Designing Software Product Lines with UML

Hassan Gomaa
Department of Information and Software Engineering
George Mason University
Fairfax, Virginia 22030-4444

Phone: (703) 993-1652
Email: hgomaa@gmu.edu

Software Engineering Workshop Tutorial
April 2005

DESIGNING SOFTWARE PRODUCT LINES WITH UML

Hassan Gomaa
Department of Information and Software Engineering
George Mason University
Fairfax, Virginia 22030-4444
(703) 993 1652
Fax: (703) 993 1638
Email: hgomaa@gmu.edu

Preface

June 2004

Overview

This book describes an evolutionary software engineering process for the development of software product lines, which uses the Unified Modeling Language (UML) notation. A software product line (or product family) consists of a family of software systems that have some common functionality and some variable functionality. The interest in software product lines emerged from the field of software reuse when developers and managers realized that they could obtain much greater reuse benefits by reusing software architectures instead of reusing individual software components. The field of software product lines is increasingly recognized in industry and government as being of great strategic importance for software development. Studies indicate that if three or more systems with a degree of common functionality are to be developed, then developing a product line is significantly more cost-effective than developing each system from scratch.

The traditional mode of software development is to develop single systems—that is, to develop each system individually. For software product lines, the development approach is broadened to consider a family of software systems. This approach involves analyzing what features (functional requirements) of the software family are common, what features are optional, and what features are alternatives. After the feature analysis, the goal is to design a software architecture for the product line, which has common components (required by all members of the family), optional components (required by only some members of the family), and variant components (different versions of which are required by different members of the family). Instead of starting from square one, the developer creates applications by adapting and configuring the product line architecture.

To model and design families of systems, the analysis and design concepts for single product systems need to be extended to support software product lines. This book is intended to appeal to readers who are familiar with modeling and designing single systems, but who wish to extend their knowledge to modeling and designing software product lines. It is also intended to appeal to readers who are familiar with applying UML to the modeling and design of single systems but not with developing software product lines.

What This Book Provides

Several textbooks on the market describe object-oriented concepts and methods, which are intended for single systems. Very few books address software families or product lines; and of those that do, even fewer use UML.

This book provides a comprehensive treatment of the application of UML-based object-oriented concepts to the analysis and design of software product lines. In particular, it does the following:

- Describes fundamental concepts and technologies in the design of software product lines.
- Describes, in considerable detail, a UML-based object-oriented analysis and design method for software product lines. It examines how each of the UML modeling views—use case modeling, static modeling, dynamic state machine modeling, and dynamic interaction modeling—is extended to address software product lines. Each UML modeling view is extended to reflect the commonality and variability of the product line. A new view, the feature modeling view, is added to explicitly model the commonality and variability of software requirements.
- Uses the Object Management Group (OMG) concept of model-driven architecture to develop a component-based software architecture for a product line. The product line architecture is expressed as a UML platform-independent model, which can then be mapped to a platform-specific model.
- Describes how architectures for software product lines are developed through the consideration of software architectural patterns in relation to the characteristics of the product line. The product line architecture is component-based and explicitly models the commonality and variability of the product line.
- Presents three case studies illustrating how a software product line architecture is developed, starting with use cases and feature modeling in the requirements modeling phase, static and dynamic modeling in the analysis modeling phase, and the development of the component-based software architecture in the design modeling phase. The case studies focus on a microwave oven product line, an electronic commerce product line, and a factory automation product line.
- Includes a glossary, a bibliography, and two appendices, which provide (1) an overview of UML 2 notation and (2) a catalog of software architectural patterns for product lines.

The PLUS Advantage

The UML-based software design method for software product lines described in this book is called PLUS (***P***roduct ***L***ine ***U***ML-Based ***S***oftware Engineering). The PLUS method extends the UML-based modeling methods that are used for single systems to address software product lines. With PLUS, the objective is to explicitly model the commonality and variability in a software product line. PLUS provides a set of concepts and techniques to extend UML-based design methods and processes for single systems to handle software product lines. In particular, for modeling software product lines, PLUS provides the following additions to the process of modeling single systems:

Software Product Line Requirements Modeling

- **Use case modeling**. Model commonality and variability in the use case model. For this purpose, PLUS provides an approach to modeling kernel, optional, and alternative use cases, as well as an approach to modeling variation points in use cases.
- **Feature modeling**. Model product line features. Feature modeling is a key concept in software product lines. PLUS provides an approach for modeling common, optional, and alternative features, an approach for deriving the feature model from the use case model, and an approach for representing features with the UML notation.

Software Product Line Analysis Modeling

- **Static modeling**. Develop a product line context model for the product line boundary. Determine kernel, optional, and alternative external classes. Develop a product line information (entity class) model: determine kernel, optional, and alternative entity classes.
- **Dynamic interaction modeling**. Develop interaction diagrams to realize kernel, optional, and alternative use cases. Use evolutionary development: the kernel first approach is applied to determine product line commonality, followed by product line evolution to determine variability.
- **Dynamic state machine modeling**. Develop kernel, optional, and alternative statecharts. Manage state machine variability through inheritance and parameterization.
- **Feature/class dependency modeling**. Determine the dependency that common, optional, and alternative features have with kernel, optional, and variant classes.

Software Product Line Design Modeling

- **Software architectural patterns**. Determine the software architectural structure and communication patterns that are most appropriate for the product line, given the catalog of architectural patterns.
- **Component-based software design**. Develop a component-based software design for the product line, which models kernel, optional, and variant components, as well as their ports and provided and required interfaces. Design the component-based architecture that explicitly models the components and their interconnections.

Software Application Engineering

Develop a software application that is a member of the product line, by using the feature model to derive the application from the product line architecture and components.

Annotated Table of Contents

Chapter 1: Introduction

This chapter presents an introduction to software product lines, a discussion of software reuse issues, and an overview of object-oriented analysis and design with UML.

Chapter 2: Design Concepts for Software Product Lines

This chapter discusses and presents an overview of key design concepts and technologies for software product lines, including object-oriented technology, software architecture, and the software component technology.

Chapter 3: Software Product Line Engineering

This chapter introduces the software product line design method, which is described in much greater detail in subsequent chapters. One of the goals of this method is to be capable of extending other design methods, such as the author's COMET method (***C***oncurrent ***O***bject ***M***odeling and Architectural Design M***et***hod) to model and design software product lines. The acronym for the method is PLUS (Product Line UML-Based Software Engineering). However, the term *PLUS* is also intended to mean that other methods can be extended to support product lines such as COMET, ROPES, or RUP/USDP.

There are two main strategies for developing a software product line, referred to as *forward evolutionary engineering* and *reverse evolutionary engineering*. Forward evolutionary engineering is best used when a new product line is being developed with no previous systems to guide the development. Reverse evolutionary engineering is best used when the product line development begins with existing systems that are candidates for modernization and inclusion in a project to develop a product line.

Chapter 4: Use Case Modeling for Software Product Lines

This chapter describes how use case modeling concepts are extended to address software product lines—in particular, how the common and variable functionality of the product line is modeled with kernel, optional, and alternative use cases, and how variation points are used to model variability.

Chapter 5: Feature Modeling for Software Product Lines

This chapter describes feature modeling—a concept used widely in software product lines but not addressed by UML. The discussion covers how feature modeling concepts can be incorporated into the UML and how features can be determined from use cases.

Chapter 6: Static Modeling in Software Product Lines

This chapter describes how static modeling concepts are extended to address software product lines—in particular, to address modeling the boundary of the software product line and modeling entity classes, which are information-intensive classes. Also discussed is the categorization of application classes from two perspectives: the role the class plays in the application, and the reuse characteristic of the class.

Chapter 7: Dynamic Interaction Modeling for Software Product Lines

This chapter describes how dynamic interaction modeling concepts are extended to address software product lines. Communication diagrams are developed for each kernel, optional, and alternative use case. Dynamic interaction modeling to address use case variation points is also covered. The kernel first approach is used for dynamic modeling, followed by the product line evolution approach.

Chapter 8: Finite State Machines and Statecharts for Software Product Lines

This chapter describes how finite state machine and statechart modeling concepts are extended to address software product lines. In particular, each state-dependent control class—whether kernel, optional, or variant—needs to be modeled with a finite state machine and depicted as a statechart. It is also possible to model variability using inherited state machines and parameterized state machines.

Chapter 9: Feature/Class Dependency Modeling for Software Product Lines

This chapter describes how to determine which classes from the analysis model are needed to support the features from the feature model. Product line classes are categorized as kernel, optional, and variant classes. Modeling class variability using both inheritance and parameterization is described. Feature-based dynamic modeling and static modeling are also covered.

Chapter 10: Architectural Patterns for Software Product Lines

This chapter describes a range of software architectural patterns that are particularly useful in the design of software product line architectures. Both architectural structure and communication patterns are described. Architectural structure patterns address the overall structure of the software architecture. Architectural communication patterns address the ways in which distributed components can communicate with each other. This chapter also describes how product line architectures can be built from these patterns.

Chapter 11: Software Product Line Architectural Design: Component-Based Design

This chapter describes how a product line architecture is designed as a component-based architecture. Separation of concerns in component design is an important issue. Components are categorized according to their roles in the software architecture. The design of component interfaces is described. The chapter also discusses how component-based software architectures can be depicted with the structured class and composite structure diagram notation introduced in UML 2, which allows components, ports, connectors, and provided and required interfaces to be depicted.

Chapter 12: Software Application Engineering

This chapter describes the process for deriving a member of the software product line from the product line architecture and components. This is a tailoring process involving selection of the

appropriate components and setting of the parameter values for individual components to include in the product line member. The chapter covers how the feature model is used to help in this process.

Chapter 13: Microwave Oven Software Product Line Case Study

This chapter describes how the PLUS software product line method is applied to the design of a microwave oven software product line. Because this is a new product line, the forward evolutionary engineering product line development strategy is used, in which an iterative approach is used to determine the kernel functionality of the product line before the variable functionality is modeled.

Chapter 14: Electronic Commerce Software Product Line Case Study

This chapter describes how the PLUS software product line method is applied to the design of an e-commerce application product line. Because there are two main systems—business-to-business (B2B) and business-to-consumer (B2C)—in the electronic commerce product line, the reverse evolutionary engineering product line development strategy is applied first to each type of system, from which the product line commonality is determined first, followed by the product line variability.

Chapter 15: Factory Automation Software Product Line Case Study

This chapter describes how the PLUS software product line method is applied to the design of a factory automation product line. Because this product line starts with existing factory automation systems that are candidates for modernization and inclusion in the product line, the reverse evolutionary engineering product line development strategy is applied.

Appendix A: Overview of the UML Notation

This appendix provides an overview of the different UML 2.0 diagrams used by the PLUS method. The differences between the UML 2.0 notation and UML 1.x notation are also explained, as are the conventions used in this book.

Appendix B: Catalog of Software Architectural Patterns

In this appendix the software architectural structure and communication patterns, originally described in Chapter 10, are documented alphabetically in a common template for easy reference.

Hassan Gomaa
Biosketch

Hassan Gomaa is Chair and Full Professor in the Department of Information and Software Engineering at George Mason University, Fairfax, Virginia. He received a B.Sc.(Eng.) with First Class Honors in Electrical Engineering from University College, London University, and the DIC and Ph.D. in Computer Science from Imperial College of Science and Technology, London University.

He has over 30 years experience in software engineering, both in industry and academia, and has published over 130 technical papers and 3 textbooks. His book, "Software Design Methods for Concurrent and Real-Time Systems", is published by Addison Wesley as part of the SEI Series on Software Engineering, had its fourth printing in 1999, and was translated into Chinese in 2003. His second book, entitled "Designing Concurrent, Distributed, and Real-Time Applications with UML", was published by Addison Wesley in 2000, had its fourth printing in 2004, and was translated into Chinese in 2004. His latest textbook entitled "Designing Software Product Lines with UML" was published by Addison Wesley in July 2004.

His current research interests include object-oriented analysis and design for concurrent, real-time, and distributed systems, software architecture, software product lines, software reuse, software performance engineering, intelligent software agents, software engineering environments, and software process models. His research has been funded by the National Science Foundation, NASA, DARPA, Hughes Applied Information Systems, Siemens, Software Productivity Consortium, Virginia Center of Innovative Technology, Virginia Center of Excellence in Software Reuse, CTA, American Management Systems, and Grumman. He is the developer of the DARTS, ADARTS, CODARTS, COMET, and PLUS software design methods. He has taught several short courses for industry and academia He also consults in both the technical and management aspects of software engineering.

Previously, he held faculty positions at the Wang Institute of Graduate Studies, where he was Professor of Information Technology, and at Imperial College of Science and Technology, London University, where he was a lecturer (equivalent to Assistant Professor in North America) in the Department of Computing. He also has several years industrial experience, most recently at General Electric.

Designing Software Product Lines with UML

Hassan Gomaa
Department of Information and Software Engineering
George Mason University
Fairfax, Virginia 22030-4444

Phone: (703) 993-1652
Email: hgomaa@gmu.edu

Software Engineering Workshop Tutorial
April 2005

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

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
- *H. Gomaa, "Designing Concurrent, Distributed, and Real-Time Applications with UML", Addison Wesley Object Technology Series, 2000*

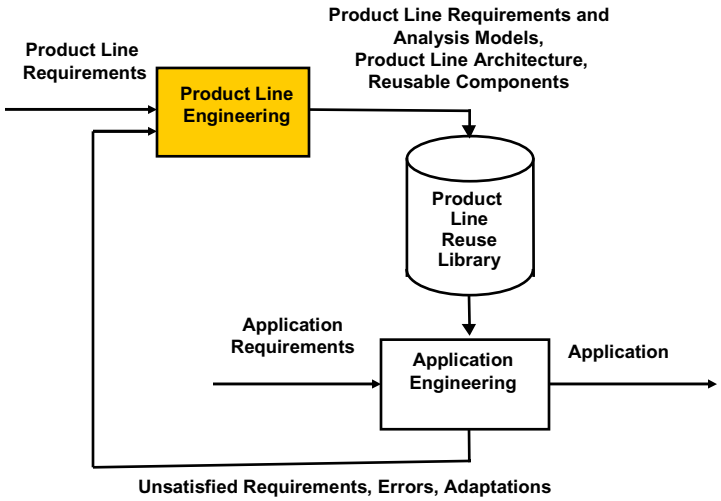
Software Reuse

- Traditional Software Reuse
  - Library of reusable code components
  - Emphasis on code reuse
- Architecture Reuse
  - Focuses on requirements and design reuse
  - Much greater potential than code reuse
- Software Product line engineering
  - Software engineering for family of products
  - Reuse of requirements and architecture
- Application engineering
  - Software engineering for one member of family
- Architecture for Software Product Lines
  - Captures similarities and variations of product family

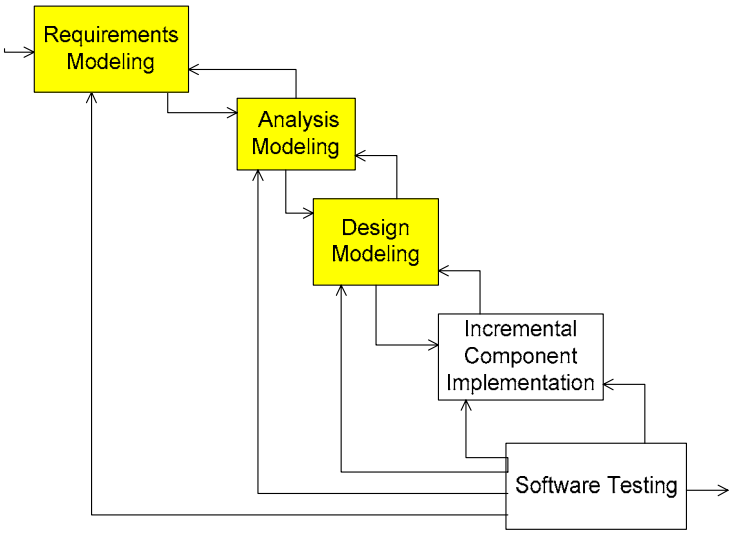
Object-Oriented Analysis and Design (OOAD) for Software Product Lines

- 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
  - H. Gomaa, "Designing Concurrent, Distributed, and Real-Time Applications with UML", Addison Wesley Object Technology Series, 2000.
- UML notation for OOAD Method for modeling software product lines
  - H. Gomaa, "Designing Software Product Lines with UML", Addison Wesley Object Technology Series, July 2004

Evolutionary Process Model for Software Product Lines



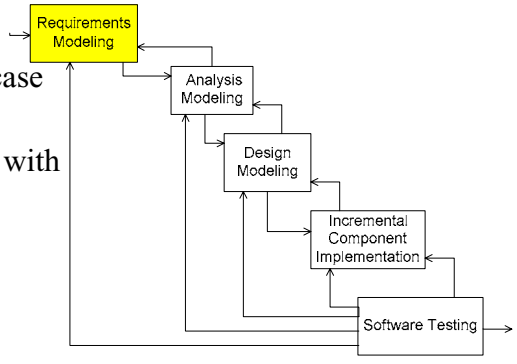
Product Line Engineering



Requirements Modeling

What should SPL Design Method provide?

- Support variability in use case modeling
- Integrate feature modeling with other UML views



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

Use Case Modeling for Software Product Lines

- Use Case Model
  - Defines functional requirements in terms of use cases and actors
  - Use case describes sequence of interactions between actor and system
- Single Systems
  - All use cases required
  - All actors required
- Software product lines
  - Kernel use cases
  - Optional use cases
  - Alternative use cases
  - Some actors may be optional

Variation Points in a Use Case Model

- Variation Point
  - Location in a use case where a change can take place
- Variation Point can be handled by
  - Variation Point within use case
    - Identify line # in use case where variability can be introduced
  - Conditional use case relationship
    - **Extend** relationship
      - Extend use case if *product line condition* is True
    - **Include** relationship
      - Include use case if *product line condition* is True

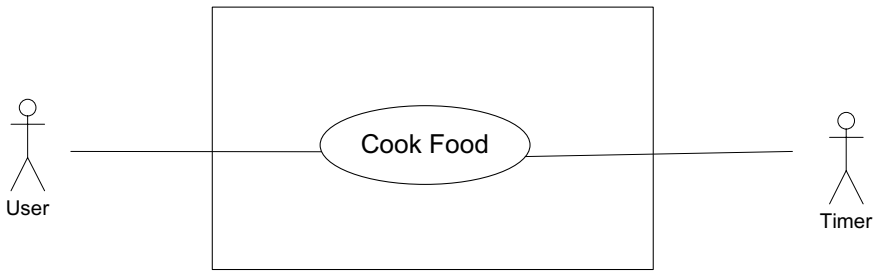
Developing Use Case Model for Software Product Lines

- Kernel First Approach
  - Develop use cases that are common to all members of the software product line
    - Kernel use cases
- Product Line Evolution Approach
  - Start with kernel use case model
  - Develop optional and alternative use cases

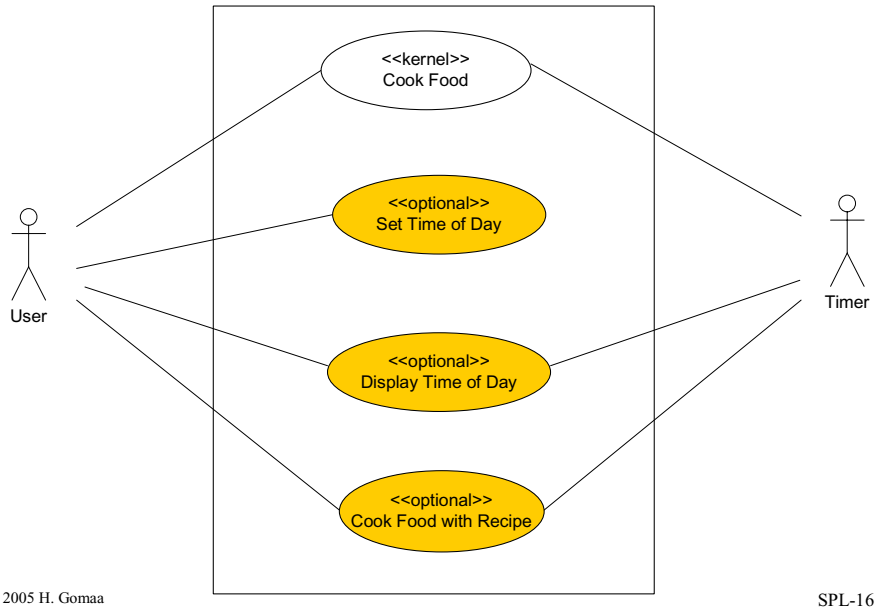
Developing Use Case Model for Software Product Lines - Kernel First Approach

- Kernel First Approach
  - Develop kernel use cases initially
- Example of Kernel First Approach
  - Kernel use case in Microwave Product Line
    - Cook Food
- Product Line Evolution Approach
  - Address Product Line use case variability
  - Cook Food kernel use case
    - Several variation points
  - Optional use cases in Microwave Product Line
    - Set Time of Day
    - Display Time of Day
    - Cook Food with Recipe

Use Case Model for Microwave Oven System



Use Case Model for Microwave Oven Product Line



Use Case Model for Microwave Oven SPL

Use case name: Cook Food
Reuse category: Kernel
Summary: User puts food in oven, and microwave oven cooks food.
Actors: User (primary), Timer (secondary).
Precondition: Microwave oven is idle.

Description:
1. User opens the door, puts food in the oven, and closes the door.
2. User presses the **Cooking Time** button.
3. System prompts for cooking time.
4. User enters the cooking time on the numeric keypad and presses the **Start** Button.
5. System starts cooking the food.
6. System continually displays the cooking time remaining.
7. The timer elapses and notifies the system.
8. System stops cooking the food and displays the end message.
9. User opens the door, removes food from the oven, and closes the door.
10. System clears the display.

Alternatives:
Line 1: Uses presses Start when the door is open. System does not start cooking.
…..

Postcondition: Microwave oven has cooked the food.

Cook Food Use Case - Variation Points

- Lines 3,8: Display Language – Mandatory alternative
  - Default = English
  - Alternatives = French, Spanish, German, Italian
- Line 1: Weight Sensor – Mandatory alternative
  - Default = Boolean weight
  - Alternative = Analog weight
- Line 1: Heating Element – Mandatory alternative
  - Default = One-level Heating Element
  - Alternative = Multi-level Heating Element
- Line 2: Power level – Optional
  - Power level buttons = high, medium, or low.

Cook Food Use Case - Variation Points

- Lines 3,4,6,8,10: Display Unit - Mandatory alternative
  - Default = one-line display
  - Alternative = multi-line display
- Lines 2, 6: Minute plus. Optional
  - Add one to cooking time.
  - Start cooking if not yet cooking.
- Lines 1,5,8,9: Light – Optional
  - Lamp switched on when cooking & when door is open.
- Lines 5,8: Turntable – Optional
  - Turntable rotates for duration of cooking
- Line 8: Beeper – Optional
  - Switch Beeper on when cooking stops

Use Case Model for Microwave Oven SPL

Use case name: Set Time of Day
Reuse category: Optional
Dependency: Variation point in Cook Food use case: at Display Unit variation point, select Multi-Line Display
Summary: User sets Time of Day clock
Actor: User
Precondition: Microwave oven is idle
Description:

1. User presses **Time of Day** (TOD) Button
2. System prompts for Time of Day.
3. User enters the Time of Day by pressing the numeric keypad.
4. System displays the entered Time of Day.
5. User presses start.
6. System starts the Time of Day timer.
7. System increments the Time of Day each second.

Alternatives:

Variation Points
Line 4: 12/24 hour Clock – Mandatory alternative
TOD display is either 12 hour clock or 24 hour clock

Postcondition: TOD clock has been set

Use Case Model for Software Product Lines - View Integration Approach

- Specify use cases for each member of product line
- Compare use cases for different members
- Determine kernel, optional, and alternative use cases
  - Kernel use cases needed by all Product Line members
  - Optional use cases needed by some Product Line members
  - Alternative use cases are mutually exclusive

Figure 4.13 E-commerce: Example of Reverse Engineering Approach: Supplier use cases in B2C systems

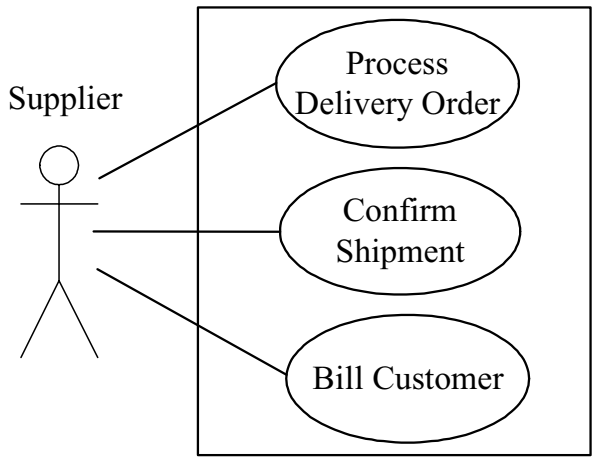


Figure 4.14 E-commerce: Example of Reverse Engineering Approach: Supplier use cases in B2B systems

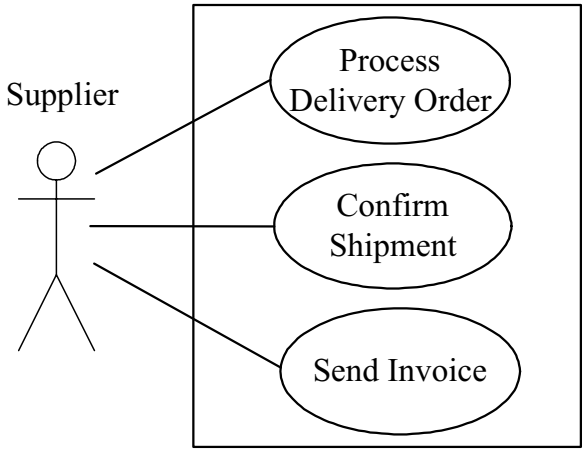
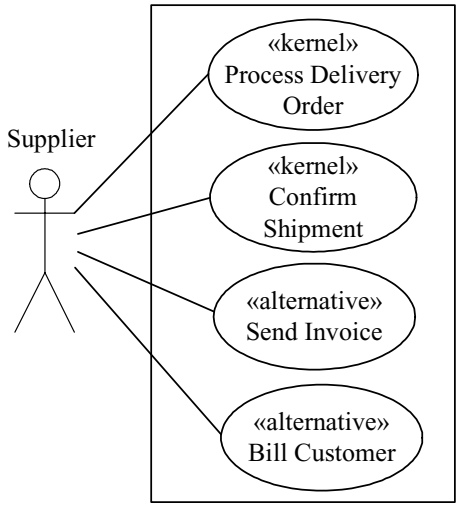


Figure 4.15 E-commerce: Example of Reverse Engineering Approach: Supplier use cases in E-Commerce product line



Feature Modeling for Software Product Lines

- Feature (Kang, SEI)
  - Function or characteristic that differentiates between members of the software product line
- Feature modeling
  - Very important for SPLs
  - PLUS integrates feature modeling with other UML modeling views

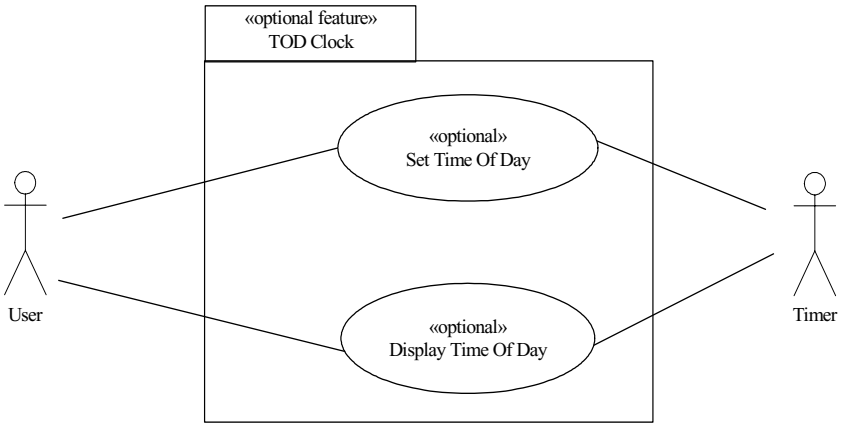
Feature Modeling

- Feature
  - Function or characteristic that differentiates between members of the software product line
- Features are categorized as
  - Common features
    - Required by all members of product line
  - Optional features
    - Required by some members of product line
  - Alternative features
    - Choice of features
    - One of the alternatives may be a default feature
  - Parameterized feature
    - Type, permitted values, default value

Feature Modeling in UML

- Model feature as a use case
  - Can use when a feature is modeled as a use case
- Model feature as a use case package
  - Can use when a feature is a grouping of use cases
- Model feature as a class
  - Using UML static modeling to model meta-classes
- Feature / use case dependency
  - Tabular representation

Figure 5.1 Optional Feature as Use Case Package



Feature Modeling

- Feature Dependencies
  - One feature depends on another
  - Dependency on common features is implicit
  - Dependency on optional features is explicitly specified
- Feature Relationships
  - Mutually exclusive features
    - Zero or One out of a group of features
  - Exactly one of a group of features
    - One and only one out of a group of features
  - One or more of a group of features
    - One or more out of a group of features
  - Mutually inclusive
    - If one feature is picked, the other must be picked

Feature Modeling with UML

- Derive features from use cases and variation points
  - Concentrate on modeling variability
- Use static modeling meta-class notation
  - Classes depict *features* and *feature groups*
- Features are categorized using UML stereotypes
  - <<common feature>>
  - <<optional feature>>
  - <<alternative feature>>
  - <<default feature>>
  - <<parameterized feature>>
- Model Feature Dependencies and Feature Relationships

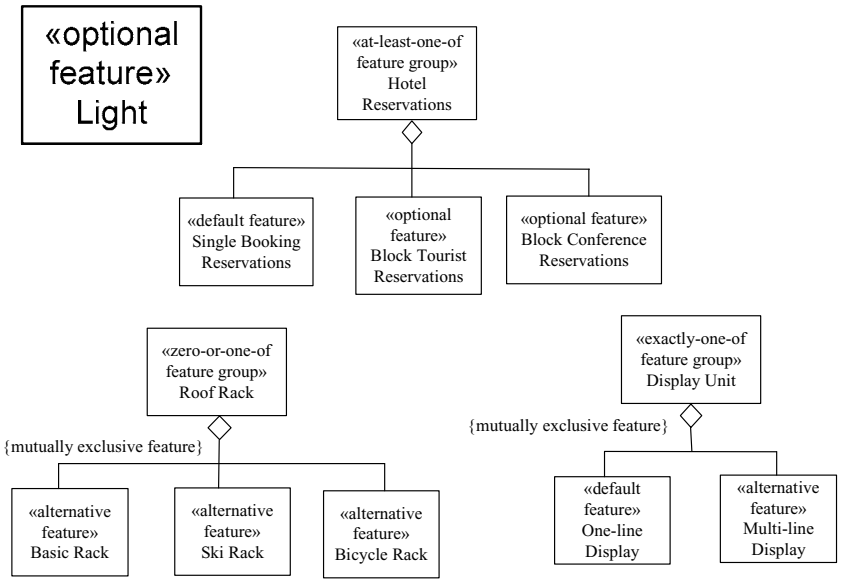
Feature Notation

- Uses extension mechanisms of UML
  - Stereotypes, tagged values, constraints
- «common feature» Feature Name,
  - «common feature» Factory Kernel
- « optional feature» Feature Name {prerequisite = P}
  - «optional feature» Light, «optional feature» Beeper
- «alternative feature» Feature Name {prerequisite = P}
  - «alternative feature» French, «alternative feature» Spanish
- «default feature» Feature Name {prerequisite = P}
  - «default feature» English
- « parameterized feature» Feature Name
  - «parameterized feature» ATM Password Length {type = integer, permitted value = 4..8, default value = 4}

Feature Notation

- «zero-or-one-of feature group» Feature Group Name {Alternative = A1…An, Prerequisite = P}
  - «zero-or-one-of feature group» Roof Rack {alternative = Basic Rack, Ski Rack, Bicycle Rack}
- «exactly-one-of feature group» Feature Group Name {default = D, alternative = A1…An, prerequisite = P}
  - «exactly-one-of feature group» Display Unit {default = One-line Display, alternative = Multi-line Display}
- «at-least-one-of feature group» Feature Group Name {default = D, feature = O1, …, On, prerequisite = P}
  - «at-least-one-of feature group» Hotel Reservations {default = Single Booking Reservations, feature = Block Tourist Reservations, Block Conference Reservations}

Figure 5.8 Features and Feature Groups in UML



Use Cases and Features in Software Product Lines

- Use Cases
  - Define functional requirements of system or SPL
- Features
  - Identify reusable requirements
- Use cases
  - Basis to determine features in Software Product Lines
- Functional feature modeled as
  - One or more use cases reused together
    - Feature modeled as use case or use case package
- Use Case Variation Point as feature
  - Optional functional requirement within a use case
  - Alternative functional requirement within a use case
- Feature / use case dependency
  - Can use tabular representation

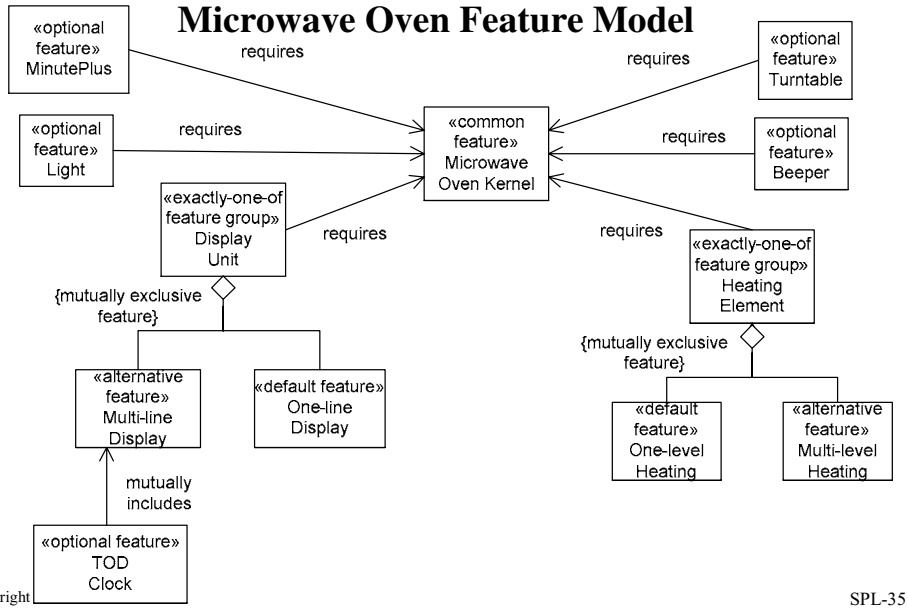
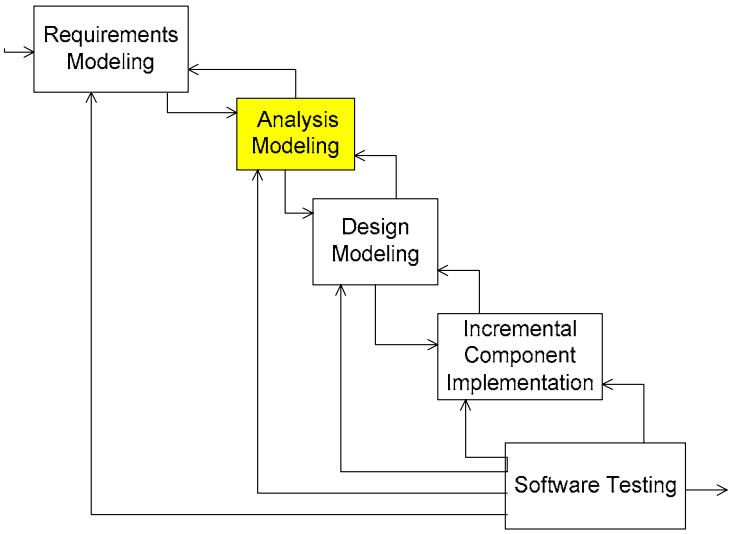


Table 5.1 Feature/use case relationships: microwave oven SPL

Feature Name	Feature Category	Use Case Name	Use Case Category/ Variation Point (vp)	Variation Point Name
Microwave Oven Kernel	common	Cook Food	kernel	
Light	optional	Cook Food	vp	Light
Turntable	optional	Cook Food	vp	Turntable
Beeper	optional	Cook Food	vp	Beeper
Minute Plus	optional	Cook Food	vp	Minute Plus
One-line Display	default	Cook Food	vp	Display Unit
Multi-line Display	alternative	Cook Food	vp	Display Unit
English	default	Cook Food	vp	Display Language
French	alternative	Cook Food	vp	Display Language
Spanish	alternative	Cook Food	vp	Display Language
German	alternative	Cook Food	vp	Display Language
Italian	alternative	Cook Food	vp	Display Language
Boolean Weight	default	Cook Food	vp	Weight Sensor
Analog Weight	alternative	Cook Food	vp	Weight Sensor
One-level Heating	default	Cook Food	vp	Heating Element
Multi-level Heating	alternative	Cook Food	vp	Heating Element
Power Level	optional	Cook Food	vp	Power Level
TOD Clock	optional	Set Time of Day	optional	
		Display Time of Day	optional	
12/24 Hour Clock	parameterized	Set Time of Day	vp	12/24 Hour Clock
		Display Time of Day		
Recipe	optional	Cook Food with Recipe	optional	

Product Line Engineering
- Analysis Modeling



Analysis Modeling
What should SPL Design Method provide?

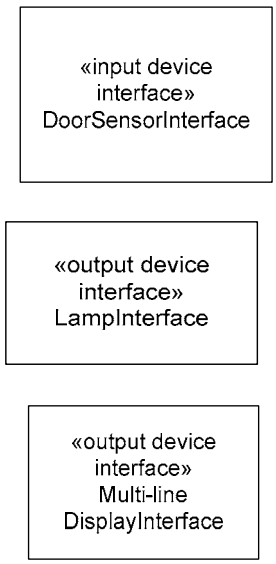
- For product lines
  - Support variability
    - Static Modeling
    - Dynamic Modeling

Static Modeling for Software Product Lines

- Static Model
  - Define structural relationships between classes
  - Depict classes and their relationships on class diagrams
- Static Modeling for software product lines
- Depict class categorization using UML stereotypes
  - **Stereotype** defines a new building block that is derived from an existing UML modeling element but is tailored to the modeler's problem
  - Depicted using guillemets
    - «entity», «interface», «control»
- UML 1.4 upwards supports multiple stereotypes for class
  - Use UML stereotypes to depict reuse category
  - Use UML stereotypes to depict application role category

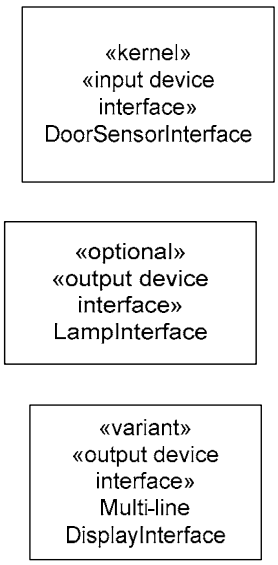
Static Modeling for Single Systems

- UML 1.4 upwards supports multiple stereotypes for a modeling element
- Single systems (COMET)
  - Categorize each class by application role using stereotype
  - «control», «entity», «interface»



Static Modeling for Software Product Lines

- UML 1.4 upwards supports multiple stereotypes for a modeling element
- Single systems
  - Categorize each class by application role using stereotype
  - «control», «entity», «interface»
- **Software Product Lines (PLUS)**
  - **Second UML stereotype depicts reuse category**
  - **«kernel», «optional», «variant»**



Software Product Line Context Model

- Defines boundary between software product line and external environment
  - Depicted on UML product line context class diagram
- Software product line system
  - Consider as one aggregate class
- Application role categories of external classes
  - «external I/O device», «external user», «external system», «external timer»
- Reuse categories of external classes
- Develop using:
  - Kernel First Approach or
  - View Integration Approach

Microwave Oven product line
Kernel System context class diagram

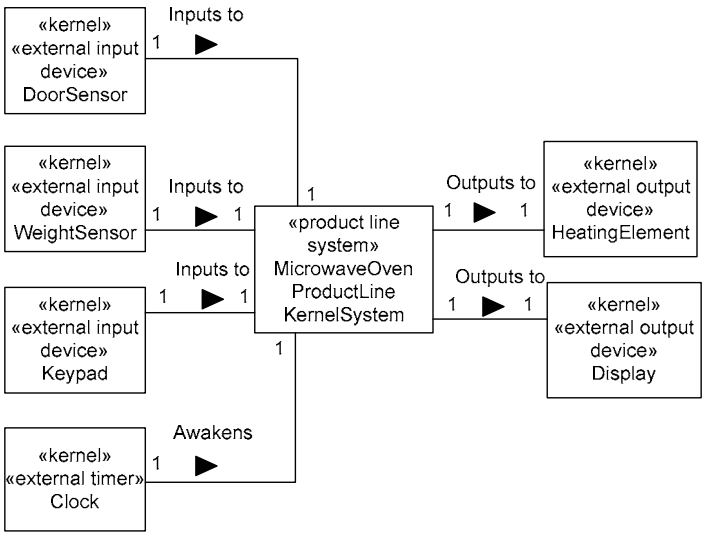
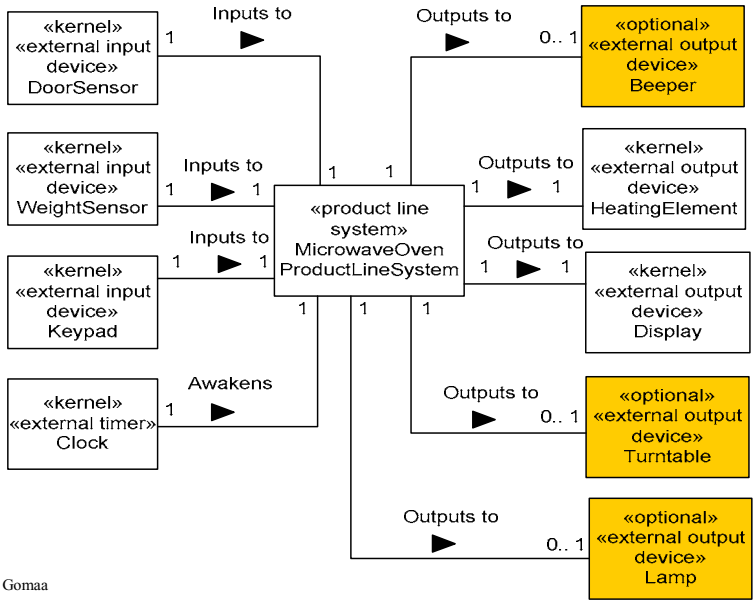


Figure 6.6 Microwave Oven product line context class diagram



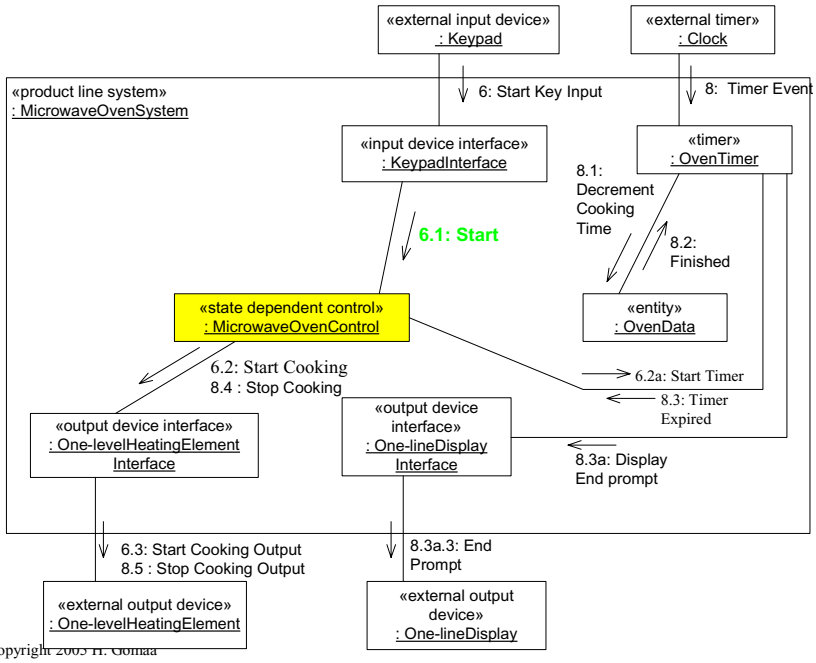
Dynamic Interaction Modeling for Software Product Lines

- Object interaction model
  - Defines how objects participate in use cases
  - Use communication (collaboration) diagrams or sequence diagrams
- Software Product Line Interaction model
  - Determine objects that participate in each use case
  - Determine sequence of messages sent between objects
  - Develop at least 1 interaction diagram for each use case
    - Kernel communication diagrams
    - Optional communication diagrams
    - Alternative communication diagrams

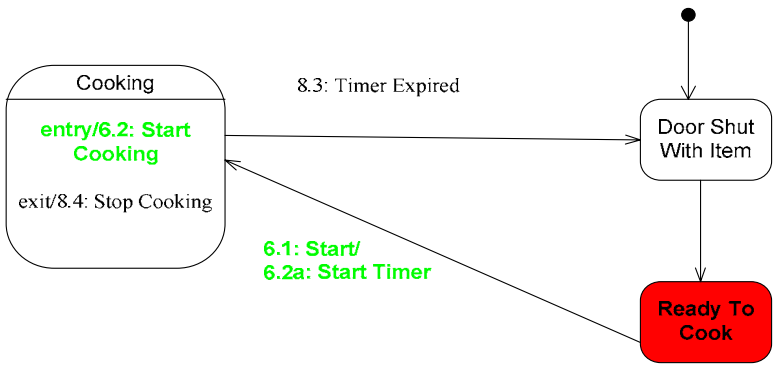
Dynamic Analysis for Software Product Lines

- Kernel First Approach
  - Collaboration diagrams for kernel system are similar to collaboration diagrams for single system
  - Only kernel objects participate in kernel collaboration diagrams
  - Kernel system may or may not be a member of product line
- Software Product Line Evolution
  - Start with kernel collaboration diagrams
  - Consider optional and alternative collaboration diagrams

Communication diagram for Cook Food use case

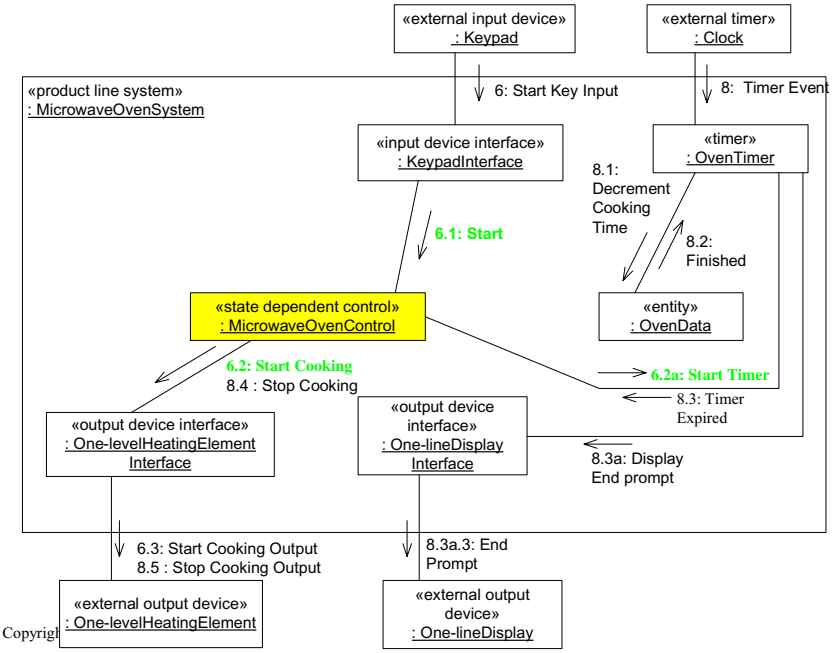


Statechart for Microwave Oven Control

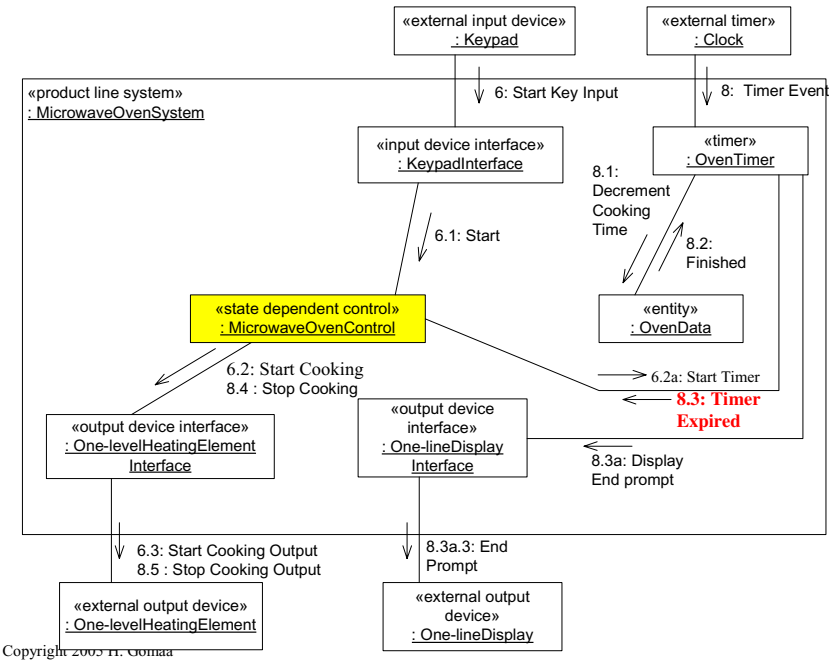


- **Incoming message to object-> input event on statechart**
- **Output event on statechart -> outgoing message from object**

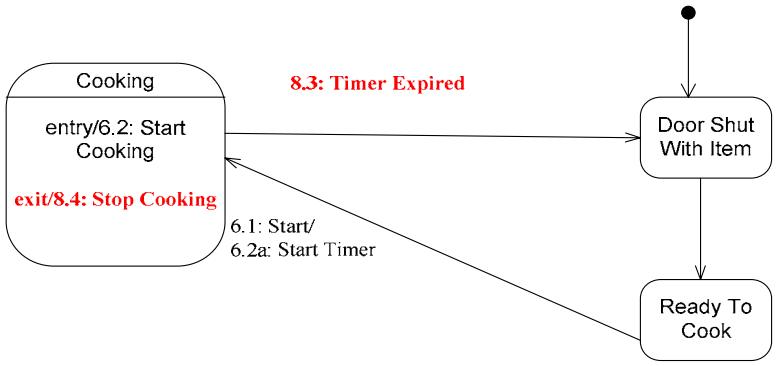
Communication diagram for Cook Food use case



Communication diagram for Cook Food use case



Statechart for Microwave Oven Control



- **Incoming message to object-> input event on statechart**
- **Output event on statechart -> outgoing message from object**

Communication diagram for Cook Food use case

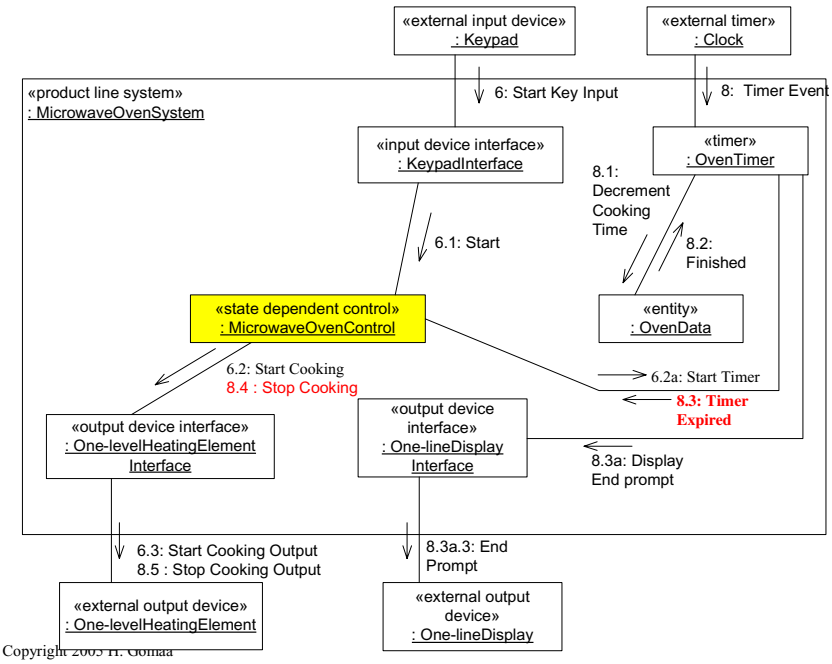


Figure 7.6 Communication diagram for kernel use case: Cook Food

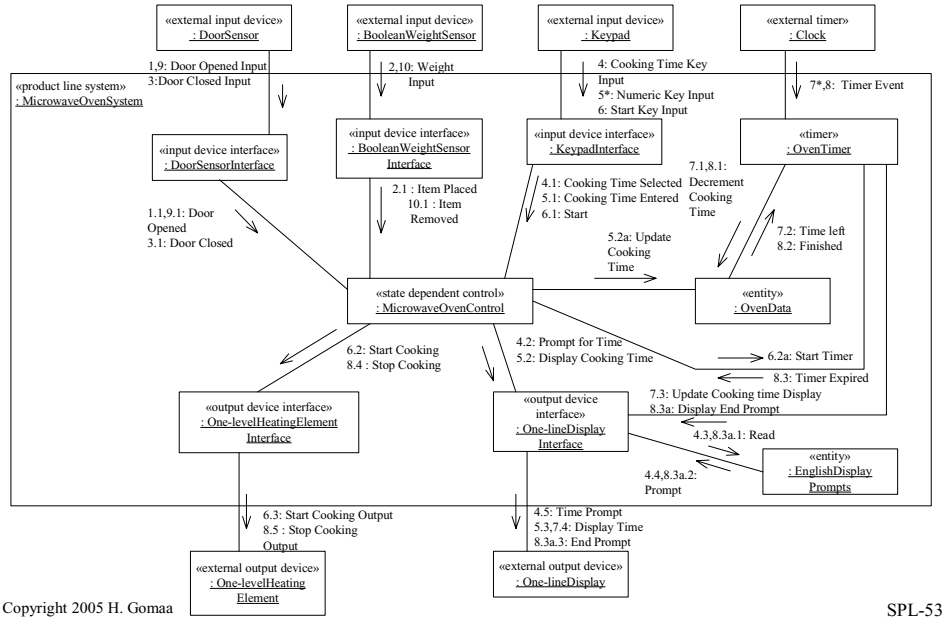
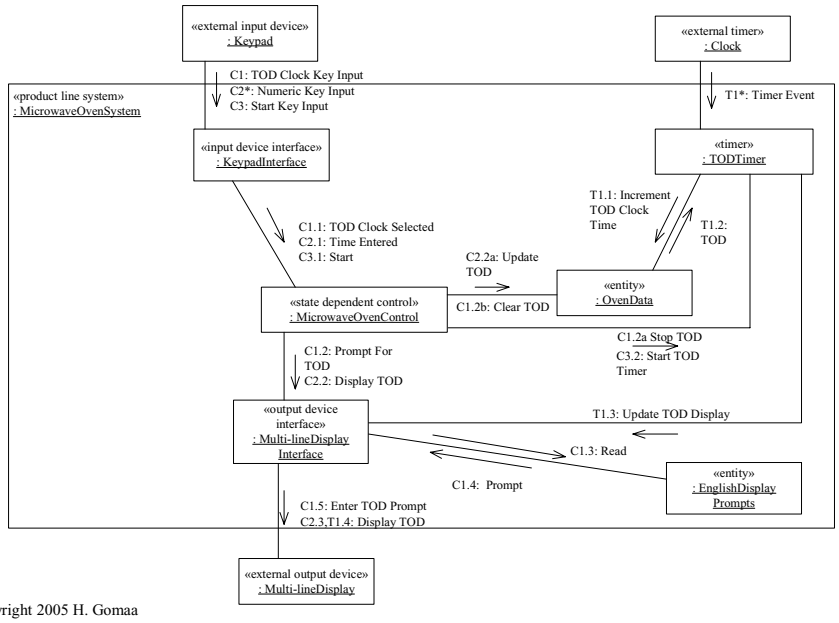


Figure 7.12 Communication Diagram for optional TOD feature



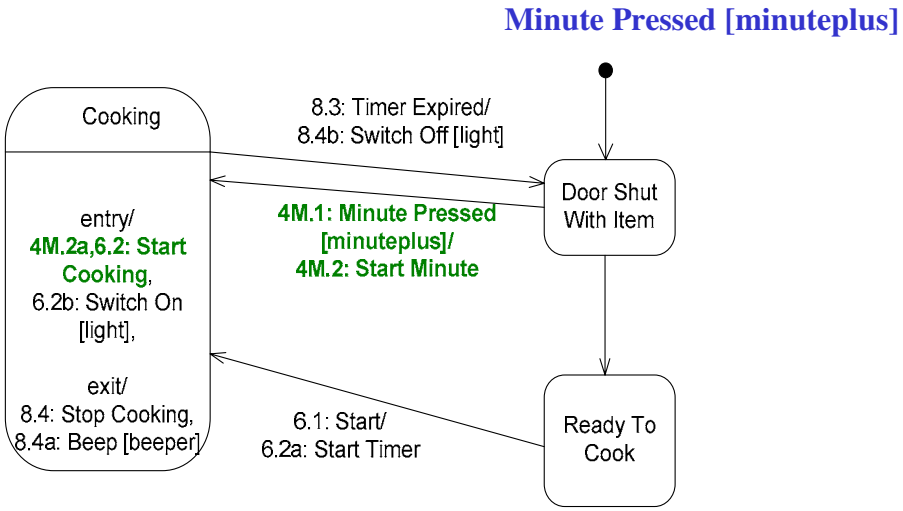
State Machine Models for SPLs

- Design statechart for each state dependent class
- Model state machine variability
  - Inherited vs Parameterized State Machines
- Inherited State Machine
  - Different statechart for each alternative or optional feature
- Disadvantage:
  - Each feature & feature combination needs an inherited state machine
  - Leads to combinatorial explosion of inherited state machines
- Often better to design parameterized state machine

Parameterized State Machine Models

- Design one state machine with states, transitions, events, actions, corresponding to all features
- Feature dependent transition
  - Use feature condition as guard on transition
    - Event [Feature Condition]
  - Feature condition is
    - True if feature Selected
    - False if feature not selected
    - E.g., Minute Pressed [minuteplus]

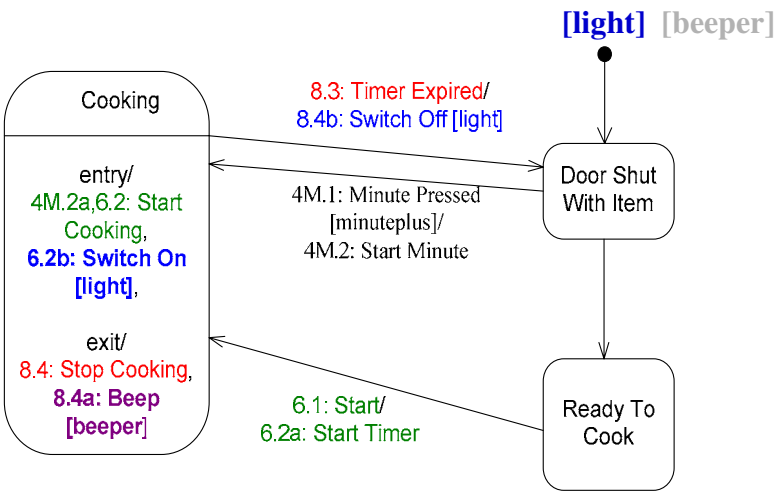
Parameterized Statechart for Microwave Oven Control
(all features)



Parameterized State Machine Models

- **Feature dependent action**
- **Action is only executed if Feature Condition is True**
  - **Action [Feature Condition]**
    - **Switch On [light], Switch Off [light]**

Parameterized Statechart for Microwave Oven Control (all features)



Parameterized State Machine Models

- Feature dependent state identified by UML constraint
- State can only be entered if Feature is selected
  - {feature = recipe} Recipe State
- Transition into state is guarded

Parameterized Statechart for Microwave Oven Control (all features)

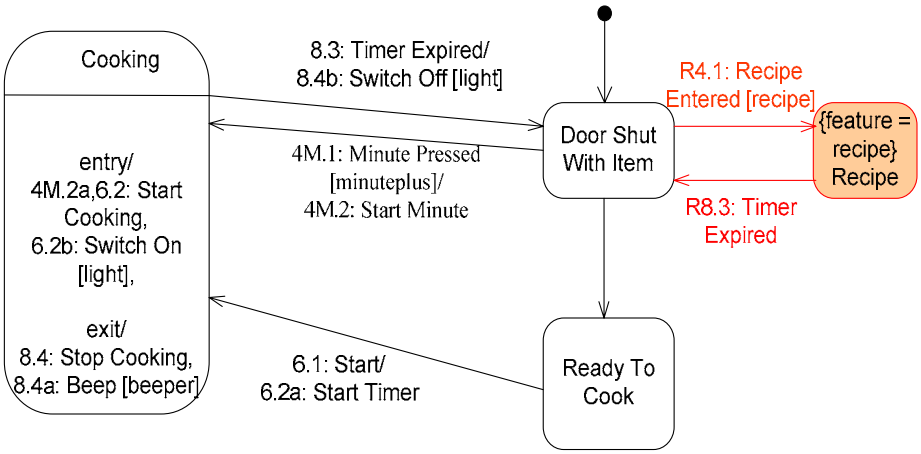


Figure 8.1 Statechart for Microwave Oven Control (kernel functionality)

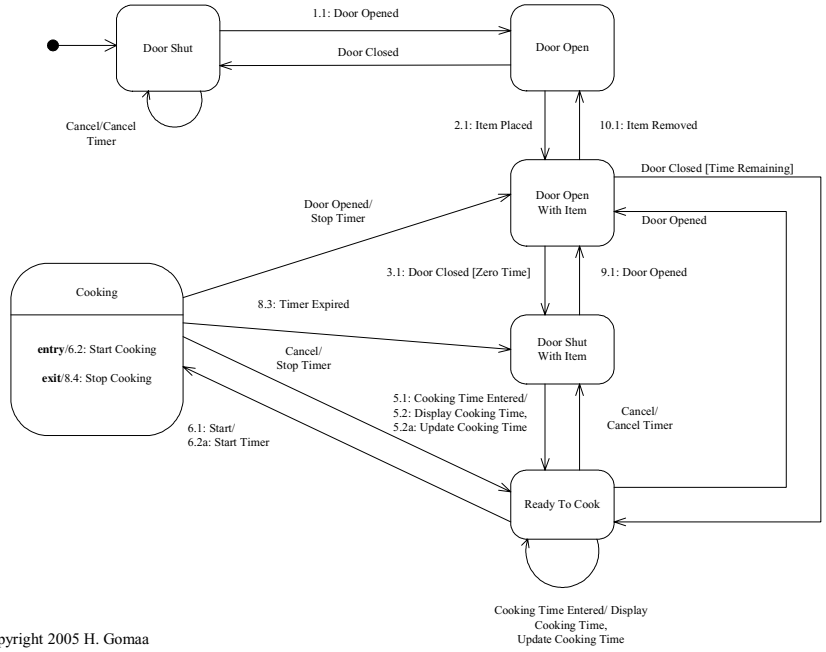
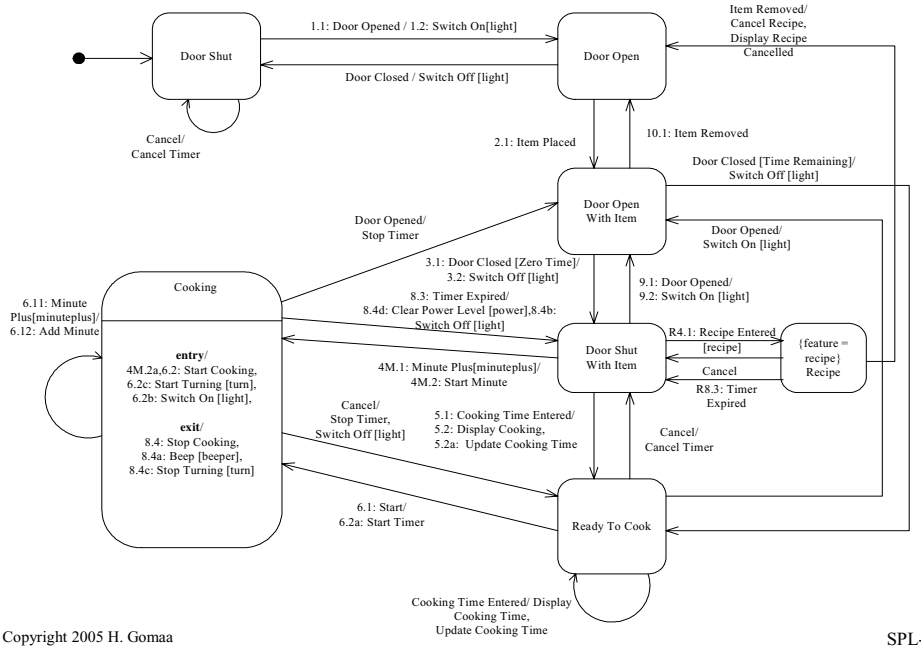


Figure 8.12 Parameterized Statechart for Microwave Oven Control (all features)



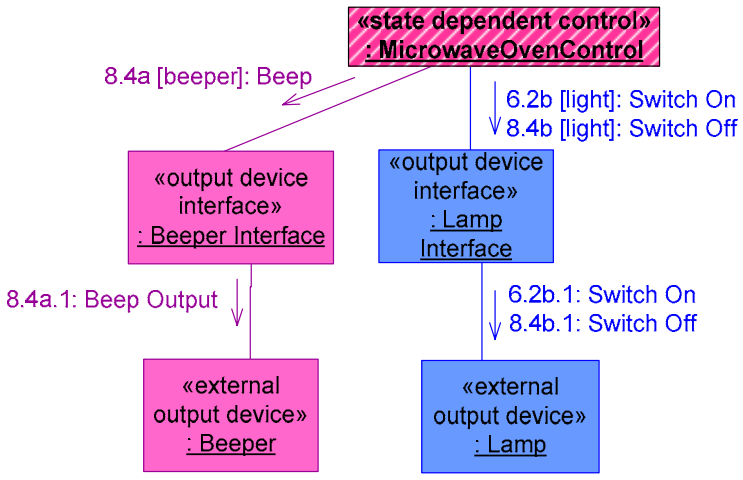
Feature Based Impact Analysis

- Kernel First Approach
  - Develop kernel communication diagrams
- Product Line evolution approach
  - Consider impact of optional and alternative features on kernel communication diagrams
- Analyze impact of each feature on kernel communication diagrams
  - Consider optional objects that need to be added
  - Consider replacement of default objects with variant objects
  - Consider impact on existing kernel objects that communicate with optional / variant objects

Feature Based Impact Analysis

- Kernel First Approach
  - Develop kernel interaction diagrams to realize kernel use cases
- Product Line evolution approach
  - Consider impact of optional and alternative features on kernel
- Analyze impact of each feature
  - Optional object(s) can be added
  - Variant object can replace default object
  - Determine impact on existing kernel objects
    - Communicate with optional / variant objects

Communication Diagram For Microwave Oven SPL
- Impact Of Beeper and Light Features



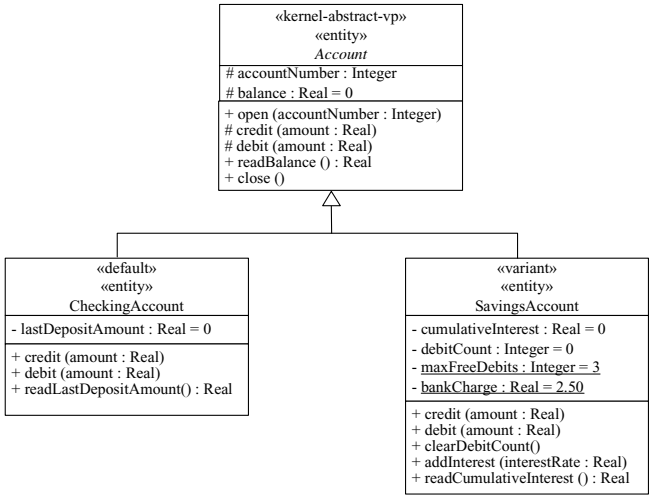
Variation Points in Static Modeling

- A variation point identifies a location at which change will occur in a software product line
- The variation point also identifies the mechanism for a reuser to make the change
- Variation point mechanisms
  - Abstract classes and inheritance
    - Abstract superclass
    - Specialized differently for various members of SPL
  - Parameterization
    - Configuration Parameters
    - Different values for different members of SPL

Abstract Classes in Software Product Lines

- Abstract Class
  - Defines common interface for subclasses
  - Defers implementation of operation(s) to subclasses
  - Captures common properties for all related classes
  - Often represents kernel class
  - Could also be optional
- Subclasses are variant classes
  - Inherit common properties from abstract superclass
  - Extend with variant properties
    - New attributes
    - New operations
    - Alternative implementations of abstract operations

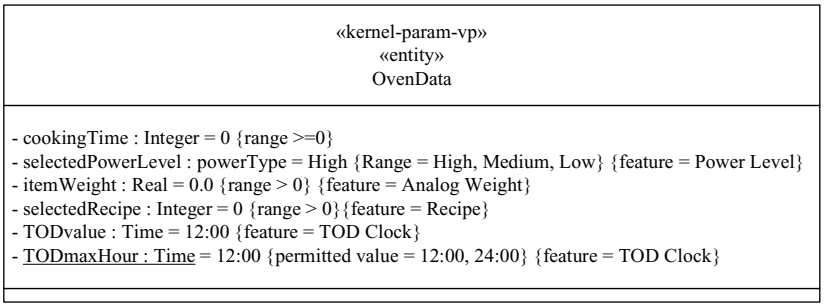
Figure 9.1 Example of abstract superclass and subclasses



Variant vs. Parameterized Classes

- Abstract superclass and variant subclasses
  - Advantage
    - Isolates each variation in one variant class
    - A variant class is impacted by only one feature
  - Disadvantage
    - Could lead to combinatorial explosion of variant classes
- Parameterized Classes
  - Advantage
    - One parameterized class instead of many variant classes
  - Disadvantage
    - A parameterized class can be impacted by more than one feature

Figure 9.2 Example of parameterized class



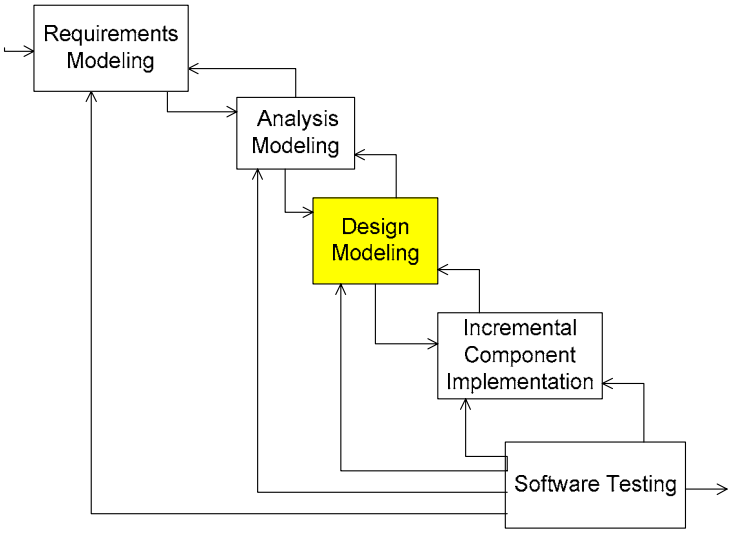
Feature / Class Dependencies

- For each optional or alternative feature, define
  - Optional and/or variant classes required to support feature
  - If feature has prerequisite features
    - Classes in feature may use classes in prerequisite features
  - Features determine which classes can co-exist in same system
- Some variants are mutually exclusive
  - Used by different members of product line
  - E.g., High Volume Workstation Controller, Flexible Workstation Controller, Monitoring Workstation Controller
- Some variants co-exist in same system
  - E.g., High Volume Workstation Controller, Receiving Workstation Controller, Shipping Workstation Controller

Table 9.1 Feature/class dependencies: microwave oven SPL

Feature Name	Feature Category	Class Name	Class Reuse Category	Class Parameter
Microwave Oven Kernel	common	Door Sensor Interface	kernel	
		Weight Sensor Interface	kernel-abstract-vp	
		Keypad Interface	kernel-param-vp	
		Heating Element Interface	kernel-abstract-vp	
		Display Interface	kernel-abstract-vp	
		Microwave Oven Control	kernel-param-vp	
		Oven Timer	kernel-param-vp	
		Oven Data	kernel-param-vp	
		Display Prompts	kernel-abstract-vp	
Light	optional	Lamp Interface	optional	
		Microwave Oven Control	kernel-param-vp	light : Boolean
Turntable	optional	Turntable Interface	optional	
		Microwave Oven Control	kernel-param-vp	turntable : Boolean
Beeper	optional	Beeper Interface	optional	
		Microwave Oven Control	kernel-param-vp	beeper : Boolean
Minute Plus	optional	Keypad Interface	kernel-param-vp	minuteplus : Boolean
		Microwave Oven Control	kernel-param-vp	minuteplus : Boolean
		Oven Timer	kernel-param-vp	minuteplus : Boolean
One-line Display	default	One-line Display Interface	default	
Multi-line Display	alternative	Multi-line Display Interface	variant	
English	default	English Display Prompts	default	
French	alternative	French Display Prompts	variant	
Spanish	alternative	Spanish Display Prompts	variant	
German	alternative	German Display Prompts	variant	
Italian	alternative	Italian Display Prompts	variant	

Product Line Engineering – Design Modeling



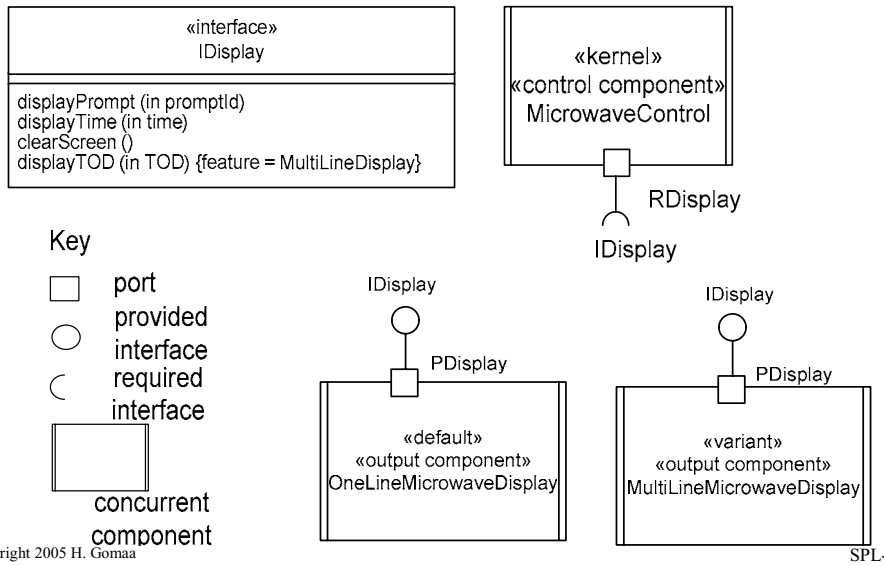
Component-based Distributed Software Architecture

- Executes on multiple nodes in distributed configuration
  - Consists of distributed components
- Distributed component
  - Well-defined *provided* and *required* interfaces
  - Concurrent object
  - Logical unit of distribution and deployment
  - Communicates with other components using messages
  - Structure
    - Composite object consisting of other objects
    - Simple object
  - Capable of being reused

Modeling Components in UML 2.0

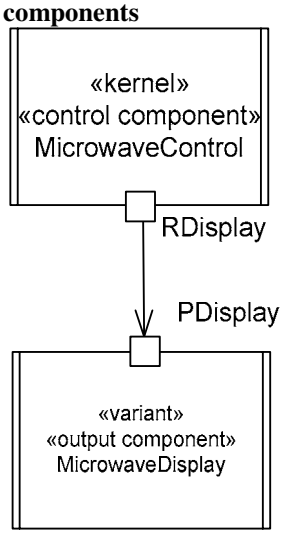
- **Components**
  - Modeled as UML 2.0 structured classes
  - Depicted on UML 2.0 composite structure diagrams
- **Components**
  - Communicate with each other through **ports**
- **Port**
  - Consists of *provided* and/or *required* interfaces
- **Connector**
  - Joins *required* port of one component to *provided* port of another component

Design of individual components

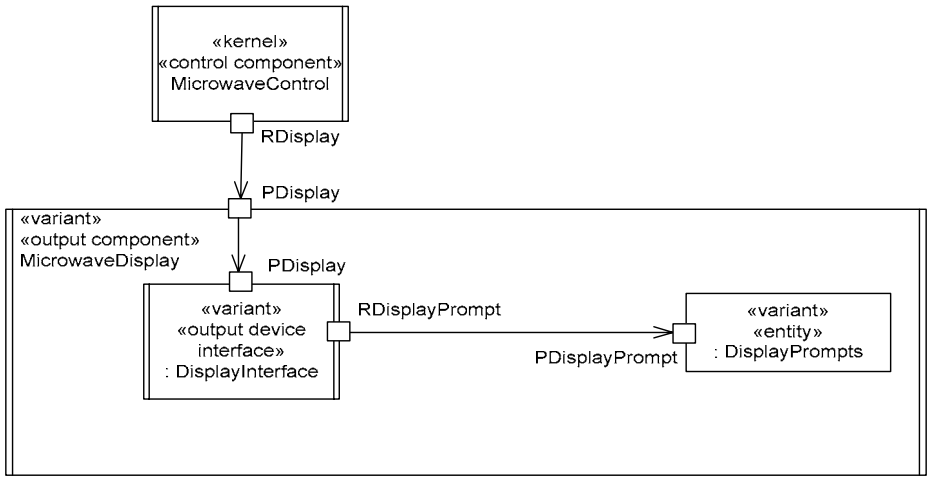


Design of "plug compatible" components

- **Connector**
- Joins *required* port of one component to *provided* port of another component
- **Microwave Control can be connected to either version of Microwave Display**
  - **Less variability**



Design of Composite Component



Software Design and Architectural Patterns

- Design Patterns
  - Small group of collaborating objects
  - Gang of Four (Gamma, Helms, Johnson, Vlissides)
- Software Architectural Patterns
  - Recurring architectures used in various software applications
  - Buschmann, etc. at Siemens
- Architectural Structure Patterns
  - Address structure of major subsystems of system or PL
- Architectural Communication Patterns
  - Reusable interaction sequences between components

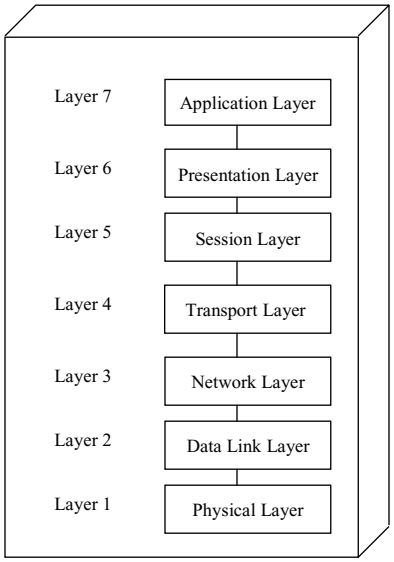
Architectural Structure Patterns for Software Product Lines

- Layered patterns *very important in SPLs*
  - Layers of Abstraction
  - Kernel
- Client/Server patterns
  - Basic Client/Server
  - Client/Broker/Server
  - Client/Agent/Server
- Control Patterns *very important in RT Design*
  - Centralized Control
  - Distributed Control
  - Hierarchical Control

Architectural Structure Patterns for Software Product Lines

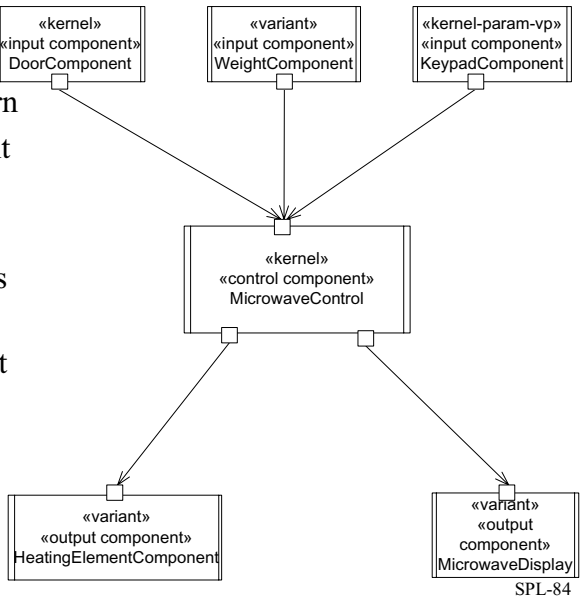
- Layers of Abstraction
  - Structure product line into hierarchical levels
  - Each layer provides services for higher layers
- Layers of Abstraction in Product Lines
  - Allows use of subsets and extensions
  - Lower layers do not depend on upper layers
    - Kernel components at lowest layer
  - Higher layers depend on lower layers
    - Optional and variant components at higher layers
- Kernel Pattern in Software Product Lines
  - Kernel of product line is at lowest layer of hierarchy
  - Optional and variant components at higher layers depend on kernel components in kernel layer

Figure 10.1 Example of layers of abstraction pattern
- ISO open systems interconnection reference model



Centralized Control Pattern
E.g., Microwave Oven Control

- Centralized Control Pattern
  - One control component
    - Executes statechart
  - Receives sensor input from input components
  - Controls external environment via output components

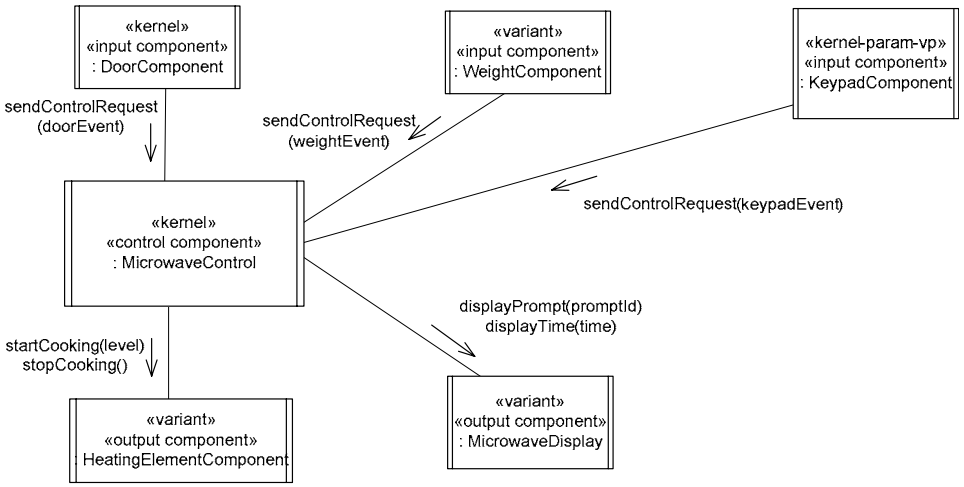


Architectural Communication Patterns for Software Product Lines

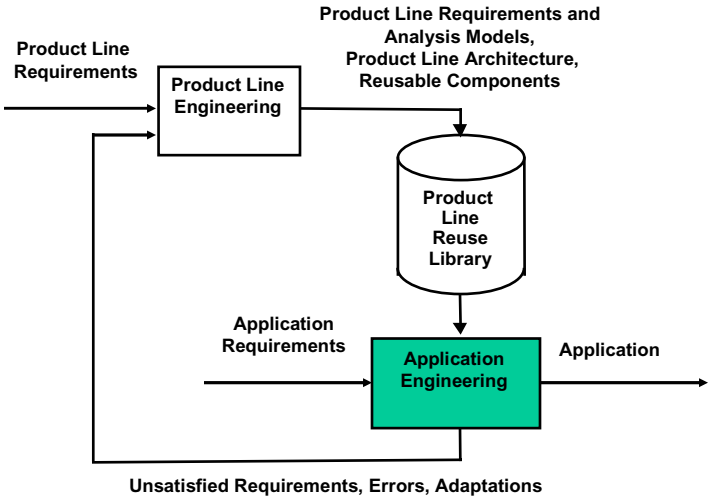
- Peer-to-Peer Communication Patterns
  - Asynchronous communication
  - Bi-directional asynchronous communication
- Client/Server Communication Patterns
  - Synchronous communication with reply
  - Synchronous communication with Callback

- **Very important for evolutionary design:**
- Broker Communication Patterns
  - Broker forwarding
  - Broker handle
  - Discovery
- Group Communication Patterns
  - Broadcast
  - Subscription/notification (Multicast)

Software Architecture for Microwave Oven - asynchronous communication Pattern

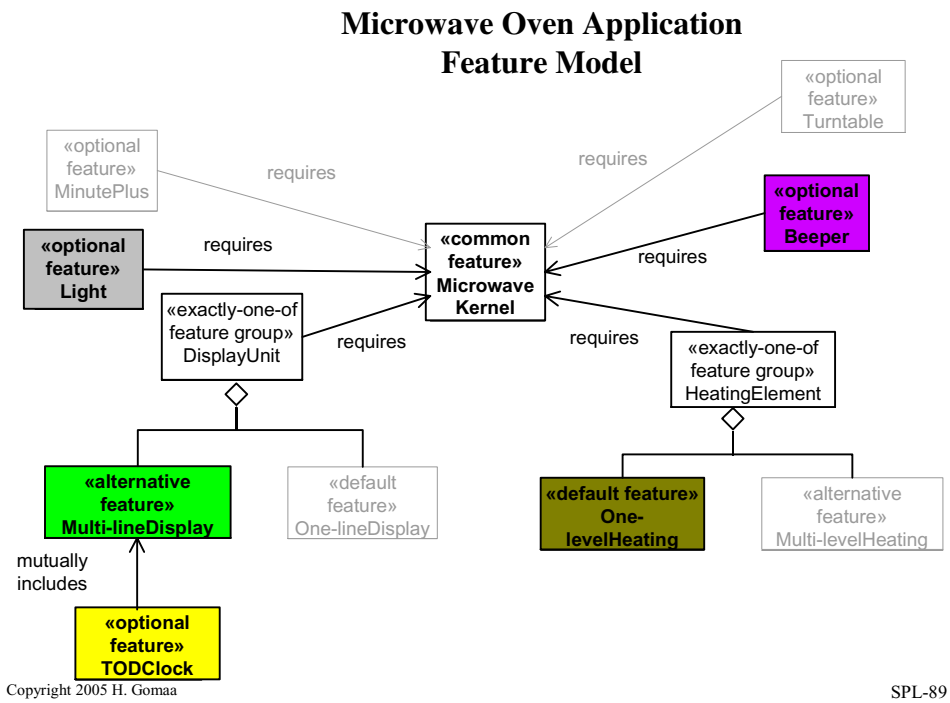


Evolutionary Process Model for Software Product Lines



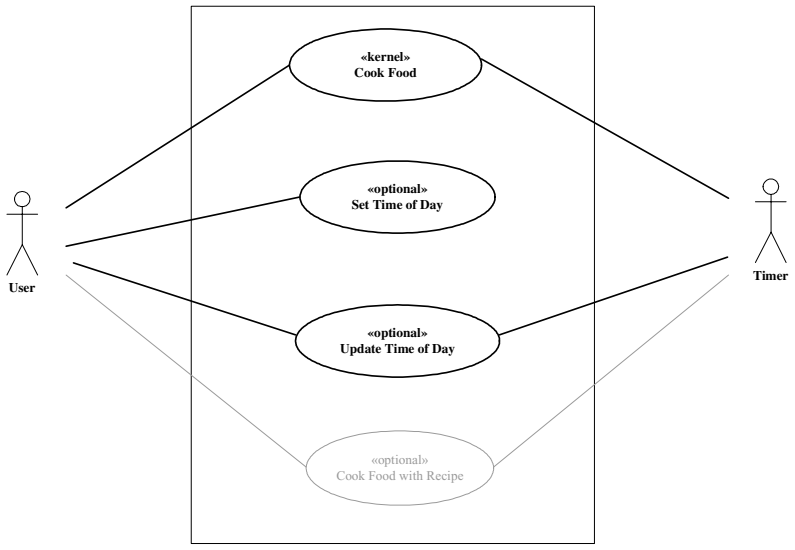
Software Application Engineering

- Software Application
  - Member of software product line
- Software Application Engineering
  - Derive application architecture from SPL architecture
- Select application features subject to
  - Feature dependencies and relationships
- Derive software application architecture
  - Kernel components always selected
  - Optional and variant components correspond to features selected



SPL-89

Figure 12.7 Use Case model for Microwave Application



SPL-90

Table 12.1 Microwave oven application: feature/use case dependencies

Feature Name	Feature Category	Use Case Name	Use Case Category/ Variation Point (vp)	Variation Point
Microwave Oven Kernel	common	Cook Food	kernel	
Light	optional	Cook Food	vp	Light
Beeper	optional	Cook Food	vp	Beeper
Multi-line Display	alternative	Cook Food	vp	Display Unit
French	alternative	Cook Food	vp	Display Language
Boolean Weight	default	Cook Food	vp	Weight Sensor
One-level Heating	default	Cook Food	vp	Heating Element
TOD Clock	optional	Set Time of Day Display Time of Day	optional optional	
12/24 Hour Clock	parameterized	Set Time of Day Display Time of Day	optional	12/24 Hour Clock

Figure 12.8 Microwave Oven Application context class diagram

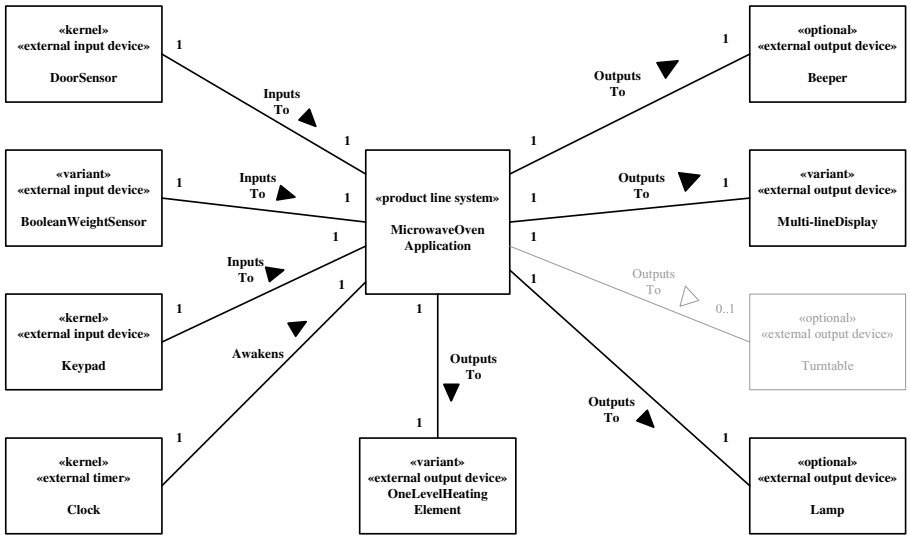


Table 12.2 Microwave oven application: feature/class dependencies

Feature Name	Feature Category	Class Name	Class Reuse Category	Class Parameter
Microwave Oven Kernel	common	Door Sensor Interface	kernel	
		Weight Sensor Interface	kernel-abstract-vp	
		Keypad Interface	kernel-param-vp	
		Heating Element Interface	kernel-abstract-vp	
		Display Interface	kernel-abstract-vp	
		Microwave Oven Control	kernel-param-vp	
		Oven Timer	kernel-param-vp	
		Oven Data	kernel-param-vp	
		Display Prompts	kernel-abstract-vp	
Light	optional	Lamp Interface	optional	
		Microwave Oven Control	kernel-param-vp	light : Boolean
Beeper	optional	Beeper Interface	optional	
		Microwave Oven Control	kernel-param-vp	beeper : Boolean
Multi-line Display	alternative	Multi-line Display Interface	variant	
French	alternative	French Display Prompts	variant	
Boolean Weight	default	Boolean Weight Sensor Interface	default	
One-level Heating	default	One-level Heating Element Interface	default	
TOD Clock	optional	TOD Timer	optional	
		Keypad Interface	kernel-param-vp	TODClock : Boolean
		Microwave Oven Control	kernel-param-vp	TODClock : Boolean
		Oven Data	kernel-param-vp	TODvalue : Real
12/24 Hour Clock	Parameterized	Oven Data	kernel-param-vp	TODmaxHour : Integer

Software Architecture for Microwave Oven Application - Component Structuring

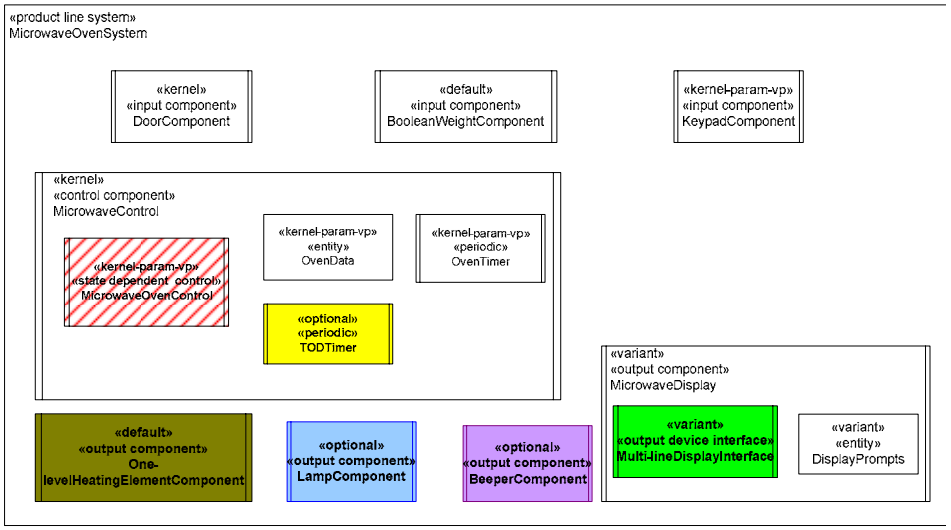
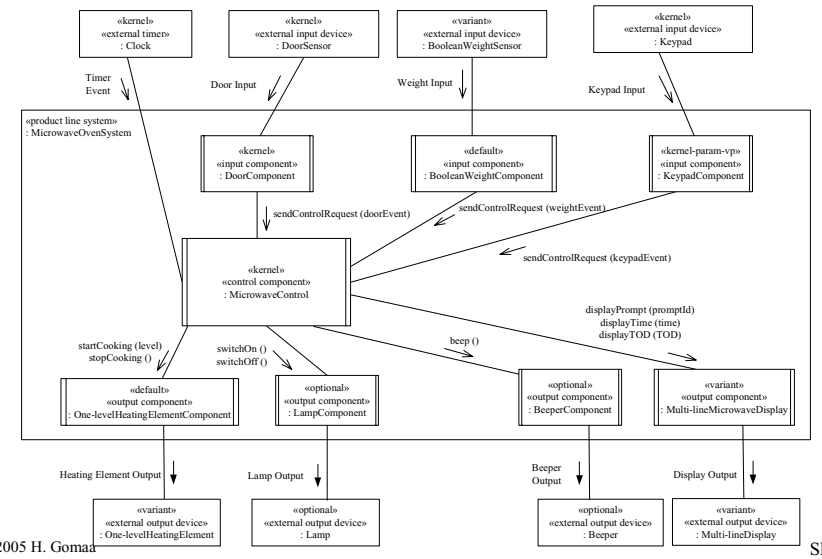


Figure 12.14 Software architecture for Microwave Oven Application - message communication



Microwave Oven Application architecture

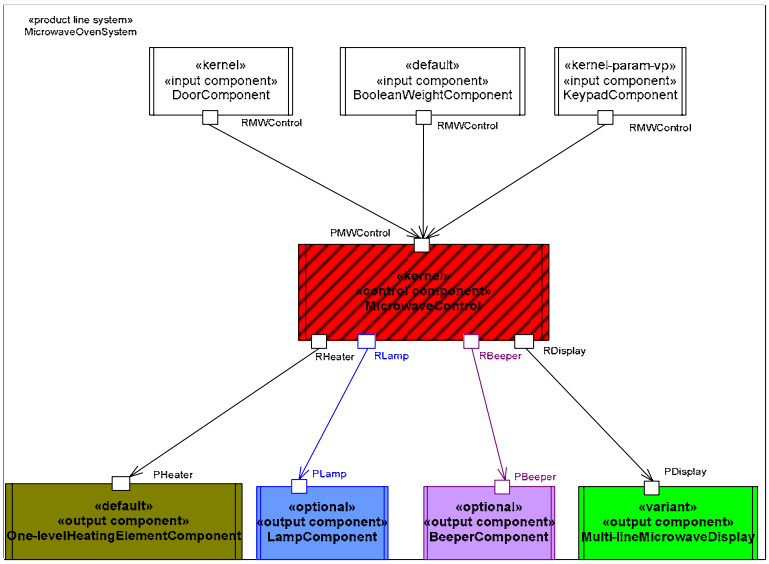
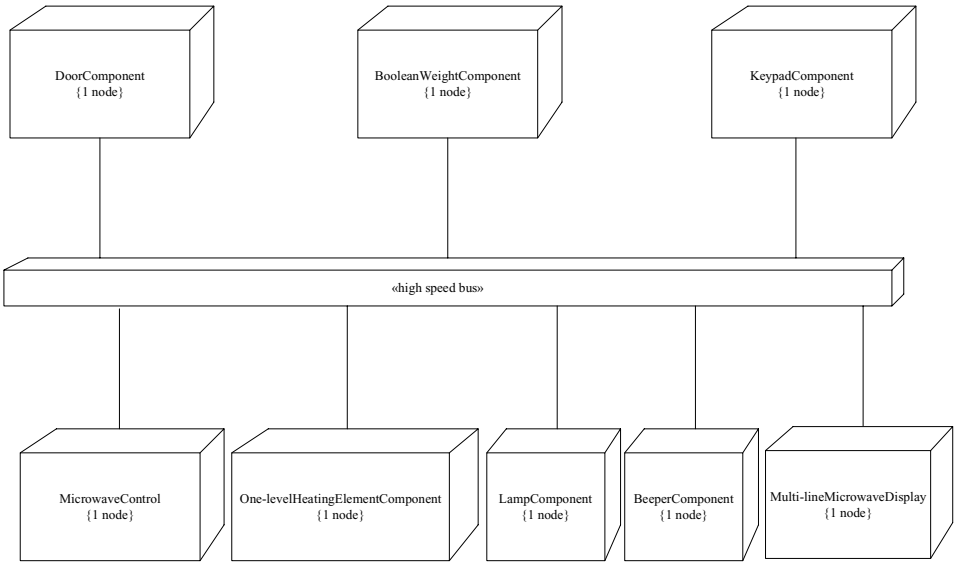


Figure 12.16 Distributed Application Configuration



Review

- 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
  - H. Gomaa, “Designing Software Product Lines with UML: From Use Cases to Pattern-based Software Architectures”, Addison Wesley Object Technology Series, July 2004

ABSTRACT: Software Product Line Development advocates software reuse by modeling common and variable artefacts separately across members of a family of products. Aspect-Oriented Software Development aims at separation of concerns with â€œaspectsâ€ to increase modularity, reusability, maintainability and ease of evolution. In this paper, we apply an as-pect-oriented use case modeling approach to product line system modeling. A use case specification captures stake-holders concerns as interactions between a system and its actors. We adapt our previous work with the introduction of a â€œvariabilityâ€ r 40 Open Source, Free and Top Unified Modeling Language (UML) Tools : Review of Top Open Source and Free Unified Modeling Language (UML) Tools including ArgoUML, StarUML, UMLet, Dia, BOUML, Violet, EclipseUML, gModeler, RISE, NClass, NetBeans IDE, GenMyModel, Plantuml, UML Modeller, Open ModelSphere, Oracle Jdeveloper, Papyrus, Oracle SQL Developer are the Top Open Source and Free Unified Modeling Language (UML) Tools.Top Unified Modeling Language (UML) Tools : Review of Top Unified Modeling Language (UML) Tools including IBM Rational Rose, Visio, StarUML, Visual Paradigm, Sparx Enterprise Arch Explore a Service-Oriented Software product lines (SoSPL) methodology that applies SPL variability analysis techniques to Web services to design customized service-based applications. Find out how software product lines (SPL) promote agile and flexible application development for evolving system families. And discover how the adoption of SPL principles can provide a systematic way to analyze and design service-oriented applications.Â You can model business processes using Unified Modeling Language (UML) activity diagrams. Activity diagrams are particularly useful in detailing the workflow of business processes during domain business modeling. Figure 1 shows a hotel reservation process.