Lecture 11: Software Reuse and Component Based Software Engineering

Resue(复用)

什么是软件复用：软件复用(Software Reuse)是将已有软件的各种有关知识用于建立新的软件，以缩减软件开发和维护的花费。软件复用是提高软件生产力和质量的一种重要技术。早期的软件复用主要是代码级复用，被复用的知识专指程序，后来扩大到包括领域知识、开发经验、设计决定、体系结构、需求、设计、代码和文档等一切有关方面。

Benefits of Reuse(好处)

- Increased Reliability(提高可靠性)：组件已经在工作系统中运行
- Reduced Process Risk(降低流程风险)：减少开发成本的不确定性
- Effective Use of Specialists(有效利用)：复用组件代替人开发
- Standards Compliance(标准符合性)：在可复用组件中嵌入标准
- Accelerated Development (加速开发)：避免定制开发，加速交付

Problem(带来的问题)

- Increased maintenance costs(增加维护成本)
- Lack of tool support(缺少工具支持)
- Pervasiveness of the “not invented here” syndrome(无处不在的"非我发明"综合征，就是代码都必须是我写的，别人的都是垃圾)
- Need to create and maintain a component library(需要创建和维护一个组件库)
- Finding and adapting reusable components(查找和调整复用组件)

Economics of Reuse(复用的经济效益)

- Quality(质量): 随着每次复用，额外的组件缺陷被识别和删除，从而提高了质量。
- Productivity(生产力):由于在创建计划、模型、文档、代码和数据上花费的时间更少，同样级别的功能可以用更少的工作来交付，从而提高了生产力。
- Cost(花费): 通过估算从头构建系统的成本，并减去与复用相关的成本和交付的软件的实际成本，可以预计节省的费用。
- Cost analysis using structure points(使用结构点进行成本分析): 可以根据关于维护、确认、适应和整合每个结构点的成本的历史数据进行计算。

Program Generators(应用生成器)

借助于它们，无须编写代码便可创建一个完整的应用程序。

Type of Program Generators(类型)

- Applications generators for business data processing(业务数据处理)
- Parser and lexical analyzers generators for language processing(用于语言处理的语法分析器和词法分析器)
- Code generators in CASE tools(CASE工具中的代码生成器，CASE是一种软件工具，对某个具体的软件生命周期的任务实现自动化，至少是某一部分的自动化)
- User interface design tools(用户界面设计工具)

Advantages(优点)

- Generator reuse is cost effective(花费有效)
- It is easier for end-users to develop programs using generators than other CBSE techniques(比其他CBSE技术更方便给end-user用户开发)

Disadvantages(缺点)

- The applicability of generator reuse is limited to a small number of application domains(应用仅限于少数的几个领域)

Component-Based Engineering(基于组件的工程)

CBSE(Component-Based Software Engineering)基于组件的软件工程:

- 是强调使用可复用的软件组件来设计和构造基于计算机的系统
- CBSE是一种依赖于复用用的软件开发方法
- CBSE出现于面向对象开发的失败中，未能有效地支持复用
- 对象(类)过于具体和详细，无法支持复用用工作的设计
- 组件比类更抽象，可以视为独立的服务提供者

Component Abstractions(组件的抽象)

- Functional Abstractions(函数抽象): 组件实现单个函数
- Casual Groupings(随意分组): 组件是松散关联实体(如声明和函数)的一部分
- Data Abstractions(数据抽象): 抽象数据类型或者对象
- Cluster Abstractions(集群抽象): 来自协作对象组的组件
- System Abstraction(系统抽象): 组件是一个自包含的系统

Definition of Terms(一些名词解释)

- Component Qualification: candidate components are identified based on services provided and means by which consumers access them(根据提供的服务和使用者访问它们的方式来标识候选组件，类似tag?)
- Component Adaptation: candidate components are modified to meet the needs of the architecture or discarded(候选组件被修改以满足体系结构的需求或被丢弃)
- Component Composition: architecture dictates the composition of the end product from the nature of the connections and coordination mechanisms(体系结构根据连接和协调机制的性质规定了最终产品的组合)
- Component Update: updating systems that include COTS is made more complicated by the fact that a COTS developer must be involved(由于必须有COTS开发人员参与，因此更新包含COTS的系统会变得更加复杂)

COTS(Commercial Off-the-Shelf Software)商业现成的软件: 1. COTS系统通常是应用程序编程接口(API)之外的完整应用程序库。2. 通过集成COTS组件来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)

Developing Components for Reuse(为复用开发组件)

Components may constructed with the explicit goal to allow them to be generalized and reused.(组件的构造可以带有明确的目标，即允许它们被泛化和重用), Component reusability should strive to

- reflect stable domain abstractions(反映稳定的域抽象)
- hide state representations(隐藏状态表示)
- be independent (low coupling)(独立(低耦合))
- propagate exceptions via the component interface(通过组件接口传播异常)

Reusable Components(可复用性组件)

- Tradeoff between reusability and usability(在可复用性和可用性之间权衡)
  - generic components can be highly reusable(通用组件可以高度复用)
  - reusable components are likely to be more complex and harder to use(可复用组件可能更复杂，更难使用)
- Development costs are higher for reusable components than application specific components(可复用组件的开发成本高于特定于应用程序的组件)
- Generic components may be less space-efficient and have longer execution times than their application specific analogs(通用组件可能比它们的应用特定的类似组件空间效率更低，并且有更长的执行时间)

Domain Engineering(领域工程)

领域工程是在构造一个特定领域内的系统或者系统的某些部分时，以可重用方面的形式（也就是说，可重用的工作产物），收集、组织并保存过去的经验的活动，以及在构造新系统时，提供一种充分的方法来重用这些资源（也就是说，获取、限定、改造、装配等等）。

- Domain analysis
  - define application domain to be investigated
  - categorize items extracted from domain
  - collect representative applications from the domain
  - analyze each application from sample
  - develop an analysis model for objects
- Domain model
- Software architecture development
- Structural model
  - consists of small number of structural elements manifesting clear patterns of interaction
  - architectural style that can be reused across applications in the domain
  - structure points are distinct constructs within the structural model (e.g. interface, control mechanism, response mechanism)
- Reusable component development
- Repository of reusable components is created

Component-Based Development(基于组件的开发)

- Analysis
- Architectural design
  - component qualification
  - component adaptation
  - component decomposition
- Component engineering
- Testing
- Iterative component update

Component Adaptation Techniques(组件适配技术)

- White-box Wrapping(白盒包装): integration conflicts removed by making code-level modifications to the code(通过对代码进行代码级修改来消除集成冲突)
- Grey-box Wrapping(灰盒包装): send when component library provides a component extension language or API that allows conflicts to be removed or masked(当组件库提供允许删除或屏蔽冲突的组件扩展语言或API时发送)
- Black-box Wrapping(黑河包装): requires the introduction of pre- and post-processing at the component interface to remove or mask conflicts(需要在组件接口引入预处理和后处理，以移除或屏蔽冲突)

----------以上部分都是Lab10中已经有过的内容----------

Abstration(抽象)

忽略不重要的细节，关注关键特性

Kinds of abstraction in S.E.(各种抽象)

- Procedural abstraction(程序抽象)
  - Naming a sequence of instructions(明明指令序列)
  - Parameterizing a procedure(参数化过程)
- Data abstraction(数据抽象)
  - Naming a collection of data(命名数据集合)
  - Data type defined by a set of procedures(由一组过程定义的数据类型)
- Control abstraction(控制抽象)
  - W/o specifying all register/binary-level steps(不指定所有寄存器/二进制级步骤)
- Performance abstraction O(N)(性能抽象)

Abstract data types(抽象数据类型)

- Complex numbers: +, -, *, real, imaginary
- Queue: add, remove, size

How to get good abstractions(如何获得一个好的抽象)

(说了一堆不重要的方法，，，)

- Get them from someone else
  - Read lots of books
  - Look at lots of code
- Generalize from examples
  - Try them out
  - Gradually improve them
- Look for duplication in your program and eliminate it

Abstractions can fool you

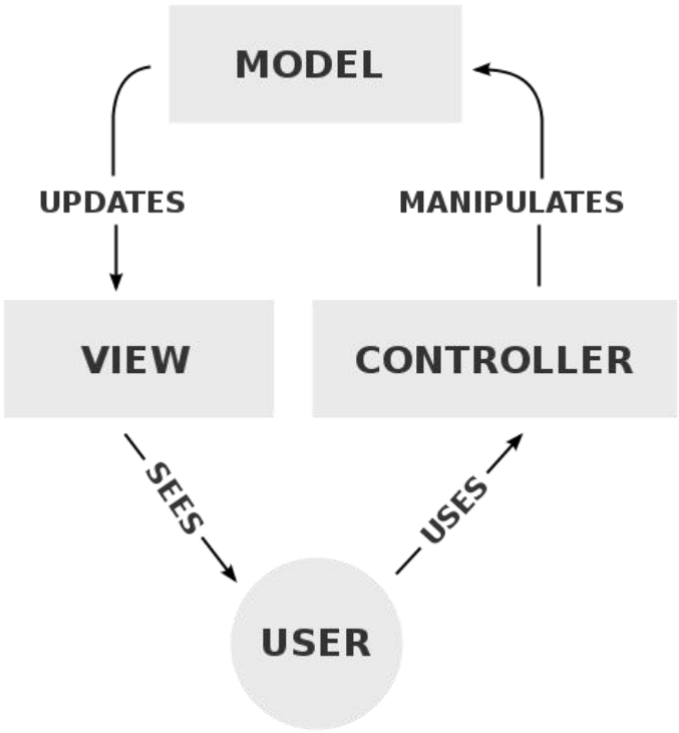
- Suppose collection has operation `getItemNumbered(int index)`
- How do you iterate?
  `for i := 1 to length { getItemNumbered(i) }`
- But what if collection is a linked list?

High-level view: Architecture(高级视图体系: 架构)

- Big picture
- Structure(s) that support the system
- Early design decisions
  - Expensive to change
  - Key to meeting non-functional requirements
- Divide the system into modules
  - Divide the developers into (sub)teams
  - Subcontract work to other groups
  - Find packages to reuse

(其实就是说设计一套架构，将开发人员分组，然后把工作分工下去之类的)

MVC 模式



iTrust View

- Little logic; just display info
- JSPs
- JSP one-to-one mapping to action class (from controller)
- A JSP instantiates the mapped Action class
- Logic related to persistent storage
- DB system (MySQL)
  - each DB entity maps to a single DAO and a single Bean
- Beans: placeholders for data related to an iTrust entity (e.g., Patient)
  - minimal functionality (only store data)
  - Other supporting classes

        - load beans from database result sets
        - validate beans based on input
        - any other custom logic needed.
- DAOs (DB Access Objects): Java objects that interact with the DB
    - reflect contents in the DB
    - offer to interact with the DB
    - query and update the DB (e.g., from Action classes)
    - should not have many branches (assuming incoming data is valid and any exception is handled by the Action classes)
- Handle all logic
    - validate data
    - process DB query results
- Everything between Action classes and DAO classes
    - Action classes
        - exception handling
        - a method <= 15 lines
    - Validators
    - Custom business logic
- Action classes delegate responsibilities to other classes
    - Delegate any input validation to a Validator
    - Log transactions for auditability
    - Delegate any custom business logic, such as risk factor calculations
    - Delegate database interaction
    - Handle exceptions in a secure manner

Modularity(模块性)

Split a larger program into smaller modules(将一个大块的程序分成小的模块)：A module can be procedure, class, file, directory, package, service...

Functional independence(函数无关)

- Coupling(耦合性): 模块间的互连测量，一个模块依赖于其他模块的程度。需要最小化耦合
- Cohesion(内聚性): 模块内部的互连测量，模块的一部分依赖于另一部分的程度。需要最大化聚合

Information hiding(信息隐藏)

每个模块都应该对其他模块隐藏一个设计决策。理想情况下，每个模块一个设计决策，但设计决策通常是密切相关的

Design decisions(设计决策)

- Representation of data
- Use of a particular software package
- Use of a particular printer
- Use of a particular operating system
- Use of a particular algorithm

Other reasons for modularity(模块化的其他原因)

- 让开发人员在系统上并行工作
- 康威(Conway)定律:系统的架构与开发它的团队的架构是相同的。
- Security - compartmentalization(安全-划分)
- Reliability - localization of failure(可靠性-故障本地化)
- Parallelism – load balance processes(并行性-负载平衡进程)
- Distributed programming - design modules to reduce communication(分布式编程-设计模块减少通信)

如何做到模块化

- Reuse a design with good modularity(重用具有良好模块化的设计)
- Think about design decisions – hide them(考虑设计决策-隐藏它们)
- Reduce coupling and increase cohesion(少耦合，增加内聚)
- Eliminate duplication(消除重复)
- Reduce impact of changes(减少变更的影响)
    - If adding a feature requires changing large part of the system, refactor so change is easy(如果添加一个特性需要改变系统的大部分，重构所以改变很容易)

Enhancing Reliability(提高可靠性)

- Name generalization(名字泛化)
    - names modified to use domain independent language(名称修改为使用域独立语言)
- Operation generalization(操作泛化)
    - operations added to provide extra functionality(添加操作以提供额外的功能)
    - domain specific operations may be removed(域特定的操作可能被删除)
- Exception generalization(例外泛化)
    - application specific exceptions removed(应用程序特定的异常被移除)
    - exception management added to increase robustness(添加异常管理以增加健壮性)
- Component certification(组件认证)
    - component warranted correct and reliable for reuse(组件保证正确和可靠的重用)

Application Frameworks(应用框架)

- 框架是子系统设计，包含抽象和具体类以及每个类
- 子系统是通过添加要填充的组件来实现的缺少设计元素和通过实例化抽象类
- 框架是可重用的实体

Framework Classes(框架类)

- System infrastructure frameworks(系统基础架构框架)
    - support development of systems infrastructure elements like user interfaces or compilers(支持系统基础设施元素（如用户界面或编译器）的开发)
- Middleware integration frameworks(中间件集成框架)
    - standards and classes that support component communication and information exchange(支持组件通信和信息交换的标准和类)
- Enterprise application frameworks(企业应用程序框架)
    - support development of particular applications like telecommunications or financial systems(支持电信或金融系统等特定应用的开发)

Extending Frameworks(拓展框架)

- Generic frameworks need to be extended to create specific applications or sub-systems(通用框架需要扩展以创建特定的应用程序或子系统)
- Frameworks can be extend by(框架拓展可以通过)
    - defining concrete classes that inherit operations from abstract class ancestors(定义从抽象类祖先继承操作的具体类)
    - adding methods that will be called in response to events recognized by the framework(添加将在响应由识别的事件时调用的方法框架)
- Frameworks are extremely complex and it takes time to learn to use them (e.g. DirectX or MFC) (框架非常复杂，学习使用它们需要时间（例如DirectX或MFC）)