Overview of Reusable Software Architectures

- Introduction to Software Reuse
- Design Concepts for Software Reuse
- Software Product Lines
  - Modeling families of systems
  - Application domain as a family of systems
  - Object Oriented Analysis and Modeling for Software Product Lines
- Software Architecture
  - Software components
  - Distributed software design
- Software Architecture Patterns
  - Reusable high-level designs
Software Design and Reuse

• Reusable component
  – Active self-contained object with a well-defined interface, capable of being used in different applications from that for which it was originally defined

• Reusable designs
  – Designs that can be reused and adapted

• Design with Reuse
  – Design using reusable components
  – Reusable components previously developed

• Design for Reuse
  – Design of reusable components
  – Design of system containing reusable components

Software Component Reuse

• Traditional Reuse
  – Library of reusable code components
    • May be functional or object-oriented
  – Designer has to design the architecture
  – Developer is responsible for the overall structure of the program and the overall flow of control
  – Developer responsible for selecting reusable components (functional or OO) to fit the architecture
  – Main program
    • Calls procedure of module (functional)
    • Calls operation of object (OO)

• Emphasis on code reuse
Examples of Software Component Reuse

• Subroutine libraries
  – Collection of reusable subroutines in a given application area
  – E.g., statistical subroutine library
• Toolkit
  – A set of related and reusable classes designed to provide useful, general purpose functionality.
  – Toolkits emphasize code reuse.
    • OO equivalent of subroutine libraries.

Architecture Reuse

• Design reuse
  – Reuse components and their interconnections
  – Reuse the control structure of the application
• Architecture reuse has much greater potential than component reuse
  – Large grained reuse
  – Focuses on requirements and design
• Generic architecture
  – One architecture for the application domain
  – Manually adapted (tailored) for a specific application
• Architecture for Software Product Line
  – Captures similarities and variations of product family
Approaches to Architecture Reuse

- Software Product Line modeling
  - A.k.a. Domain modeling, Domain specific software architecture, Software Product Line architecture
  - Architecture for a family of systems
  - Tailored and configured for a given application
    - Member of product family
- Design Patterns
  - Description of communicating objects and classes that are customized to solve a general design problem in a particular context
  - Larger grained reuse than component
- Reference:

Evolutionary Process Model for Software Product Lines
Pattern Categories

- Design Patterns
  - Small group of collaborating objects
  - Gang of Four (Gamma, Helms, Johnson, Vlissides)
- Architecture Patterns
  - Address the structure of major subsystems of a system
  - Buschmann, etc.
- Analysis Patterns
  - Recurring patterns found in Analysis
- Domain Specific Patterns
  - Used in a specific application area (e.g., factory automation)
- Idioms
  - Low-level patterns specific to a programming language

Approaches to Architecture Reuse

- Frameworks
  - Set of abstract classes & the way that instances of these classes interact
  - Classes are specialized, extended and instantiated for a given application
- Component-based Frameworks
  - Provide a capability for components to interface to each other and communication with each other using predefined infrastructure, e.g., CORBA, COM
- Middleware
  - Layer of software sits above heterogeneous operating system
  - Provides uniform platform for distributed applications
Software Design Concepts for Reuse

- Object-oriented concepts
  - Promote software system maintainability, reuse, and evolution
- Information hiding
  - Leads to self-contained components
  - Hence components more modifiable and reusable
- Inheritance
  - Adapting object types by specialization
  - Helps in software reuse and evolution
- Reference:

Object-Oriented Design

- Taxonomy of object-oriented design methods (based on Wegner)
- Object-based design method = Objects
  - Objects are information hiding modules
- Class-based design method = Objects + Classes
  - Classes are object types
- Object-oriented design method = Objects + Classes + Inheritance
  - Inheritance is specialization of a class
  - Superclass: animal, subclasses: cat, dog
Modularity

Structure system into smaller components
Well defined interface between components
Each module
    Designed, coded, tested separately
Module as work assignment
Pre-OO view
    Module corresponds to
        Procedure / function / subroutine
OO view
    Module corresponds to class

Information Hiding

Each object hides design decision
    E.g., data structure
        interface to I/O device
Information hiding object
    Hides (encapsulates) information
        Accessed by operations
Basis for Object-Oriented Design
Advantage
    Objects are more self-contained
    Results in more modifiable -> maintainable system
Example of Information Hiding

- Example of Stack
- Conventional approach
  - Stack data structure is global
  - Stack accessed by modules
  - Module corresponds to procedure / function / subroutine
  - Problem
    - Change to stack data structure has global impact
- Consider
  - Array implementation changed to
  - Linked list implementation
- Every module is impacted by change

Example of Global Access to Data
Example of Global Access to Data

Stack Implemented As Linked List

Example of Information Hiding

- Example of Stack
- Information hiding solution
  - Hide stack data structure and internal linkage
  - Specify operations on stack data structure
  - Access to stack only via operations
- Consider
  - Array implementation changed to
  - Linked list implementation
- Change to stack only impacts Stack object
Figure 2.4 Example of Information Hiding

Figure 2.5 Example of Information Hiding
Classes and Operations

- Operation
  - Is function or procedure that may be applied to objects in a class
  - All objects in class have same operations
- Class has one or more operations
  - Operations manipulate values of attributes maintained by object
- Operations may have
  - Input parameters
  - Output parameters
  - Return value
- Signature of operation
  - Operation’s name
  - Operation’s parameters
  - Operation’s return value
- Interface of class
  - Set of operations provided by class
- Figure 2.6 Example of Information Hiding Class

Inheritance in Design

- Subclass inherits generalized properties from superclass
  - Property is Attribute or Operation
- Inheritance
  - Allows sharing of properties between classes
  - Allows adaptation of parent class (superclass) to form child class (subclass)
- Subclass inherits attributes & operations from superclass
  - May add attributes
  - May add operations
  - May redefine operations
Abstract Class

- Abstract Class
  - Template for creating subclasses
  - Has no instances
  - Only used as superclass
  - Defines common interface for subclasses
- Abstract operation
  - Operation declared in abstract class but not implemented
- Abstract Class defers implementation of some or all of its operations to subclasses
- Different subclasses can define different implementations of same abstract operation

Example of Abstract Class and Inheritance

- Attributes of Account Superclass
  - accountNumber, balance
- Operations of Account Superclass
  - open (accountNumber : Integer)
  - close ()
  - readBalance () : Real
  - credit (amount : Real) {abstract}
  - debit (amount : Real) {abstract}
- Example: Fig. 9.1
Example of Abstract Class and Inheritance

• Attributes of Checking Account Subclass
  – Inherits accountNumber, balance
  – Adds lastDepositAmount

• Operations of Checking Account Subclass
  – Inherits specification and implementation of
    • open, readBalance, close
  – Inherits specification of credit and debit
    • Defines implementation
    • credit sets lastDepositAmount = amount
  – Adds new operation
    • readLastDepositAmount () : Real

Example of Abstract Class and Inheritance

• Attributes of Savings Account Subclass
  – Inherits accountNumber, balance
  – Adds instance attributes cumulativeInterest, debitCount
  – Adds static class attributes maxFreeDebits, bankCharge

• Operations of Savings Account Subclass
  – Inherits specification and implementation of
    • open, readBalance, close
  – Inherits specification of credit and debit
  – Defines debit
    • Debit balance and deduct bank Charge
      if debit Count > max Free Debits
  – Adds Operations
    • addInterest (interestRate) Add daily interest
    • readCumulativeInterest () : Real
    • clearDebitCount () Reinitialize debit Count to zero
Introduction

- Software Product Line
  - Family of products / systems
  - Some common components, some optional, some variant
- Designing Software Product Lines
  - Object Oriented Analysis and Design of Software Product Lines
  - Emphasis on modeling commonality and variability in software product lines
- Unified Modeling Language (UML)
  - Standardized notation for object-oriented development
  - UML notation extended to model software product lines
  - Use UML standard extension mechanisms
    - Stereotypes
    - Constraints
    - Tagged values
- UML 2.0
  - New concepts for depicting software architectures and components
Object-Oriented Analysis and Design (OOAD) for Software Product Lines

- Unified Modeling Language (UML)
  - Standard notation for describing a software design
  - Needs to be used with an analysis and design method
- UML notation for OOAD method for single systems
- UML notation for OOAD Method for modeling software product lines

Evolutionary Process Model for Software Product Lines
How Software Product Line Modeling Method Differs

- Focus on analysis and modeling of software product lines
- Categorization of software product line classes
  - Kernel classes
    - Classes in every member of software product line
  - Optional classes
    - Classes in only some members of product line
  - Variant classes
    - Used differently by individual product line members
- Feature analysis
  - Emphasis on optional software product line requirements
  - Basis for differentiating among product line members
  - Define feature/class dependencies

Software Product Line Scoping

- Needs to be well understood software product line
  - Need software product line expert(s)
- Needs to be relatively stable software product line
- Need to be several systems that are members of the product line
  - Justify additional work for software product line analysis
- Significant common functionality among the members of the product line as well as significant differences
- Start with single system or known product line
  - Identify variant systems
Approach to Software Product Line Scoping

• Analysis needs to emphasize differences between systems
  – Must not abstract away differences between systems
  – “Generic” architecture not enough
• Need to describe each system in common notation (e.g., UML)
• Analysis of legacy systems may require analyzing code
• Essential to extract requirements of existing system
  – Design and coding decisions must not appear in requirements model
• Need to anticipate future changes where possible

Software Product Line Scoping

• Software product line scoping as first cut feasibility study
• Identify boundary of software product line
  – What is inside, outside software product line
• Preliminary feature based analysis of each system
  – Identify common features
  – Identify optional features
• Preliminary assessment of
  – Scope of optional features
  – Size of kernel vs total software product line
  – Degree of variation in software product line
Object-Oriented Analysis and Modeling for Software Product Lines

- Why UML?

- Unified Modeling Language (UML)
  - Standard approach for describing a software design
  - Needs to be used with an analysis and design method
- UML notation used for OO Analysis and Design method for concurrent, real-time and distributed applications
- Method extended to support modeling of software product lines
- OO software product line analysis and modeling for software product lines

Design Notation and Method

- Design notation
  - A means of describing a software design
    - e.g., Unified Modeling Language (UML)
- Design Method
  - Systematic approach for creating a design
    - Design decisions to be made
    - Order in which to make them
  - Designer documents resulting design using design notation
    - UML
Unified Modeling Language (UML)

- UML
  - A standardized notation for object-oriented development
  - Combines notations of OMT, Booch, and use cases
  - A graphical language for describing the products of OO requirements, analysis, and design
  - Approved as a standard by Object Management Group (OMG)
  - Methodology independent
- Needs to be used with an analysis and design method

UML and COMET

- Unified Modeling Language (UML)
  - OMG Standardized notation for describing design
  - Methodology independent
- Concurrent Object Modeling and architectural design mETHod (COMET)
  - Object Oriented Analysis and Design Method
  - Targeted for concurrent, distributed, and real-time applications
  - Uses UML notation
- COMET = UML + Method

Designing Software Product Lines with UML

- Software Product Line (SPL)
  - Family of products / systems (Parnas, Weiss, SEI)
- OO Analysis and Design for Product Lines (PLUS)
  - Extends COMET, other methods for single systems
  - Model commonality and variability among members of product line
- Apply standard UML extension mechanisms
  - Stereotypes, constraints, tagged values
- UML 2.0
  - New notation for depicting software architectures and components

H. Gomaa, “Designing Software Product Lines with UML”, Addison Wesley Object Technology Series, July 2004

Model Driven Architecture

- Promoted by Object Management Group (OMG)
- Model Driven Architecture
  - Develop UML models of software architecture before implementation
- Platform Independent Model (PIM)
  - Precise model of software architecture before commitment to specific platform
- Platform Specific Model (PSM)
  - Map PIM UML model to a specific middleware technology
    - CORBA, .NET, J2EE, Web Services
  - Tool support for mapping from PIM to PSM
UML Modeling for Single Systems

- Use Case Model
  - Use case diagram
- Static Model
  - Class diagram
- Dynamic Model
  - State Machine Model
    - Statechart
  - Interaction Model
    - Communication (collaboration) or sequence diagram

UML Modeling for Software Product Lines

- Use Case Model
  - Model kernel, optional, and alternative use cases
  - Model variation points in use cases
- Static Model
  - Model kernel, optional, variant classes and relationships
- Dynamic State Machine Model
  - Statechart for each state dependent object
- Dynamic Interaction Model
  - Communication diagram for each use case
  - Communication diagrams depend on objects in prerequisite use cases
- Feature modeling
  - Model product line variability in software requirements
What should Design Method for Product Lines Provide?

- Requirements Modeling
  - Support variability in use case modeling
  - Support feature modeling to capture SPL commonality and variability
- Analysis Modeling
  - Support variability in Static Modeling
  - Support variability in Dynamic Modeling
    - Statecharts
    - Interaction diagrams
- Design modeling
  - Support variability in software architecture
Design Method for Software Product Lines

• PLUS
  – Product Lines + UML for Software Engineering
• Extends COMET/UML method for Software Product Lines
• Requirements Modeling
  – Develop Use Case Model
    • Kernel, optional, alternative use cases
    • Use extension points
    • Determine parameters for variables that can change
  – Develop Feature Model
    • Feature is product line requirement
    • Map use cases into features
      – Functional feature as use case or use case package
      – Parametric feature as software product line variable
    • Develop Feature Dependency Model

Design Method for Software Product Lines

• Analysis Modeling
  – Develop static model of problem software product line
    • Determine kernel, optional, variant classes
    • Develop software product line context static model
      – Define scope and boundary of software product line
    • Develop entity class model
  – Develop dynamic models
    • Develop communication / sequence diagrams for each use case
    • Non-state dependent use case
      – Develop communication diagram
    • State dependent use case
      – Develop communication diagram and statechart
Design Method for Software Product Lines

• Analysis Modeling (continued)
  – Develop feature based communication diagrams
    • Communication diagram for each functional feature
  – Develop feature/class dependencies
    • Class diagram for each functional feature
    • Each class supports only one feature
• Design Modeling
  – Develop Software Architecture for product line
    • Component based software architecture
  – Decide on architecture patterns required
  – Design component inter-connection patterns
    • Inter-component communication

Software Application Engineering

• Software Application
  – Member of software product line
• Software Application Engineering
  – Adapt and tailor software product line architecture to derive a given software application
• Select functional features required for application subject to
  – Feature dependencies
  – Feature relationships
    • Mutually exclusive (0 or 1)
    • 1 and only 1 feature from feature set
    • 1 or more from feature set
  – Decide on values of parametric features
• Select optional and variant classes corresponding to features selected
  – Kernel classes always selected