就构建了一个归纳的系统分析。因此，书中也探讨某些特定的控制面损失如何影响飞机的飞行，或探讨预算中某些项目的取消如何影响学区的整体运作。也可以询问不插入给定的控制棒如何影响紧急停堆系统的性能，或者给定的初始事件(如管道破裂)如何影响工厂安全。

归纳系统分析的方法很多，该方法的实例有:初步危害分析(PHA)、失效模式与后果分析(FMEA)、失效模式效应与临界性分析(FMECA)、故障危害分析(FHA)和事件树分析。

本书中也再次强调——在归纳方法中，书中假设一些可能的组成条件或初始事件，并试图确定其对整个系统的相应影响。

第二种是演绎法。演绎法是从一般原理推导至具体事件的方法。在演绎法中，他假设一个系统以某种形式失效，并试图找出是系统或者组件的什么行为模式导致了导致了这次失败。按照通俗的讲法，书里把这种方法叫做“夏洛克福尔摩斯”原理。福尔摩斯面对证据，需要重新再现引发犯罪的各类事件。事实上，所有成功的侦探都是演绎法的专家。

生活中常见的演绎法使用场景是事故调查。是什么事件链导致了“不沉之船”泰坦尼克号在首航的沉没？是仪器故障还是人为故障导致了一架商业客机坠毁在山腰上？

这本书的主题——故障树分析，也是演绎系统分析的一个例子。在这种技术中，假定了某些特定的系统状态(通常是故障状态)，并以系统的方式建立了导致这种不希望发生的事件的更基本的故障链。

总之，使用归纳方法来确定哪些系统状态(通常是失败状态)是可能的;演绎方法用于确定给予者系统状态(通常是失败状态)是如何发生的。

刘亚琦：

* Kruchten, P.B., The 4+1 Views Model of Architecture, IEEE Software, Nov 95, pp 42-50.

这篇文章是解决在一个软件架构图上呈现多个关注点的问题。其中 4+1 视图模型是利

用 Phillipe Kruchten 作为 Rational software 股份有限公司软件架构师的经验开发的。五个并发视图是：逻辑视图、过程视图、开发视图、物理视图和场景。每个视图显示所建模系统的特定方面。这使得每组利益相关者能够关注他们所关注的观点。大多数视图使用 Booch 符号。该过程主要由逻辑视图驱动，逻辑视图可以用类图和状态转换图来描述。使用面向对象的分析和设计方法的软件开发人员将发现本文有助于组织他们对系统的看法。

* + Perry, Dewayne E. and Wolf, Alexander L., Foundations for the Study of Software Architecture, alternate source, ACM SIGSOFT Software Engineering Notes, 17:4, October 1992 pp 40-52.

这篇文章旨在为软件架构奠定基础。作者首先通过吸引几个成熟的架构学科来发展对软件架构的直觉。基于这种直觉，作者提出了一个由三个组件组成的软件架构模型：它由三个组件组成：元素、形式和基本原理。元素是处理、数据或连接元素。形式是根据元素的属性和元素之间的关系来定义的——即对元素的约束。该基本原理根据系统约束为体系结构提供了基础，这些约束通常源自系统需求。作者在架构和架构风格的背景下讨论模型的组件，并提供一个扩展示例来说明一些重要的架构和风格注意事项。最后，作者展示了他们的软件架构方法的一些好处，总结了他们的贡献，并将他们的方法与 Schwanke 等人、Zachman 和 Mary Shaw 联系起来。

Dawayne Perry 和 Alexander Wolf 认为软件架构=元素+组成+原理

其中架构元素：具有一定形式的结构化元素，包括处理元素、数据元素和廉洁元素

架构组成：由加权的属性和关系构成。属性用来约束架构元素的选择，关系用来约束架构元素的放置

架构原理：捕获在选择架构风格、架构元素和架构形式的选择动机

而 Mary Shaw 和 David Garlan 则认为软件架构是软件设计过程的层次之一，该层次超越计过程中的算法设计和数据结构设计

其中软件架构包含：组件、连接件、和约束三大要素

组件：可以是一组代码，也可以是独立的程序

连接件：可以是过程调用、管道和消息等，用于表示组件之间的相互关系

约束：一般为组件连接时的条件

李怡霏：

我主要阅读了 Software Arthitecture-a Roadmap 文献。

该文章主要从六个部分讲述了关于软件体系结构的路线图 roadmap。

第一部分为导言。提出了软件的体系结构是在软件系统的设计和构建中的一个关键问题，提出了其重要性。接着说明即使现在取得了很多进展，但是随着工程学科的发展，软件体系结构领域仍然相对不成熟，仍然存在许多挑战和位置，并且技术面貌的变化带来了一些根本性的新机遇。最后提出了本文章主要研究的内容为：研究软件体系结构在研究和实践中的一些主要趋势。

第二部分为软件构架的作用。说明了软件构架至少可以在软件开发的六个方面发挥重要作用：1、理解。2、重用。3、构造。4、进化。5、分析。6、管理。并分别进行了详细的说明。

第三部分为过去（YESTERDAY）。先指出软件构架在过去不被重视，但慢慢人们意识到其重要性，并且在行业内，有两种去实突出了构架的重要性：1、认识到用于构建复杂软件系统的方法、技术、模式和习惯用法的共享剧目。2、关注利用特定领域中的共性，为产品系列提供可重用的框架。

第四部分为现在（TODAY）。数排名了软件构架在当今的软件开发者作为一种重要而

明确的设计活动更为明显。此外，构架设计的技术基础也有了显著改善。三大重要进步为：1、构架描述语言和工具的发展；2、产品线工程和构架标准的出现；3、构架设计专业知识的编纂和传播。并分别详细说明了如今这三个方面的发展变化。

第五部分为未来（TOMORROW）。提出尽管软件构架的基础比十年前坚实的多，但它还没有作为一门在整个软件行业中普遍教授和实践的学科而建立起来。并说明了其发展困难的原因有两个分别为：1、传播新的方法和观念需要实践；2、构架设计的技术基础还不成熟。接着就是从三个方面（更改构建与购买平衡、以网络为中心的计算、普适计算）说明在未来应如何发展软件构架设计。

第六部分为结论。总结全文。

从该文章可以明确：软件体系结构的设计在软件系统的设计和构建中十分重要，应当引起软件开发人员们的重视，应积极完善并发展软件体系结构的设计和构建，使其在未来的软件开发中更加方便开发人员们的使用。

张雪琛：

软件体系结构所考虑的质量属性

在课本中，软件体系结构所考虑的质量属性指的是用户希望在开发者所构造的产品中看到的特点，在设计系统时，开发者希望选择那些能够提高必需的质量属性的体系结构风格，可以通过使用策略以改进设计，达到指定的质量目标。《Software Architecture in Practice 》（3rd Edition）指出系统功能到软件结构上的映射决定了架构对质量的支持，并将质量属性（QA） 定义为系统的可度量或可测试属性，用于指示系统满足其利益干系人需求的程度，也可以将质量属性视为沿着利益相关者感兴趣的某个维度来衡量产品的“好坏”。

课本上重点提到的质量属性有可修改性、性能、安全性、可靠性、健壮性、易使用性及商业目标。可修改性描述了程序能够被正确修改的难易程度。一个可修改的程序应该是可理解的、通用的、简单的、灵活的。可修改性，是便于以后的修改和升级。要实现这个目的，我们就要在设计之初就进行考虑代码的布局与结构，方便自己之后的修改和优化。性能是用来衡量某个产品被特定的用户在特定的场景中，有效、高效并且满意得达成特定目标的程度，涉及到整个系统的技术水平，提高性能至关重要。安全性是软件尤为重要的一个因素，比如说保障用户的个人隐私数据，在数据传输过程中进行加密，比如说防止 SQL 注入。可靠性和健壮性意味着软件应该输出正确的结果或在无法得到结果时输出错误提示。易使用性需要在设计软件体系时考虑用户的心理需求及操作习惯，从而设计出人性化的软件。商业目标则需要考虑软件投入和收益的关系，从而设计出能够带来利益的软件。

《Software Architecture in Practice》中则概括性列出了：功能要求——规定系统必须执行的操作，以及它必须如何表现或对运行时刺激做出反应；质量属性要求——功能要求或整体产品的要求；约束——具有零自由度的设计决策。架构需要对每种质量属性的需求做出响应，通过分配适当的职责顺序来满足功能需求整个设计。

李嘉昊：

我主要阅读 Software Architecture in Practice，3rd Edition，本书主要面向内部的如何设计、评估和记录软件扩展到包括外部影响——更深入地理解对架构的影响，更深入地理解对架构的影响。”体系结构对生命周期、组织和管理都有影响。”

本书彻底修改了本版中涵盖的许多主题。特别是，本书提出的架构设计、分析和文档编制方法是如何实现特定目标的一个版本，但还有其他版本。这导致我们将详细介绍的方法与其基础理论分离。我们现在首先提出了具体的理论方法作为理

论可能实现的说明。本版中的新主题包括以架构为中心的项目管理；架构（architecture）能力；需求建模与分析；敏捷方法；实施和测试；云；还有边缘。