Lecture9 Software Reuse and Component-Based Software Engineer

1. 软件重用 Software Reuse

介绍

- 在大多数工程学科中，**系统 system** 是通过**组合 composition**（用在其他系统中使用过的**组件 component** 构建系统）来设计的
- 软件工程关注于组件的定制开发
- 为了以更低的成本、更快地获得更好的软件质量，软件工程师开始采用**系统重用 systematic reuse** 作为设计过程

软件重用的类型

- **应用系统重用 Application System Reuse**
  - 通过将一个应用程序合并到另一个应用程序中来重用整个应用程序（COTS 重用）
  - 应用程序族的开发（eg: MS Office）
- **组件重用 Component Reuse**
  - 一个应用程序的组件（例如子系统或单个对象）在另一个应用程序中重用
- **功能重用 Function Reuse**
  - 重用一个实现了单一定义良好的函数的软件组件

重用的好处

- 增加可靠性
  - 组件已经在工作系统中使用
- 减少过程风险
  - 开发成本的不确定性较小
- 有效使用专家
  - 重用组件而不是人
- 标准兼容
  - 将标准嵌入到可重用组件中
- 加速开发
  - 避免定制开发和加速开发速度

使用重用设计的需求

- 需要能够找到适当的可重用组合
- 必须确信计划重用的组件是可靠的，并且将按照预期的方式运行
- 必须对要重用的组件进行文档化，以便理解和修改（如果需要的话）

重用的问题

- 增加维护开销
- 缺少工具支持
- 无处不在的“未发明”问题
- 需要创建和维护一个组件库
- 寻找和调整可重用的组件

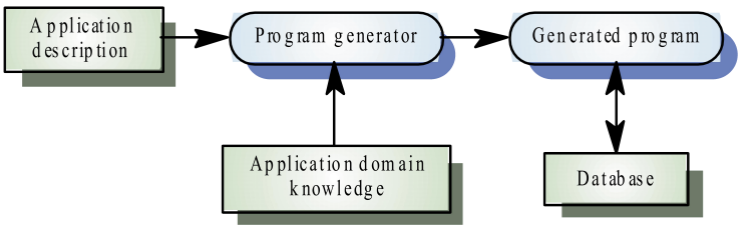
重用的经济学理论

- 质量
  - 随着每一次重用，额外的组件缺陷被识别和删除，从而提高质量
- 生产力
  - 由于花在创建计划、模型、文档、代码和数据上的时间更少，可以用更少的努力交付相同级别的功能，从而提高了生产率
- 开销
  - 通过估算从头开始构建系统的成本，减去与重用相关的成本和交付时软件的实际成本，预计可以节省成本
- 利用结构点进行成本分析
  - 可以根据关于维护、确认、适应和集成每个结构点的成本的历史数据进行计算

基于生成器的重用

- 程序生成器重用标准模式和算法
- 程序根据用户定义的参数自动生成
- 当可以识别领域抽象及其到可执行代码的映射时，就可能实现
- 要领域特定的语言来组合和控制这些抽象

程序生成器的种类



- 用于业务数据处理的应用程序生成器
- 用于语言处理的解析器和词法分析器生成器
- CASE 工具中的代码生成器
- 用户界面设计工具

评估程序生成器重用

- 优点
  - 生成器重用具有成本效益
  - 对于最终用户来说，使用生成器开发程序比使用其他 CBSE 技术更容易
- 缺点
  - 生成器重用的适用性仅限于少数应用领域

2. 基于组件的工程

介绍

- CBSE （Component-Based Software Engineering ）是一种依赖于重用的软件开发方法
- CBSE 出现于面向对象开发的失败中，有效地支持重用
- 对象（类）太具体、太详细，无法支持重用工作的设计
- 组件比类更抽象，可以看作是独立提供的服务

组件的抽象

- 功能抽象
    - 组件实现了一个函数（eg: ln）
- 随意分组
    - 组件是松散相关实体（如声明和函数）的一部分
- 数据抽象
    - 抽象数据类型或对象
- 集群抽象
    - 组件来自合作对象组
- 系统抽象
    - 组件是一个自包含的系统

基于组件的工程系统

- 软件团队引出系统需求
- 架构设计建立
- 团队确定需求是适合组合的而不是构造的
    - 商业现货软件（COTS）组件是否可用来实现需求?
    - 内部开发的可重用组件是否可用来实现需求?
    - 可用组件的接口在提议的系统体系结构中兼容吗?
- 团队试图删除或修改不能用COTS或内部组件实现的需求
- 对于那些可以用可用组件解决的需求，将进行以下活动
    - 组件评估 Component Qualification
        - 候选组件是根据提供的服务和使用者访问的方式来标识的
    - 组件适配 Component Adaptation
        - 修改候选组件以满足体系结构的需要或丢弃它们
    - 组件组合 Component Composition
        - 体系结构根据连接和协调机制的性质决定了最终产品的组成
    - 组件更新 Component Update
        - 由于必须有 COTS 开发人员的参与，更新包含 COTS 的系统变得更加复杂
- 系统剩余部分的详细设计活动开始

可重用组件

- 可重用性和可用性之间的权衡
  - 通用组件可以高度重用
  - 可重用组件可能更复杂，也更难
- 可重用组件的开发成本要高于特定于应用程序的组件
- 与特定于应用程序的模拟组件相比，通用组件的空间利用率较低，执行时间较长

商用现成软件（COTS Commercial Off-the-Shelf Software）

- COTS 系统通常是完整的应用程序库，脱离了一个应用程序编程接口（API）
- 通过集成 COTS 组件构建大型系统对于某些类型的系统（例如电子商务或电子游戏）来说是一种可行的开发策略

COTS 集成问题

- 缺乏开发人员对功能和性的控制
- 由于 COTS 供应商对用户做出了不同的假设，因此组件互操作性存在问题
- COTS 供应商可能不会向用户提供任何对其组件演进的控制
- 供应商可能不会在使用 COTS 组件构建的产品的生命周期内提供支持

领域工程 Domain Engineering

- 领域分析
  - 定义要研究的应用领域
  - 对从域中提取的项进行分类
  - 从域中收集有代表性的应用程序
  - 分析示例中的每个应用程序
  - 开发一个对象分析模型
- 领域结构模型
  - 由少量的结构元素组成，显示出清晰的交互模式
  - 以跨域中的应用程序重用的体系结构风格
  - 结构点是结构模型中的不同结构（例如界面、控制机制、响应机制）
- 可重用组件开发
- 创建了可重用组件的存储库

结构点特征

- 应用程序中具有有限实例数量的抽象，并在域中的应用程序中重复出现
- 管理结构点使用的规则应该容易理解，结构点接口应该简单
- 结构点应该通过隔离结构点本身中包含的所有复杂性来实现信息隐藏

组件的开发

开发组件以供重用

- 组件的构造可以有明确的目标，以允许它们被一般化和重用
- 组件的可重用性应该努力做到
  - 反映稳定的领域抽象
  - 隐藏状态表示
  - 独立（低耦合）

    - 通过组件接口传播异常

基于组件的开发

- 分析
- 架构设计
    - 组件评估 component qualification
    - 组件适配 component adaptation
    - 组件分解 component decomposition
- 组件工程搭建
- 测试
- 迭代进行组件的更新

增加组件适配性的技巧

- 白盒测试
    - 通过对代码进行代码级修改，消除了集成冲突
- 灰盒测试
    - 当组件库提供了允许删除或屏蔽冲突的组件扩展语言或API时使用
- 黑盒测试
    - 需要在组件接口引入预处理和后处理，以删除或屏蔽冲突

增加组件可靠性的技巧

- 名称泛化
    - 名称修改为使用独立于域的语言
- 操作泛化
    - 为提供额外功能而添加的操作
    - 域的具体操作可能会被删除
- 异常泛化
    - 特定于应用程序的异常被删除
    - 增加了异常管理以增强鲁棒性
- 组件认证
    - 可重用的组件保证正确和可靠

组件组成架构元素

- 数据交换模型
    - 应该为所有可重用组件定义类似于拖放类型的机制
    - 允许人到软件和组件到组件的传输
- 自动化
    - 应该实现工具、宏和脚本，以促进可重用组件之间的交互
- 结构化存储
    - 异构数据应该组织并包含在单个数据结构中，而不是几个单独的文件中
- 基本对象模型
    - 确保用不同语言开发的组件在计算平台上是可互操作的

3. 抽象 Abstraction

介绍

- （之前的）抽象 Abstract
  - 与任何特定实例断开关联
  - 难以理解
- 抽象化 Abstraction
  - 忽略不重要的细节，专注于关键功能
  - 这里后面指定抽象都是 Abstraction

关于抽象的事实

- 抽象是强大的
- 人们更注重具体细节而不是抽象
- 人们从例子中学习抽象
- 创造一个好的抽象需要很多例子

如何得到好的抽象

- 从别人那里得到
  - 读大量的书
  - 查看大量代码
- 从例子中归纳
  - 试试
  - 逐步完善它们
- 在你的程序中寻找重复并消除

软件工程中的抽象化

- 过程抽象 Procedural abstraction
  - 命名一个指令序列
  - 参数化的一个过程
- 数据抽象 Data abstraction
  - 命名一个数据集合
  - 一组已处理的数据类型
  - 示例
    - +, - , * , 实数，虚数
    - 队列：add, remove, size
- 控制抽象 Control abstraction
  - W/o指定所有寄存器/二进制级步骤
- 表现抽象 Performance abstraction

高层视图：架构

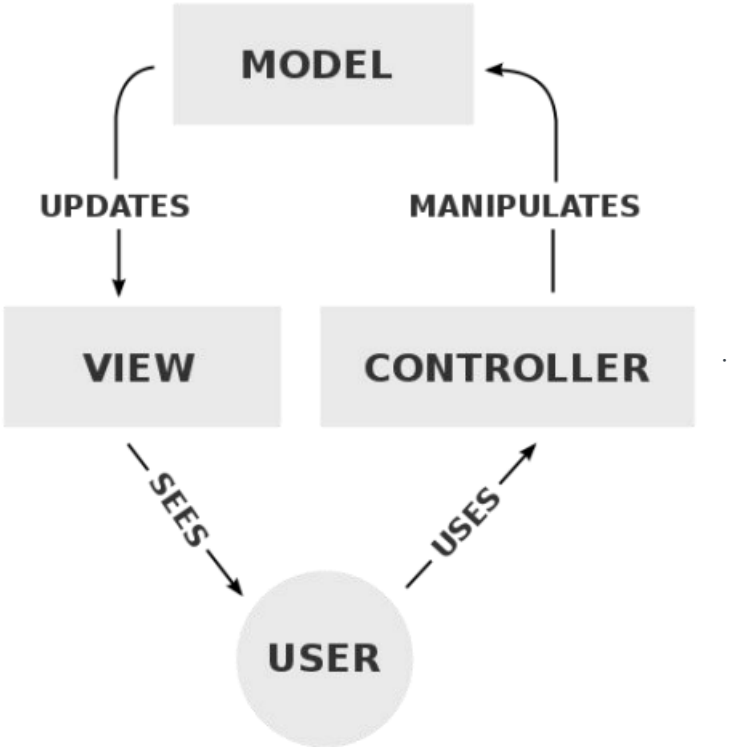
- 大图片
- 支撑系统的结构
- 早期的设计决策
    - 昂贵的改变
    - 满足非功能性需求的关键
- 将系统分解成模块
    - 将开发人员分成（子）团队
    - 将工作分包给其他组
    - 查找要重用的包

选择架构

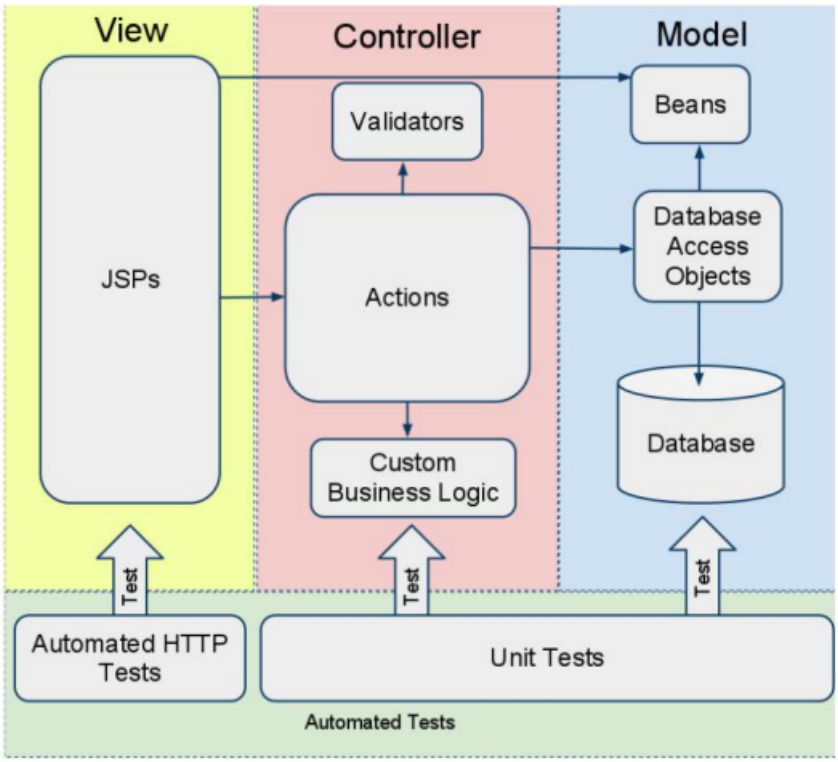
- 许多标准体系结构
    - 每个都有优点和缺点
    - 每个解决了一些问题，也创造了另一些问题
- 如何选择一种架构
- 什么是重要的?
    - 灵活/易于更改，效率，可靠性
- 在Wiki上阅读更多（注意架构模式，包括MVC）

MVC 架构

http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller



示例 iTrust 架构



iTrust View 层

iTrust Model 层

iTrust Controller 层

模块化 Modularity

- 模块化是软件的单一属性，它允许程序在智能程度上可管理
- 将一个较大的程序分割成较小的模块
    - 模块可以是 procedure, class, file, directory, package, service

内聚 Cohesion

每个模块应该做一件事

- 模块**内部**互连的度量
- 模块的一部分对另一部分的依赖程度
- 最大化内聚

内聚的情况

- 巧合的：偶然分组在一起
- 逻辑的：相同的想法
- 时间的：相同的时间
    - 将系统启动或初始化期间使用的所有代码放在一起
- 过程上的：一个调用另一个
- 通讯的：共享数据

耦合 Coupling

每个模块都应该有一个简单的接口

- 模块**之间**互连的度量
- 一个模块对其他模块的依赖程度
- 最小化耦合

信息隐藏 Information Hiding

- 每个模块都应该对其他模块隐藏一个设计决策
- 理论上，每个模块只有一个设计决策，但通常设计决策是紧密相关的

设计决策 Design Decision

- 数据表示
- 特定软件包的使用
- 特定 printer 的使用
- 特定操作系统的使用
- 特定算法的使用

其它模块化的原因

- 让开发人员并行地处理系统
- 安全：划分
- 可靠性：故障本地化
- 并行：负载均衡进程
- 分布式编程：设计模块，减少通信

权衡

- 假设将功能从一个模块移动到另一个模块
    - 减少模块之间的耦合
    - 增加一个模块的内聚
    - 降低另一个模块的内聚
- 是否要做呢?

XP 简化原则

- 运行所有的测试
- 最小化耦合
- 最大化内聚
- 每一个事情"当且仅出现一次"

如何获得模块化

- 重用具有良好模块化的设计
- 考虑一下设计决策——隐藏它们
- 减少耦合并增加内聚
    - 例如，重构(有多少团队进行了重构?)
- 消除重复
- 减少变更的影响
    - 如果添加一个特性需要对系统的大部分进行更改，那么可以进行重构，使更改变得容易

软件框架 Frameworks

- 框架是子系统设计，包含抽象类和具体类的集合，以及每个类之间的接口
- 子系统是通过添加组件来填充缺失的设计元素和实例化抽象类来实现的
- 框架是可重用的实体

框架类

- 系统基础设施框架
    - 支持开发系统基础设施元素，如用户界面或编译器
- 中间件集成框架
    - 支持组件通信和信息交换的标准和类
- 企业应用程序框架
    - 支持特定应用程序的开发，如电信或金融系统

扩展框架

- 需要对通用框架进行扩展，以创建特定的应用程序或子系统
- 框架可以被扩展
    - 定义从抽象类祖先继承操作的具体类
    - 添加将在框架识别的事件响应中调用的方法
- 框架是非常复杂的，学习使用它们需要时间
    - eg: DirectX, MFC