Computer Science

ILLINOIS INSTITUTE OF TECHNOLOGY

“Software Product Line Overview”

Professor: Dr. Atef Bader

Course: CS587 Program Project Management

Project Team: Piyush Hargunani

Devinder Singh

Steve Talbot

Software Product Lines (SPLs)

- "Software Product Line" (SPL)
  - 1 or more similar variant "assets" derived from common base "assets" [5]
    - "Family of Systems" [e4]
    - Share some <u>common</u> features
    - Have some <u>variable</u> features (constitute the "Variants").
  - The SPL consists of [5]
    - Product Line Architecture (PLA)
    - Reusable components
    - Products derived from shared assets ("Variants"), product lines ("PLs").
  - Major Goals of the SPL [6]
    - Reduced "time-to-market".
    - Ability to produce products with "fewer people".
    - Ability to utilize <u>outsourcing</u> and <u>contractors</u>.
    - Ability to leverage "Legacy Systems", "Architecture Recovery"
    - "Product Alignment"
      - Ability to have products exhibit the same "look and feel".
      - Ability to <u>certify</u> safety-critical systems.

Software Product Lines (SPLs)

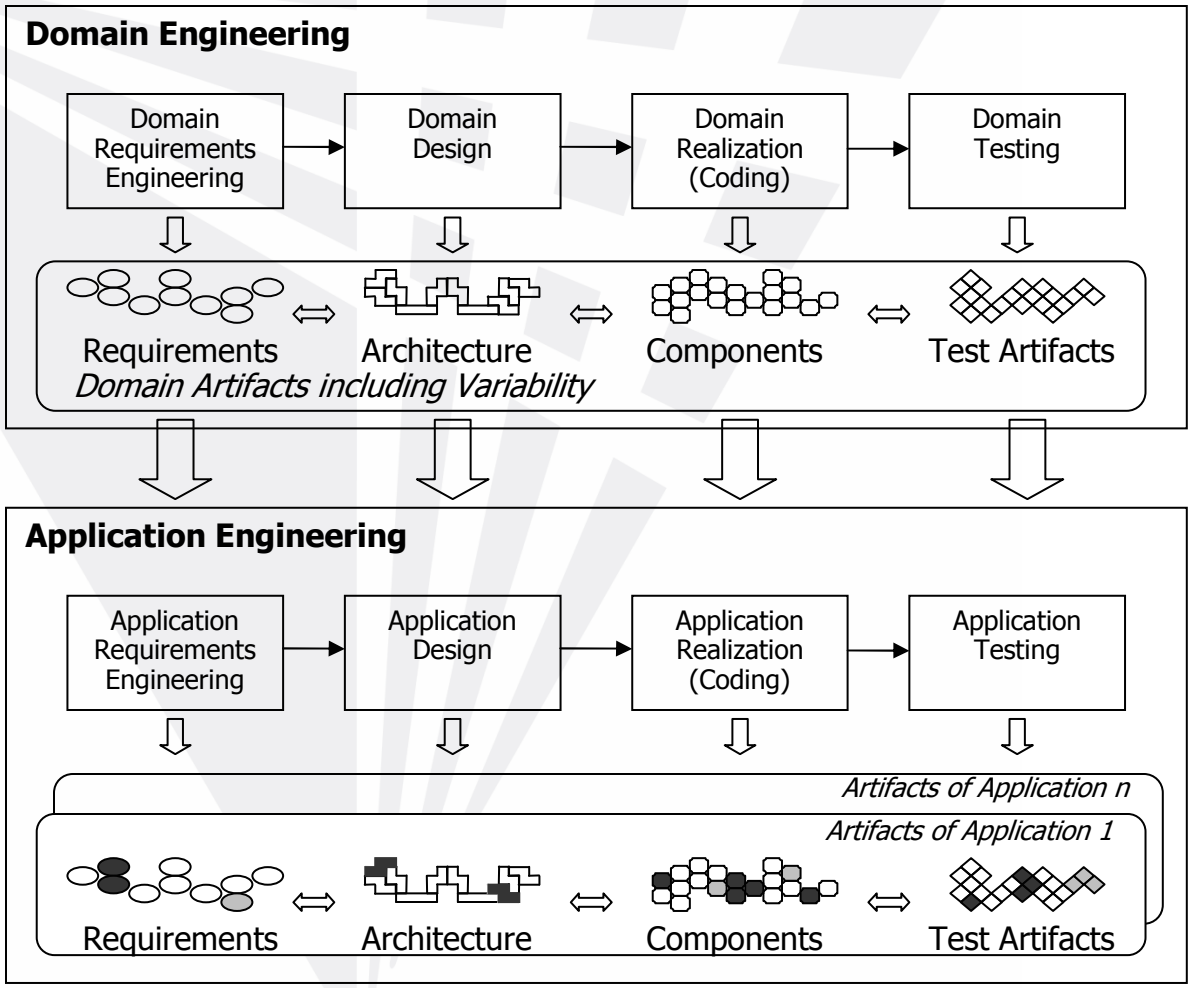
- "Software Product Line" (SPL) (cont) [e3]
  - "Core Assets"
    - "Core Artifacts"
      - "Common Assets": common across the family of products.
      - "Variant Assets": specific to products in the SPL.
      - Requirements, Design, Architecture, Code, Tests, Documentation.
    - "Core Products"

Organizational Models

- Structure of the "Typical" Organization [5]
  - A single "Domain Engineering Unit" (D.E.U.).
    - "For Reuse": <u>create software assets</u> for reuse. [e2]
  - Several "Application Engineering Units" (A.E.U.s).
    - "With Reuse": <u>Use assets</u> made by D.E.U. to assemble products. [e2]
  - Maintenance and changes to SPL difficult. [e2]
    - Changes must merge into product-specific software for every product.
    - "us-versus-them" culture between D.E.U. and A.E.U.
      - Who is responsible for changes?
- Many "Alternative" Organization Models exist for managing SPLs. [5]
  - "Development Departments"
  - "Business Units"
  - "Domain Engineering Units"
  - "Hierarchical Domain Engineering Units"

Organizational Models (cont)

- Diagram of the "Typical" organization Model. [e1]

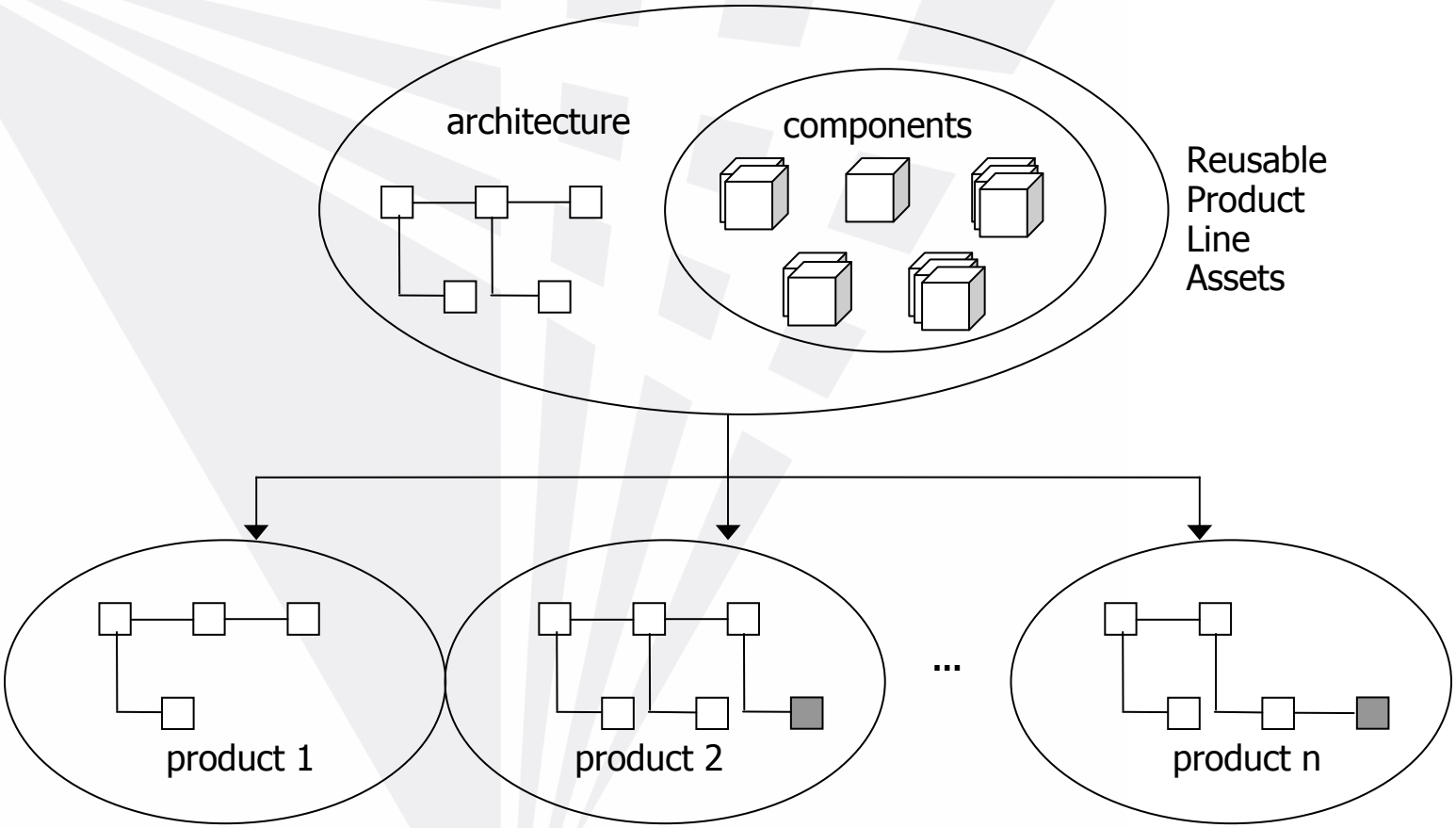


Organizational Models (cont)

- "Development Departments" [5]
  - Suitable for staff of 1 – 30 people.
  - No permanent organization; staff assigned to project temporarily.
  - "Domain Engineering"
    - Develop new, "Reusable Assets" or versions for internal use only.
  - "Application Engineering"
    - Develop new or new version of system for external customers.
    - Develop "extensions" to components for internal use only.
  - Advantages
    - <u>Simplicity</u> and <u>Ease of Use</u>
    - Staff "localized" in "same organizational context"; little overhead.
  - Disadvantages
    - If favors "Domain Engineering", high quality, low reusability.
    - If favors "Application Engineering", low quality, high reusability.

Organizational Models (cont)

- Diagram of the "Development Departments" Model. [5]



Organizational Models (cont)

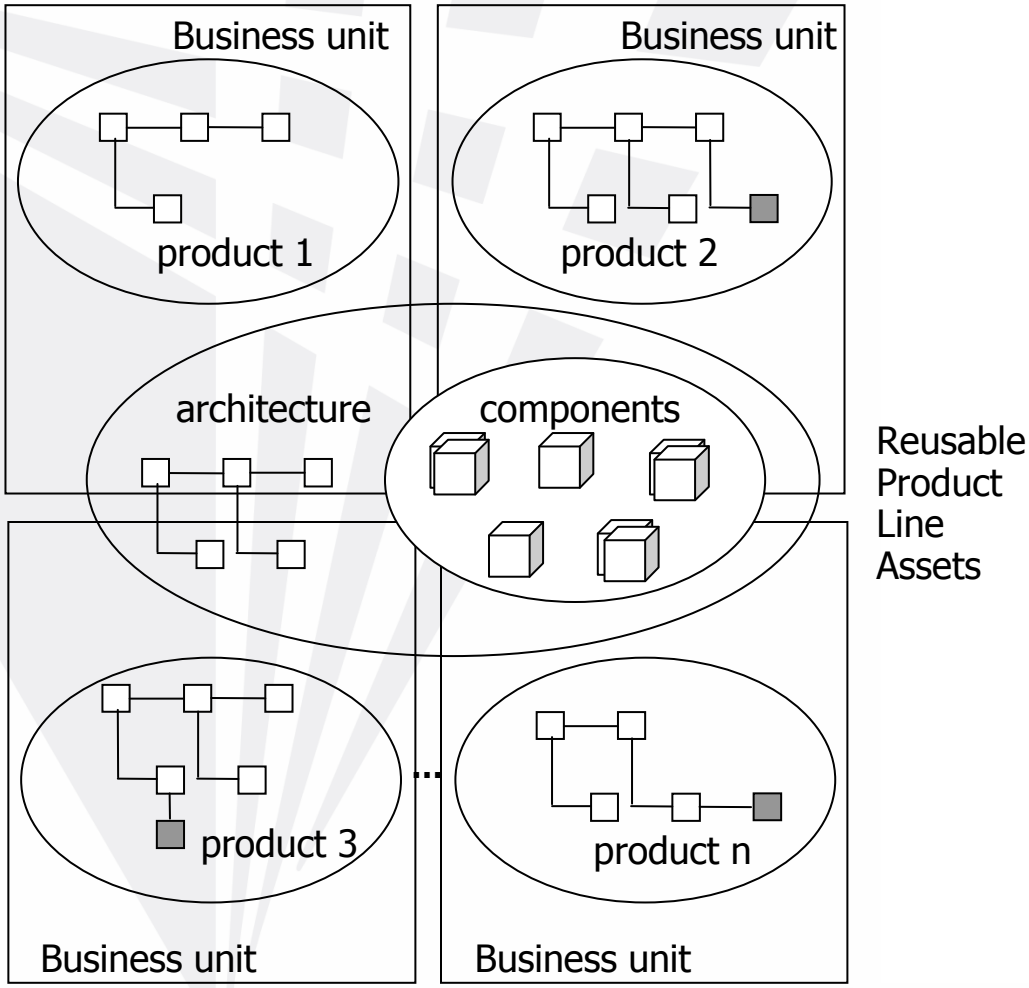
- "Business Units" [5]
  - Suitable for staff of 30 – 100 people.
  - Each unit is responsible for development and evolution of 1 or more products in SPL.
  - "Reusable Assets" are shared, and each unit is responsible for extending, testing and distributing versions to the other units.
  - Goal is to avoid "degradation of components", when it becomes easier to develop "system-specific" versions instead of extending versions.
  - Advantages
    - Allows for effective sharing of assets between "organizational units".
  - Disadvantages
    - No entity or explicit incentive to focus on "Shared Assets".
    - Quality, reliability, timely release depends on individuals only.

Organizational Models (cont)

- "Business Units" (cont) [5]
  - 3 "Levels of Maturity"
    - "Unconstrained Model"
      - Significant component degradation.
      - Any "business unit" can extend any "Shared Asset" and make it available in the "Shared Asset Base".
    - "Asset Responsibilities"
      - Less component degradation.
      - "Gatekeeper" controls extension of "Shared Assets".
    - "Mixed Responsibility"
      - Least component degradation, but high "request" overhead.
      - Each "Business Unit" is assigned the responsibility of 1 or more assets, in addition to its responsibility over product(s).
      - If "Business Units" wish to extend "Shared Asset", must "Issue a Request" to "Business Unit" responsible for "Shared Asset".

Organizational Models (cont)

- Diagram of the “Business Units” Model. [5]



Organizational Models (cont)

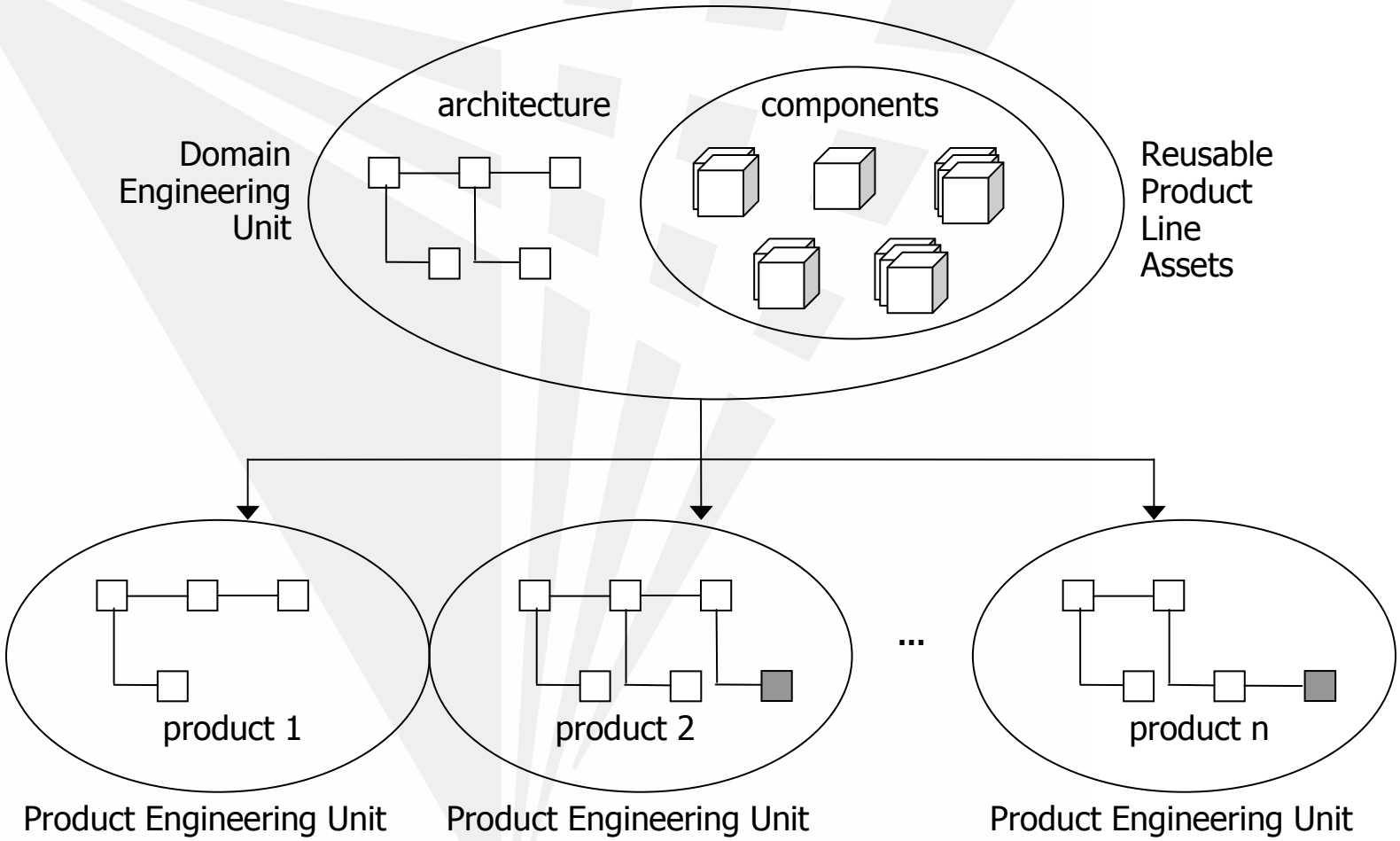
- "Domain Engineering Unit" [5]
  - Suitable for staff of 100 - 999 people.
  - Goal is to separate the development and evolution of "Shared Assets" from that of "concrete systems".
  - "Domain" and "Product" Engineering Units both handle design of
    - Architecture and Components.
    - "Domain Engineering Units" also manage
      - "Shared Assets".
    - "Product Engineering Units" also manage
      - "Concrete systems".
      - Frequent user contacts
  - "Application Engineering Units"
    - Handle implementation and integration of existing assets and systems.

Organizational Models (cont)

- "Domain Engineering Unit" (cont) [5]
  - "Single Domain Engineering Unit".
    - Suitable for units with less than 30 staff members.
    - Manages all "Shared Assets" (Architecture + Components).
    - Sole contact with "Product Engineering".
  - "Multiple Domain Engineering Units".
    - Suitable for units with greater than 30 staff members.
    - One unit responsible for Architecture, others for Components.
    - "Product Engineering" interacts with multiple "Domain" units.
  - Advantages
    - Reduces "n-to-n" communication to "1-to-n" communication.
    - More objective, conflict-resolution / compromise oriented.
  - Disadvantages
    - High intercommunication required; requirements management difficult.
    - "Domain Engineering" requirements flow via "Product Engineering".
    - "Requirements Delays" and "Feature Release Delays" common.

Organizational Models (cont)

- Diagram of the "Domain Engineering Unit" Model. [5]



Organizational Models (cont)

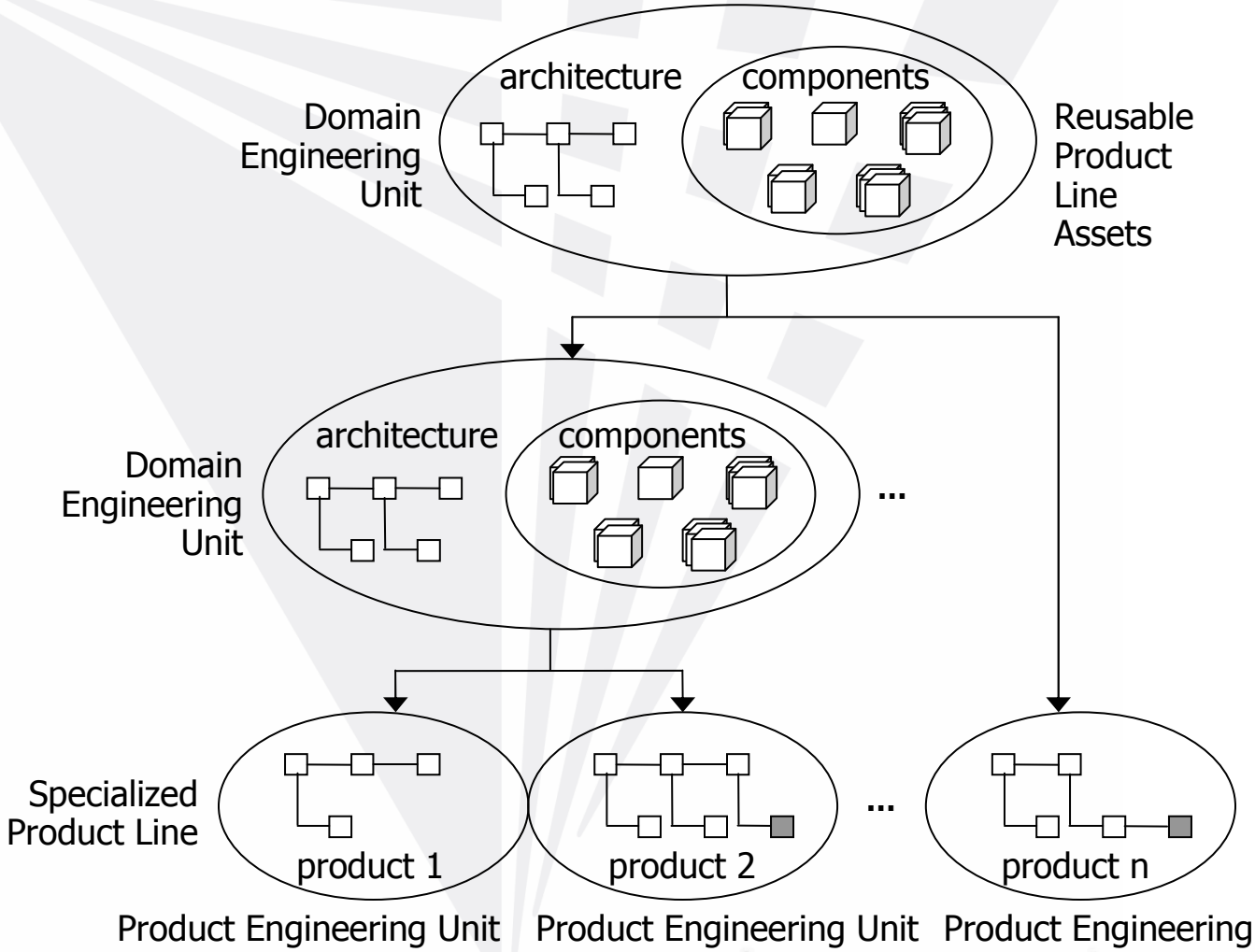
- "Hierarchical Domain Engineering Units" [5]
  - Suitable for staff of 100's to 1000's of people or greater.
  - 1 or more specialized product lines, with an extensive family of SPLs.
  - During design or evolution of product line, becomes necessary to organize the SPLs in a hierarchical manner.
  - A considerable number of staff members is involved in each product line.
  - SPL "Reusable Assets" at the top level are referred to as the "Platform".
    - The "Platform" provides the "Shared Assets".
    - "Domain Engineering Units" develop specialized SPL using "Platform".
      - Lower D.E.U.s "derive" platform features from higher D.E.U.s
      - Lower D.E.U.s are more specialized.
      - Higher D.E.U.s are more generalized.

Organizational Models (cont)

- "Hierarchical Domain Engineering Units" (cont) [5]
  - Advantages
    - Encompasses large, complex product lines.
    - Organizes a large number of engineers.
  - Disadvantages
    - Considerable overhead
      - Communication.
      - Integration.
      - Removing redundancies.
    - Difficult to <u>adjust processes</u> to match <u>market demands</u>
      - Loss of process agility.
      - Increased delays common.

Organizational Models (cont)

- Diagram of the "Hierarchical Domain Engineering Units" Model. [5]



Organizational Models (cont)

- Factors for selecting an "Organizational Model" [5]
  - Geographical distribution of company and staff
    - Disparate locations, time zones, less efficient.
    - Co-located teams, more efficient.
  - Project Management maturity.
    - Project complexity.
    - Project dependencies.
  - Organizational culture
    - Attitudes of Staff.
    - "Cowboy" or "Hero" culture.
    - SPL-approach attitudes.
  - Changes to products in SPL
    - Frequent changes suitable for smaller enterprises.
    - Stable products better for larger companies, long-lifetime projects.

Organizational Models (cont)

- "Organizational Dimensions" (cont) [5]
  - "SPL Assets"
    - "Assets" that are considered "product line wide".
      - Architecture
      - Components
    - "4 levels of asset concentration within organization as a whole"
      - "Architecture"
        - Limited integration between units.
        - Architecture of "Shared Assets" common to all products.
      - "Platform"
        - Architecture-level established.
        - Increased integration between units.
        - Architecture of "Shared Assets" common to all products.
      - "Components"
        - Platform-level established.
        - "Shared Assets" common to only a few products.
      - "Product Specifics"
        - Components-level established.
        - "Shared Assets" include unique product-specific code.

Organizational Models (cont)

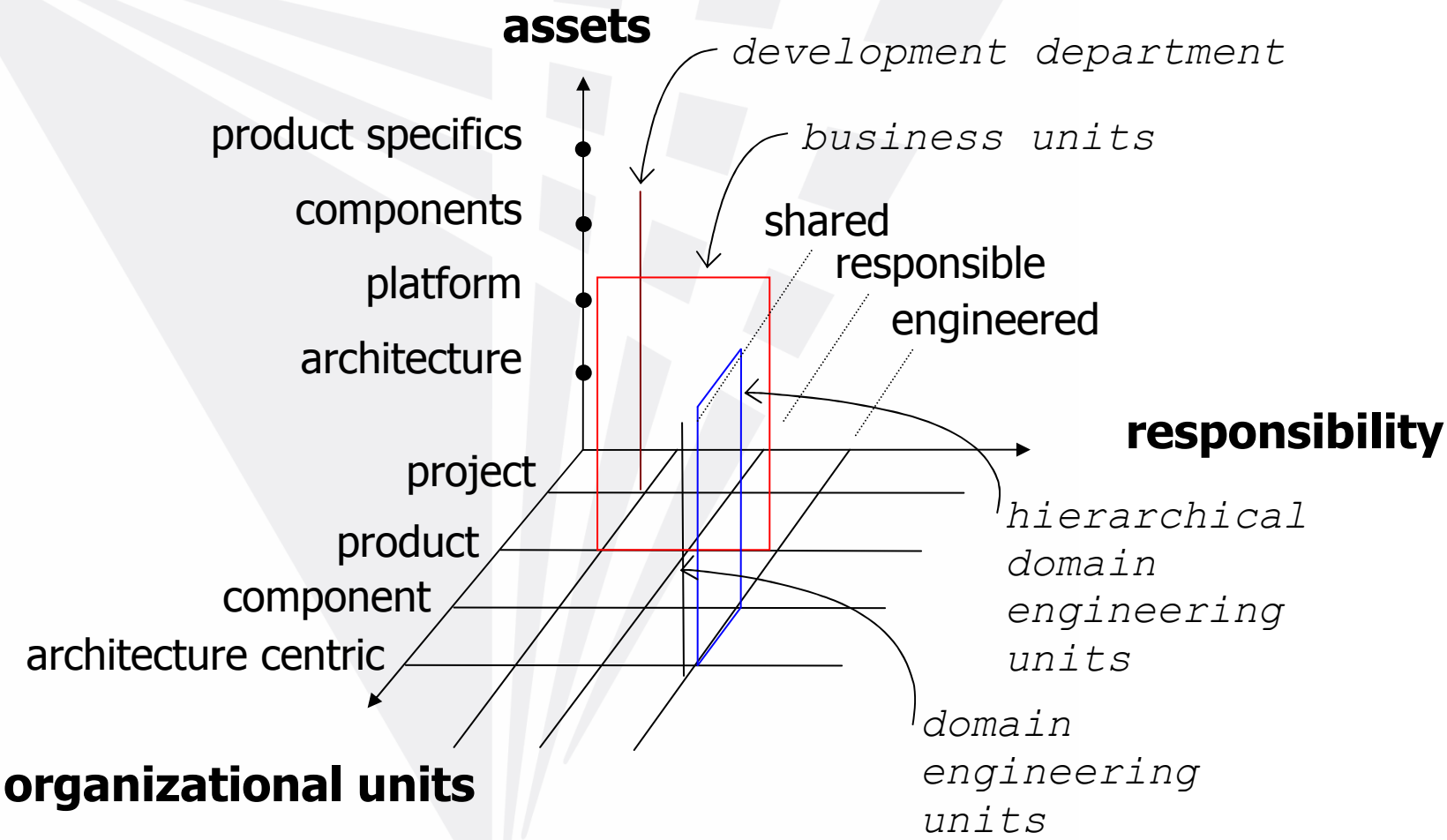
- "Organizational Dimensions" (cont) [5]
  - "Responsibility Levels"
    - Management of the responsibility for SPL assets.
    - "3 Levels of Responsibility"
      - "Shared"
        - Responsibility shared among all units.
      - "Responsible"
        - Person or small team ("the responsible") assigned each asset.
        - Manages asset changes, preventing violations of requirements.
      - "Engineered"
        - Team is assigned each asset.
        - Manages asset development and evolution.
        - Changes controlled via "Change Requests".

Organizational Models (cont)

- "Organizational Dimensions" (cont) [5]
  - "Organizational Units"
    - The nature of how the staff is organized into units.
    - "4 Perspectives"
      - "Project"
        - Staff assigned to teams for duration of project, reassigned.
      - "Product"
        - Staff are permanently assigned to a particular product.
        - Experience increases efficiency, not shared among products.
      - "Shared Components"
        - Used when different products have "considerable overlap".
        - Components are assigned to and controlled by specific units.
      - "Architecture Centric"
        - Architecture shared among products, controls development.

Organizational Models (cont)

- Diagram of the "Organizational Dimensions". [5]

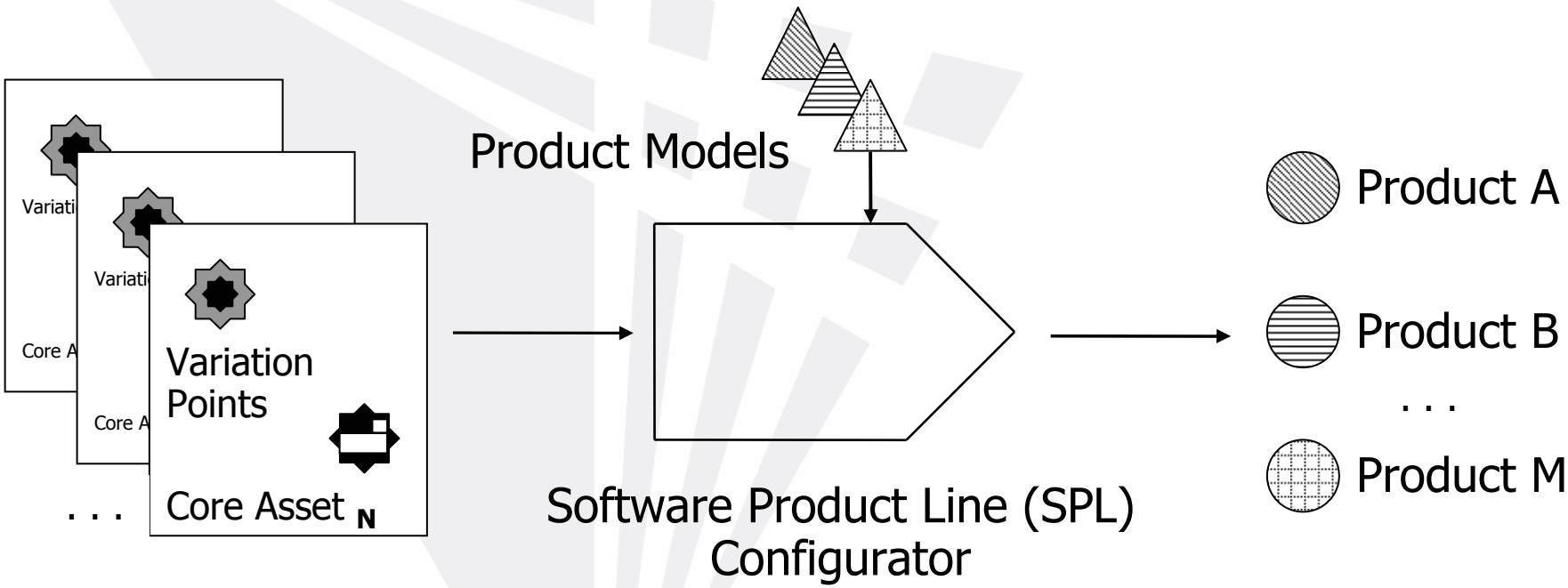


Organizational Models (cont)

- "SPL Configurator" [e2]
  - Avoids negative effects of "Application Engineering" of traditional model.
  - Everything is a candidate for "Reuse" and "Refactoring".
  - Automated instantiation of products.
    - Inputs
      - "Assets"
        - Common and Variant.
        - Requirements, Architecture, Design, Code, Tests, Docs.
      - "Product Models"
        - Concise abstractions expressed in terms of a "Feature Model".
    - Outputs
      - The instances of products.
  - Focus is on "Core Asset" development only.
    - Teams organized around the assets.
    - Don't need to add a new team each time you add a new product.
    - Changes of assets followed by auto re-instantiation of all products.

Organizational Models (cont)

- Diagram of "SPL Configurator". [e2]



Organizational Models (cont)

- "SPL Configurator" (cont) [e2]
  - "Transitions"
    - Transition from custom product ("product-centric") practices to SPL approaches.
    - Goal is "Minimal Disruption" during "Ongoing Production Schedules".
  - "Minimally Invasive Transitions"
    - Changes in SPL "surgical".
    - Minimal degrees of disruption and negative side effects to "the patient".
    - "Incremental Transitions"
      - Profit of previous incremental steps reinvested to "fuel" next steps.
    - "Reuse" of existing resources.

Organizational Models (cont)

- "SPL Configurator" (cont) [e2]
  - "Bounded Combinatorics"
    - SPL complexity reduces number of variants, tests to less than infinity.
    - Bounds also due to limits of "Reuse", "Refactoring" and "Scalability".
  - "Feature Profiles"
    - Each valid variant of SPL is demarcated as separate "Feature Profile".
  - "Modularity, Encapsulation and Aspects"
    - Hide complexity by encapsulating features.
    - Partition "Feature Model" into smaller models.
    - Localize each feature within smallest scope that it needs to influence.
  - "Composition and Hierarchy"
    - Tree-structure hierarchy representing:
      - "Compositions" of components at higher nodes.
      - "Components" at leaf nodes.
    - "Feature Model" is partitioned up among the nodes.

Decision Model

- Variability within a Software Product Line (SPL) is structured as a set of decisions to be resolved. [1]
- "Interrelations" and "Dependencies" between Products are captured. [1]
- "A [decision] model is generic", meaning that the model can be "instantiated" into a state where the set of decisions are resolved. [1]
- Each product (generic instance) connected to "open decision" in the model. [1]
- "Design Rules" [2]
  - Semantic constraints that govern legal compositions.
  - Selection of a "feature" enables or disables other "features".
  - Validates the compositions of "features".

Domain Models

- Used to create and instantiate generic SPL Products or "Assets". [1]
- Variabilities and composition rules reside at the architecture level. [1]
- Abstract way to specify building blocks of programs [2]
  - Algebra where constants and functions define a domain of programs
    - Constants represent programs.
    - Functions represent both a feature and its implementation.
    - Different functions with different implementations of the same feature
      - "k1(x)" // adds feature k, implementation 1, to program x.
      - "k2(x)" // adds feature k, implementation 2, to program x.
  - "Equation Optimization"
    - Determine which implementation (of feature "k") is best.
- "GenVoca" technique [2]
  - Equation representations scale naturally, become "Product Families" or SPL
    - A multiple-featured application is an "equation"
    - "app = i(j(f))" // application has features "i", "j" and "f"

Types of Features

- "Feature Types" that apply to 2 or more features.
  - "Mutually Inclusive"
    - 2 or more features <u>must be</u> included together in the SPL. [1]
  - "Mutually Exclusive"
    - 2 or more features <u>must not be</u> included together in the SPL. [1]
  - "Mutually Indifferent"
    - 2 or more features <u>may or may not</u> coexist within the SPL.
- "Feature Types" that apply to individual features.
  - "Mandatory"
    - The feature <u>is</u> required. [1]
  - "Optional"
    - The feature is <u>not</u> required. [1]
  - "Alternative"
    - The feature <u>may be substituted</u> by another alternative feature. [1]

Definition of "Traditional" Variability in SPLs

- "SPL" variability is directly related to the variability in its "features". [1]
- A feature spreads across many source files and modules. [1]
- "[Variability is the] ability of a software system or artifact to be changed, customized or configured for use in a particular context. A high degree of variability allows the use of software in a broader range of context, i.e.: the software is more reusable". [e5]
- "Complexity" [e5]
  - The amount of variability in a system.
- "1:N" mapping
  - 1 feature maps to many source files or modules. [1]
- Compare with "Aspect Oriented Programming" (AOP), "N:M" mapping
  - Many features map to many source files or modules. [1]

Types of Variability

- "Positive Variability"
  - Adds functionality. [1]
- "Negative Variability"
  - Removes functionality. [1]
- "Optional"
  - Code is "included" (i.e.: C++ #include preprocessor directive). [1]
- "Alternative"
  - Code is replaced / swapped. [1]
- "Function"
  - Functionality of the feature changes. [1]
- "Platform" / "Environment"
  - Platform or environment changes. [1]

Primary Parameters of Variability

- "Variation Points" (Vp)
  - The locations where changes in the pre-existing code occur. [1]

- Modules of Variation
  - Key: separate into distinct "Constant" and "Variable" modules. [1]

  - Module Examples
    - Interfaces
    - Classes
    - Components
    - Functions
    - Aspects

- "Dependencies" between modules
  - Interrelationships and constraints between modules.

  - Should be made clear
    - Otherwise, maintenance, upgrading and scalability suffer. [1]

Secondary Parameters of Variability: Traceability

- Document and track life of a "concept" throughout system development.
  - The extent that a "tailored" entity (module, aspect, etc.) can be traced efficiently to a "Decision Model". [1]
  - Goal is to reduce the gap between specification and implementation, so that requirements are easily traced and verified. [4]
  - Reveals better understanding of system variability. [e5]

- "Pre Traceability"
  - Describing the origin and evolution of a concept. [1]

- "Post Traceability"
  - Describing deployment and use of a concept. [1]

- "Cross-Reference Data"
  - Table cross-references "code artifacts" with architecture and design. [1]
  - Requires that all VPs are identified. [1]

Secondary Parameters of Variability: Scalability

- The impact of expanding the code. [1]
- The existing SPL evolves, grows, changes with time, in order to meet [1]
  - Market demand.
  - Customer expectations.
- "Degraded Implementation" at the "Variation Points" as SPL evolves possible
  - Potential Problem
    - Existing implementation does not scale well. [1]
  - Solution
    - Replace the existing SPL implementation "step-wise" (iteratively) at Variation Points ("VPs"). [1]

Secondary Parameters of Variability: Separation of Concerns (SOC)

- Separating the variant from the standard code in a way that changes for both can be made effectively. [1]
  - Split development process into “Domain” and “Application” processes. [e1]
  - Important “issues” should be represented in programs “intentionally” [1]
  - Separate the “Concerns” [e4]
    - Features
    - Components
    - Source Code
  - Separate “Concerns” that are “Common” and “Variable”. [e4]
    - Requirements: Features
    - Design: Components
    - Implementation: Source Code
- SOC Improves SPL and module Understandability, Adaptability, Reusability. [1]

Secondary Parameters of Variability: Others

- "Ease of Introduction"
  - Degree of restructuring needed in order to apply a particular technique on existing code. [1]
- "Tool Support"
  - Existence of tools that automate and facilitate variability management techniques. [1]
- "Language Support"
  - Existence of programming languages able to implement variability management techniques. [1]
- "Work Arounds"
  - Simulation of a technique using other techniques when current support not available. [1]
- "Design Fatigue"
  - Further evolution of the SPL application is difficult and costly. [4]

Approaches for Implementing Variability

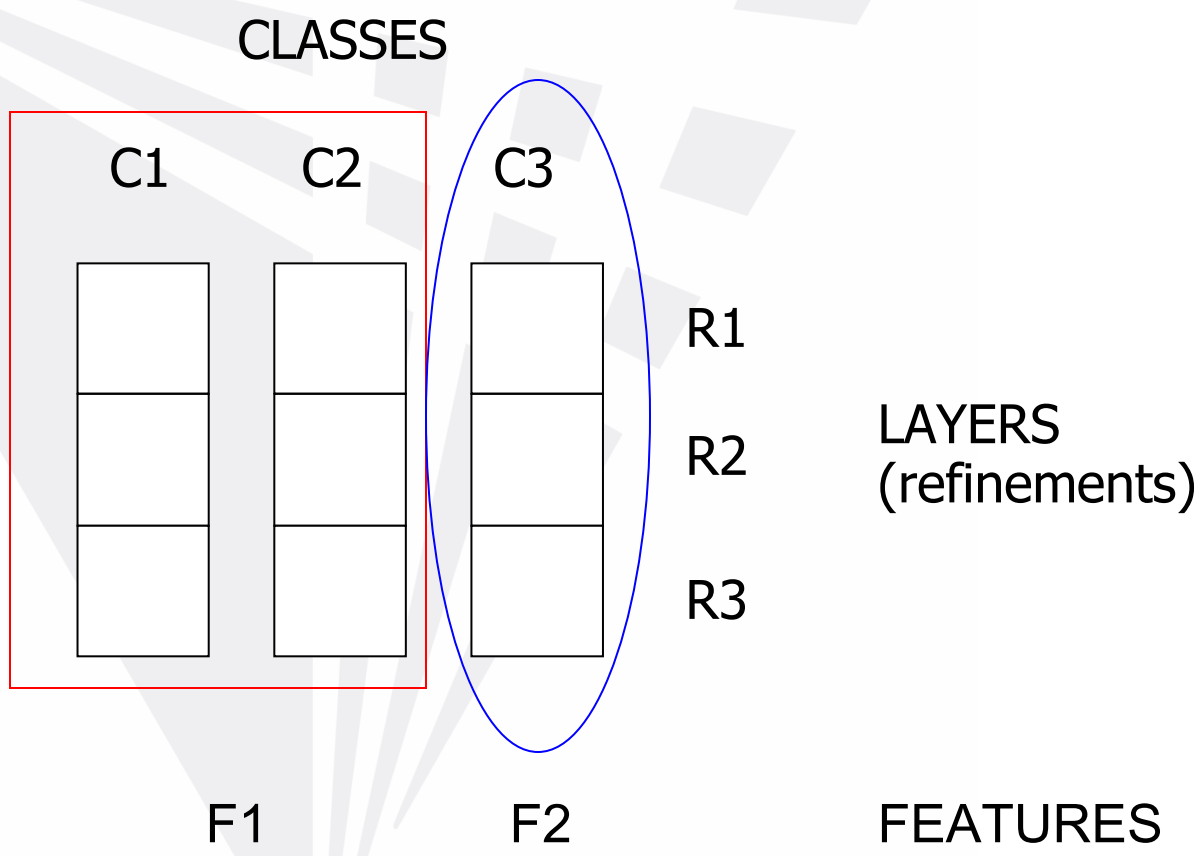
- Abstract Classes (low coupling at Vp) [1]
- Object Aggregation [1]
- Inheritance [1]
  - Standard, Virtual (late binding), Multiple, Mixin, Object, Parameterized
- Parameterization [1]
- Overloading [1]
- Overriding [1]
- Dynamically Linked Libraries (early binding) [1]
- Conditional Compilation [1]
  - C++ preprocessor directive statements (#include, #ifndef, #define, #endif)
- Reflection (not RTTI) [1]
- Feature Oriented Programming (FOP) [1]
- Aspect Oriented Programming (AOP) [1]
- Subject Oriented Programming (SOP) [1]
- Design Patterns [1]
- Generative Programming [3]
- Domain Specific Languages (DSL) [4]

Programming Approaches

- "Feature Oriented Programming" (FOP) [2]
  - Specify a target program in terms of desired "features"
  - Synthesize an efficient program that meets these specifications
  - "GenVoca" (see Domain Models) and "Stepwise Refinement" (SWR)
    - Methodology for building software by progressively adding details [2]
    - Possible to implement using the Java's "Jakarta Tool Suite". [4]
  - "Feature Refinements"[2]
    - Applications "of significant complexity" are expressed by composing a few <u>large-scale refinements</u>
      - <u>Encapsulates</u> the implementation of a "feature".
      - More general than "packages"
        - Encapsulates sets of complete classes.
        - Encapsulates sets of fragments of classes.
      - Classes split up into "layers"
        - "Layers" are the same thing as "Refinements".

Programming Approaches (cont)

- Example diagram showing elements of FOP.

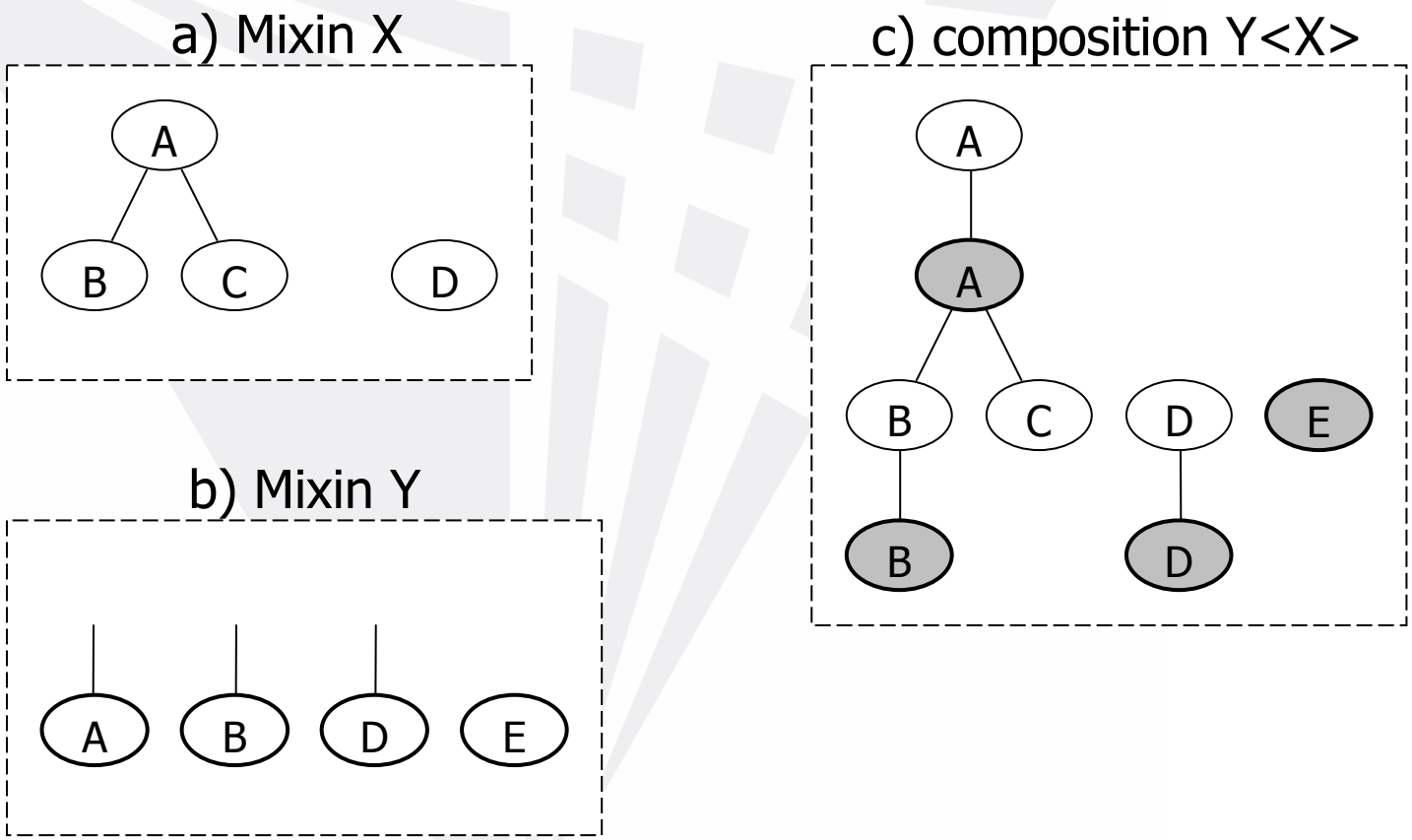


Programming Approaches (cont)

- "Feature Oriented Programming" (FOP) (cont) [4]
  - "Mixin Layers"
    - "Refinements" that are <u>inserted</u> into the source code.
    - "Mixin X"
      - <u>Encapsulates</u> many classes that have inheritance structure.
      - Any type or number of relationships can exist between the classes.
    - "Mixin Y"
      - "Refinement classes"
      - Subclasses ("Refinement" of Mixin X superclasses) and New classes.
    - "Composition Y<X>"
      - Original class at top, "Refinement Class" plugged in underneath.
        - "Refinement Class" has same name as superclass.
      - Only the <u>bottom-most</u> classes are <u>instantiated</u> and <u>subclassed</u>.
      - Lower classes contain all of the "features" or "aspects" from the higher classes.

Programming Approaches (cont)

- Example diagram showing “Mixin Layers”. [4]



Programming Approaches (cont)

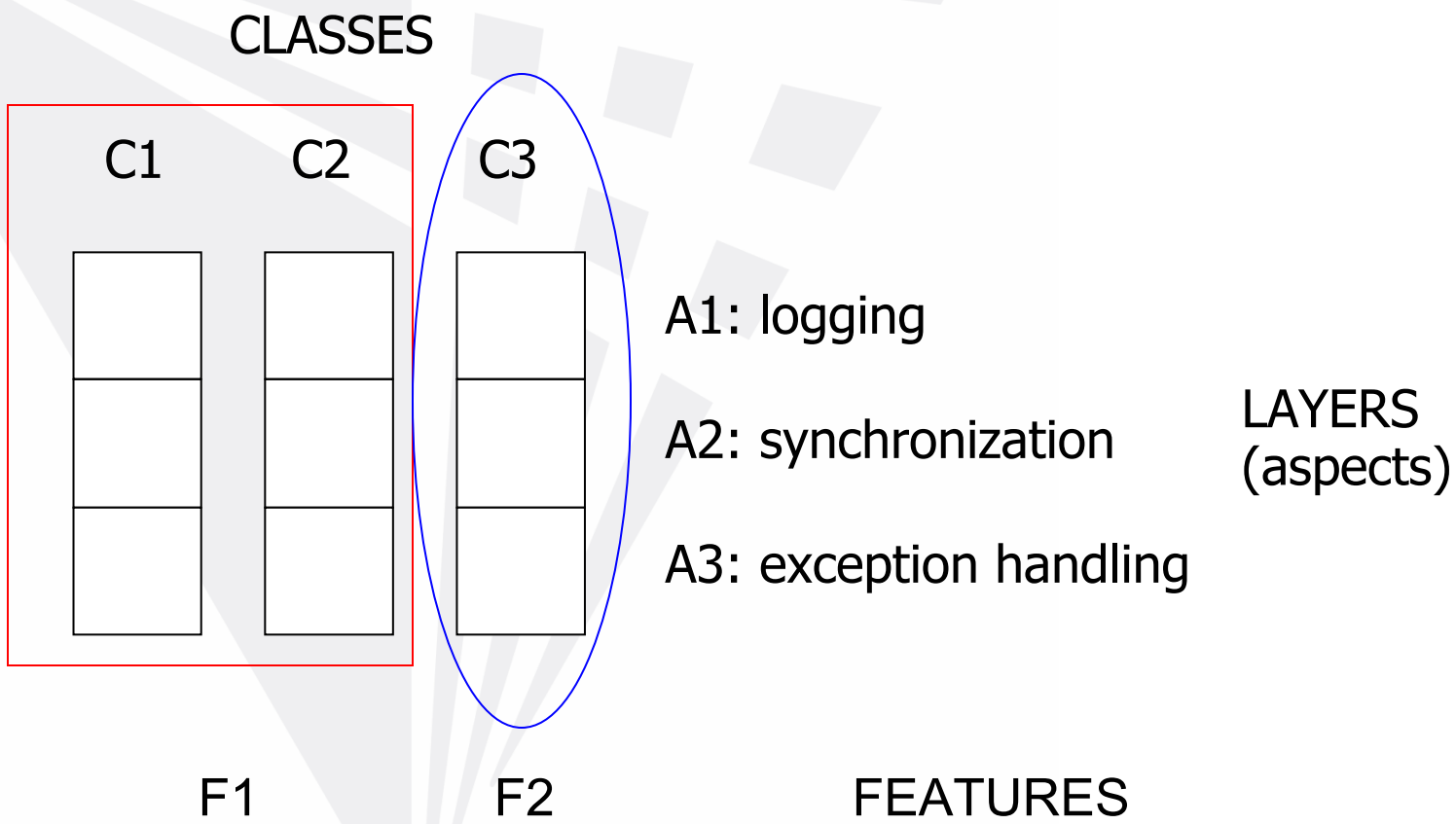
- "Aspect Oriented Programming" (AOP) [2]
  - Closely related to FOP
    - Both deal with modules <u>encapsulating</u> "cross-cuts" of multiple classes
    - Both expressed as <u>large-scale refinements</u>
  - Different from FOP [2]
    - FOP uses traditional OOP techniques, does not involve "Join Points"
  - AOP can be implemented using Java's "AspectJ" [2]
    - "Join Points"
      - Method calls, Exception Handlers, Variable References.
    - "Aspect Files" [e4]
      - Contains "Optional" / "Alternative" code, separated from main "kernel".
    - "Point-Cut"
      - Defines a set of "Join Points" where "advice" code is inserted
        - before, after, or around code.

Programming Approaches (cont)

- "Aspect Oriented Programming" (AOP) (cont) [2]
  - "Aspects"
    - aka "Cross-cutting Concerns"
      - "Mandatory" features implemented in a standard way.
      - "Optional" or "Alternative" features encapsulated in "Aspects".
    - Made up of 1 or more "Features"
    - Found at 1 or more "Join Points"
  - "Join Points"
    - "Aspects" in "Aspect File" integrated into main "kernel" at JPs. [e4]
    - Locations in code affected by 1 or more "Aspects" (N:M mapping)
  - "Weaving"
    - Integration process where Aspect effects code at 1 or more Join Points.
    - Inserting Aspect code into existing source code.
  - Examples of Use
    - Logging
    - Synchronization
    - Exception Handling

Programming Approaches (cont)

- Example diagram showing elements of AOP.



Programming Approaches (cont)

- "Subject Oriented Programming" (SOP)[1]
  - Very similar to to AOP
    - Except SOP only allows <u>operations</u> (functions) for use as "Join Points".
    - SOP calls the collections of classes and class fragments "Subjects".
  - "Aspect Decomposition"[1]
    - Separates subject-specific code pieces from each other.

Programming Approaches (cont)

- "Generative Programming" [3]
  - Engineering families of systems (SPLs) using "Generators".
  - "Generators" [3]
    - Automate the <u>assembly</u> of implementation components.
      - Based on "Configuration Knowledge" or declarative statements.
    - Programmer states desired end-product "in abstract terms and the generator produces the desired system or component".

Programming Approaches (cont)

- "Domain Specific Languages" (DSL) [4]
  - Applications or systems that have been customized based on their points of variability can be specified compactly in terms of "Domain Concepts".
    - "Extensibility": the property that a simple change of the design requires a proportionally simple change of the source code.
    - Applications <u>extensible</u> through the evolution of DSL specifications.
    - Component-level code is substantially simplified.
  - Compiling DSL
    - Specialized compilers translate DSL into source code.
  - SPL evolution
    - Changes to the SPL involve modifying the DSL specifications.
    - Reduces complexity of defining and refining system logic.

Domain Engineering

- "Domain Engineering" ("engineering-for-reuse".) [3]
  - "Domain" is the "system family" or SPL.
  - "Domain Analysis" [3]
    - "Domain Scoping"
      - "Determines which features belong to the system".
      - Identify "product lines".
    - "Feature Modeling"
      - Identify "Common" and "Variable" features.
      - Identify "Dependencies" between features.
  - "Domain Design " [3]
    - Develop a common "Product Line Architecture" (PLA) for the SPL.
  - "Domain Implementation" [3]
    - "Implement the components, generators and the reuse infrastructure".

Configuration Knowledge

- "Configuration Knowledge" [3]
  - "Manufacturing" / Domain Programmer level.
  - Used in "Generators".
  - Mapping
    - Between the "Problem Space" and "Solution Space".
    - Abstract <u>requirements</u> onto appropriate <u>configurations</u> of components.
    - "Takes a specification of a system or component and returns the finished system or component".
  - Contents
    - Feature combinations that are illegal.
    - Default settings and "Dependencies" of features.
    - Rules for combining features.
    - "Optimizations".

Problem Space

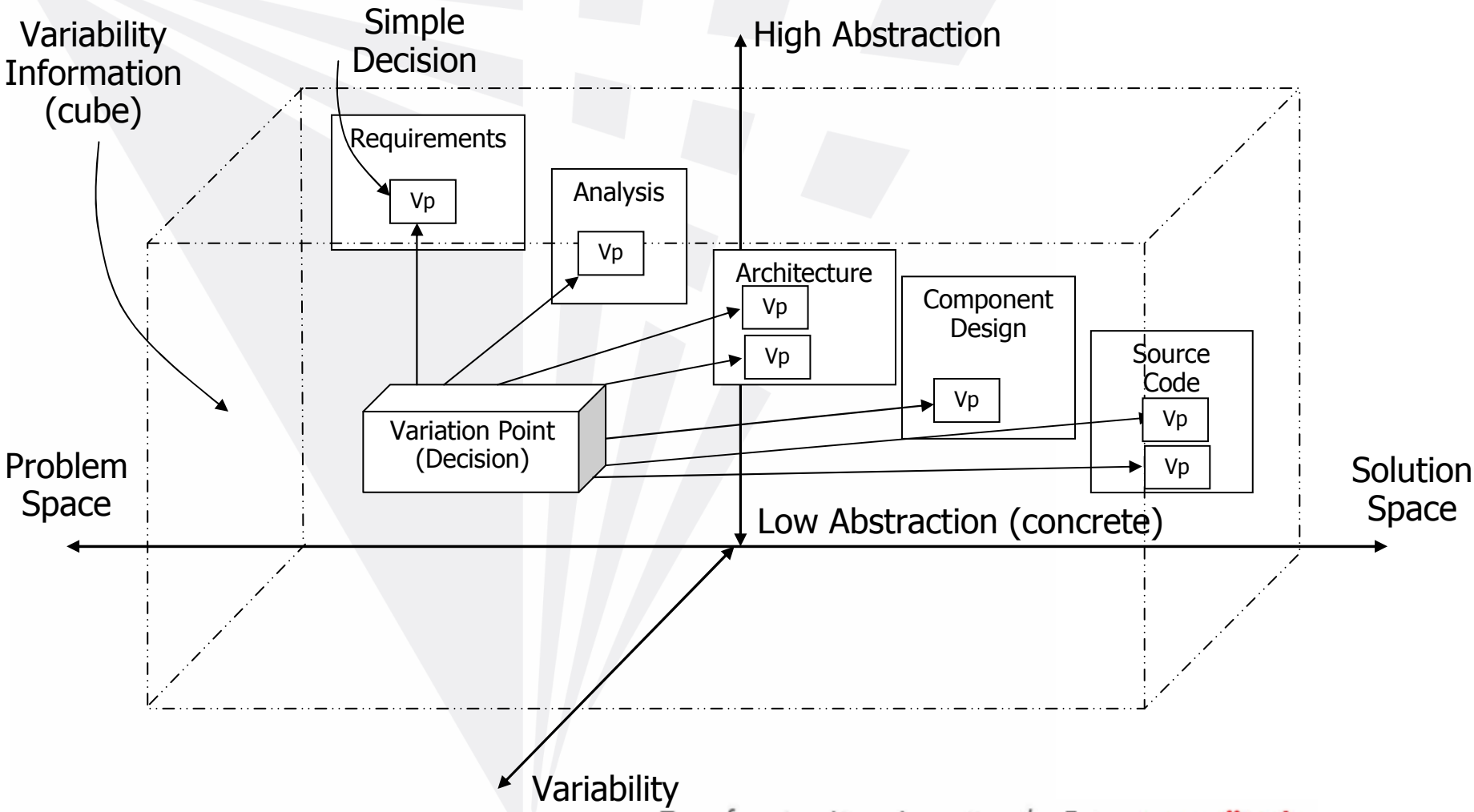
- "Problem Space" [3]
  - "Customer" / Application Programmer level.
  - "Feature Descriptions" and concepts are unique to specific SPLs.
  - "There is no particular component that makes a [particular system], but it is rather a particular combination of carefully selected [components]".
  - Difficult to make small changes to requirements, without generating a large amount of work to implement the changes (go from problem to solution). [6]

Solution Space

- "Solution Space" [3]
  - "Customer" / Application Programmer level.
  - "Consists of implementation components with all their possible combinations"
    - Goals in the "Solution Space": maximize "customizability" including
      - Maximize <u>combinability</u>.
      - Minimize <u>redundancy</u>.
      - Maximize <u>reuse</u>.
      - Focus on using <u>elementary components</u>.

3 Dimensions of SPL Diagram

- Diagram of “3 Dimensions of SPL”, containing both “artifacts” and “VPs”. [e5]

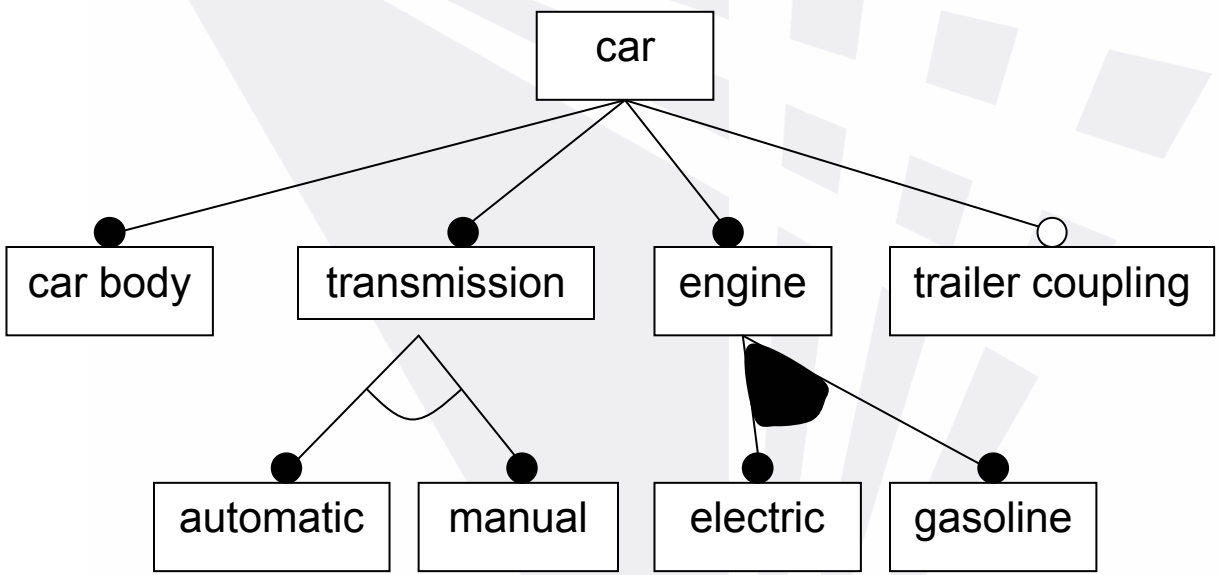


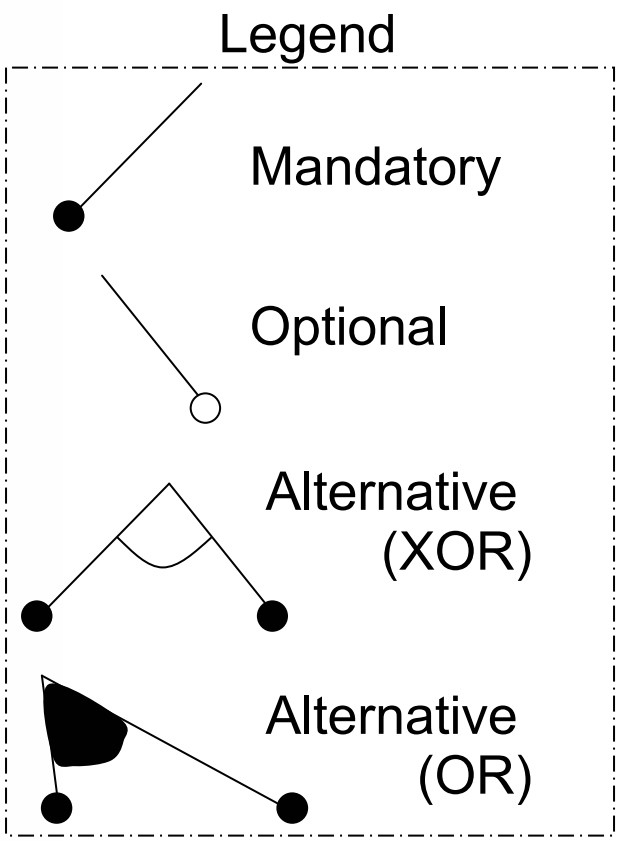
Feature Model

- "Feature Model" [3]
  - "Feature Diagram".
    - Root node is the product.
    - Remaining nodes are features.
    - "Reveals the kind of variability contained in the design space".
    - "Allows us to model variability without having to commit to a particular implementation mechanism".
  - Constraints.
  - Feature Descriptions.
  - Feature "Binding Times"
    - Pre-compile-time, Compile-time, Link-time, Run-time, Update-time.

Feature Diagram

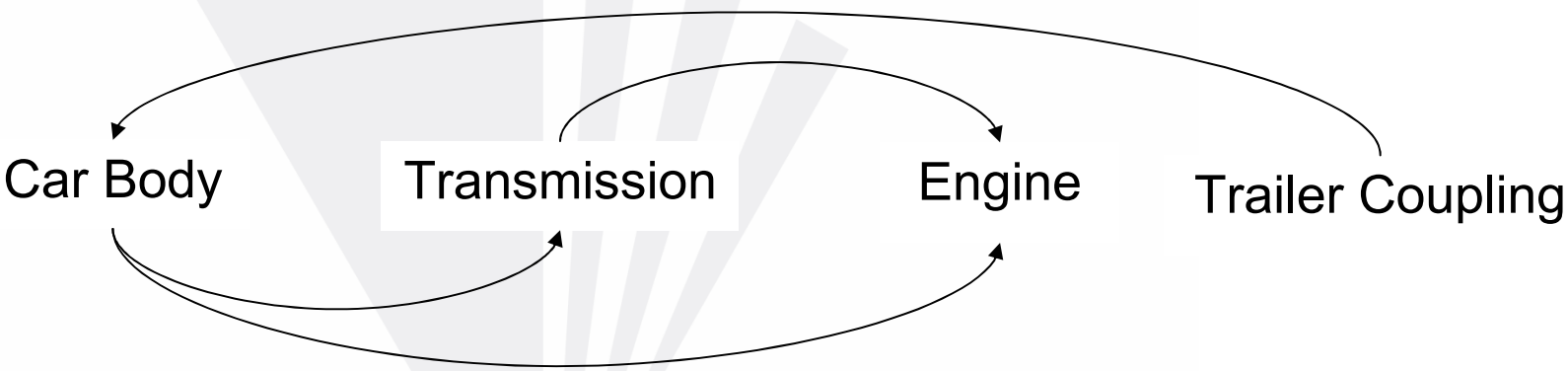
- "Feature Diagram": A hierarchy of SPL features, within the "Feature Model". [3]





Dependencies

- "Uses" Dependencies [3]
  - Dependencies between features, where one feature "uses" another.
    - "Car Body uses Transmission and Engine"
    - "Transmission uses Engine"

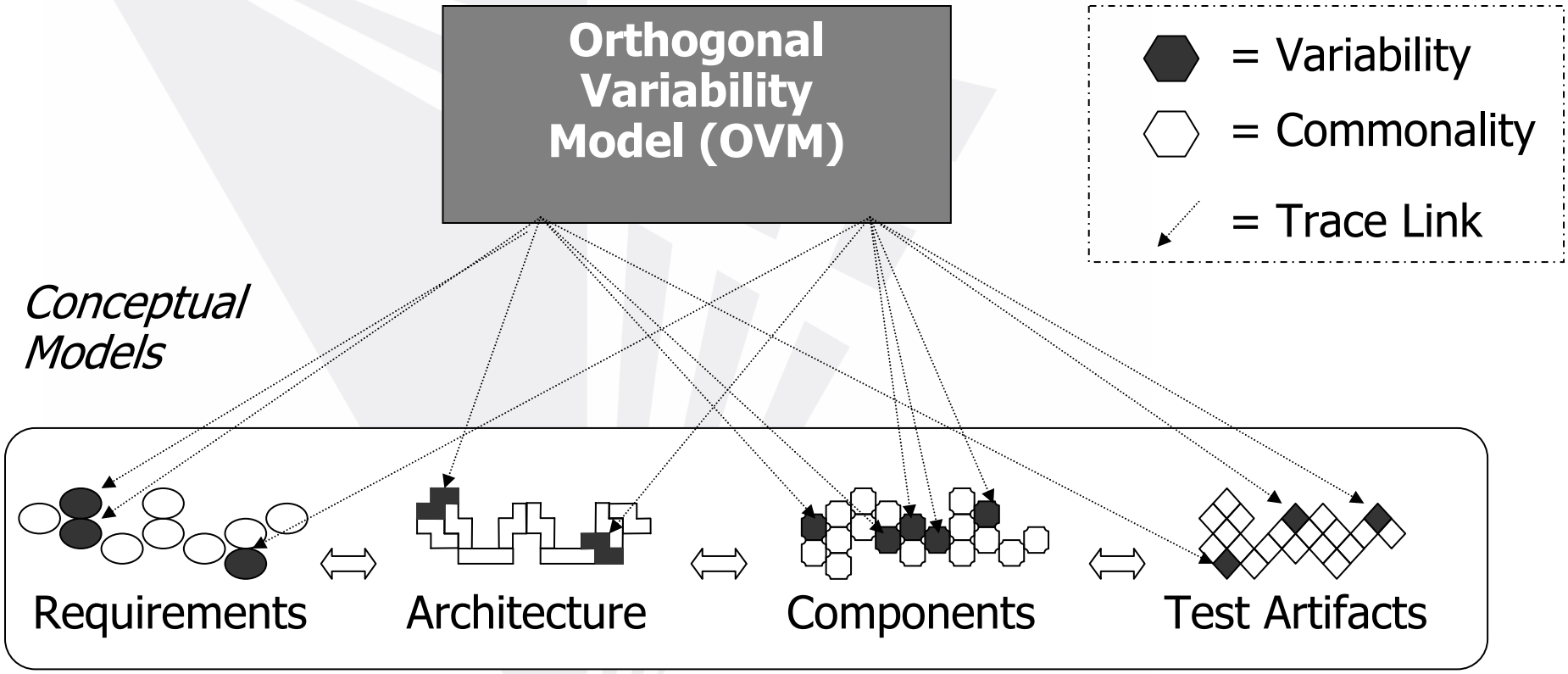


Orthogonal Variability Model (OVM)

- "Orthogonal Variability Model" (OVM) [e1]
  - Advantages over the "Feature Model".
    - Variability defined separately from existing models (i.e. UML).
      - Only variable features recorded, avoids common features.
      - Avoiding feature overhead makes it smaller, less complex.
      - Centralized definition of variability aids "Traceability".
    - OVM elements traceable to traditional model elements via "links".
    - Stakeholders don't need knowledge of modeling language (i.e. UML).
  - OVM Documentation.
    - "Variation Points" (VPs).
      - Variable items (components, etc), and variable properties of items.
    - "Variants".
      - Instantiations of SPLs, specific configurations of variation points.
    - Constraints.
      - Dependencies between variation points.
      - Project Management decisions about offering certain "Variants".
    - Visibility.
      - Audience of OVM (internal, external).

Orthogonal Variability Model (OVM) Diagram

- Diagram of the "Orthogonal Variability Model". [e1]



SEI's "Framework for Software Product Line Practice"

- SEI's "Framework for Software Product Line Practice" [e3]
  - "Product Line Practice Patterns"
    - 29 practices exist for successfully fielding a SPL.
    - Practice areas combined and coordinated to achieve useful outcomes.
    - Compare with "Design Patterns"
      - "Design Pattern" organizes software elements.
      - "Product Line Practice Pattern" organizes organization activities.
  - "Adoption Factory"
    - One of the "Product Line Practice Patterns".
    - "Roadmap" of organizational activities to achieve SPL development.
      - Manager can track the progression of development activities.
    - "Sub-Patterns" work in coordination to bring about SPL products.
    - "Phases" indicate the 3 general stages of the project.
    - Compare with "State Machine" approach
      - State Machine
        - Only 1 state at a time can be active; <u>series</u> actions.
      - Sub-Pattern
        - "Position" merely indicates "greater emphasis"; <u>parallel</u> actions.

SEI's "Framework for Software Product Line Practice" (cont)

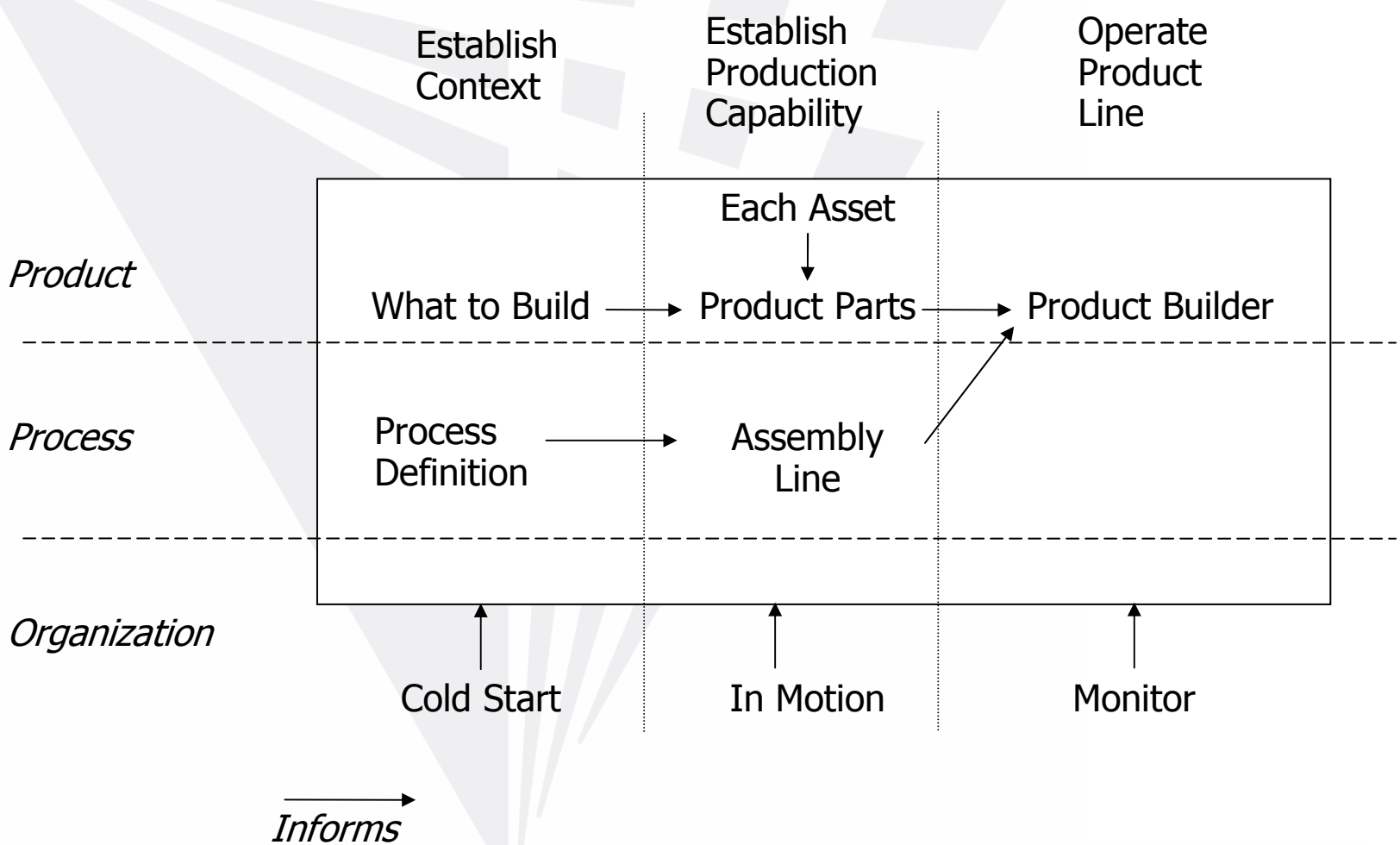
- SEI's "Framework for Software Product Line Practice" (cont) [e3]
  - "Adoption Factory" (cont)
    - Sub-Patterns
      - <u>Product</u>
        - "What to build"
        - "Each Asset"
        - "Product Parts"
        - "Product Builder"
      - <u>Process</u>
        - "Process Definition"
        - "Assembly Line"
      - <u>Organization</u>: indicates "organizational status"
        - "Cold Start": PL has not been initiated.
        - "In Motion": PL initiated, activities in progress.
        - "Monitor": PL deployed, optimization needed.

SEI's "Framework for Software Product Line Practice" (cont)

- SEI's "Framework for Software Product Line Practice" (cont) [e3]
  - "Adoption Factory" (cont)
    - Phases
      - "Establish Context"
        - Launch the project.
        - Draft initial plans.
        - Recover architecture and "Core Assets" from existing SPL.
        - Build a "Business Case": Goals, Predictions, Risks, Scope.
        - Risk Management.
        - Assign tasks, form teams.
      - "Production Capability"
        - Create "Production Plan" defining process for making product.
        - "Reusable Assets" developed for use in assembly.
        - "Requirements Engineering" identifies "Variation Points".
        - "Architecture Design" integrates new PL with existing SPLs.
      - "Operating the Product Line"
        - Resolves requirements.
        - Assembles products.
        - Performs PL integration.

SEI's "Framework for Software Product Line Practice" (cont)

- Diagram of the "Adoption Factory Pattern". [e3]

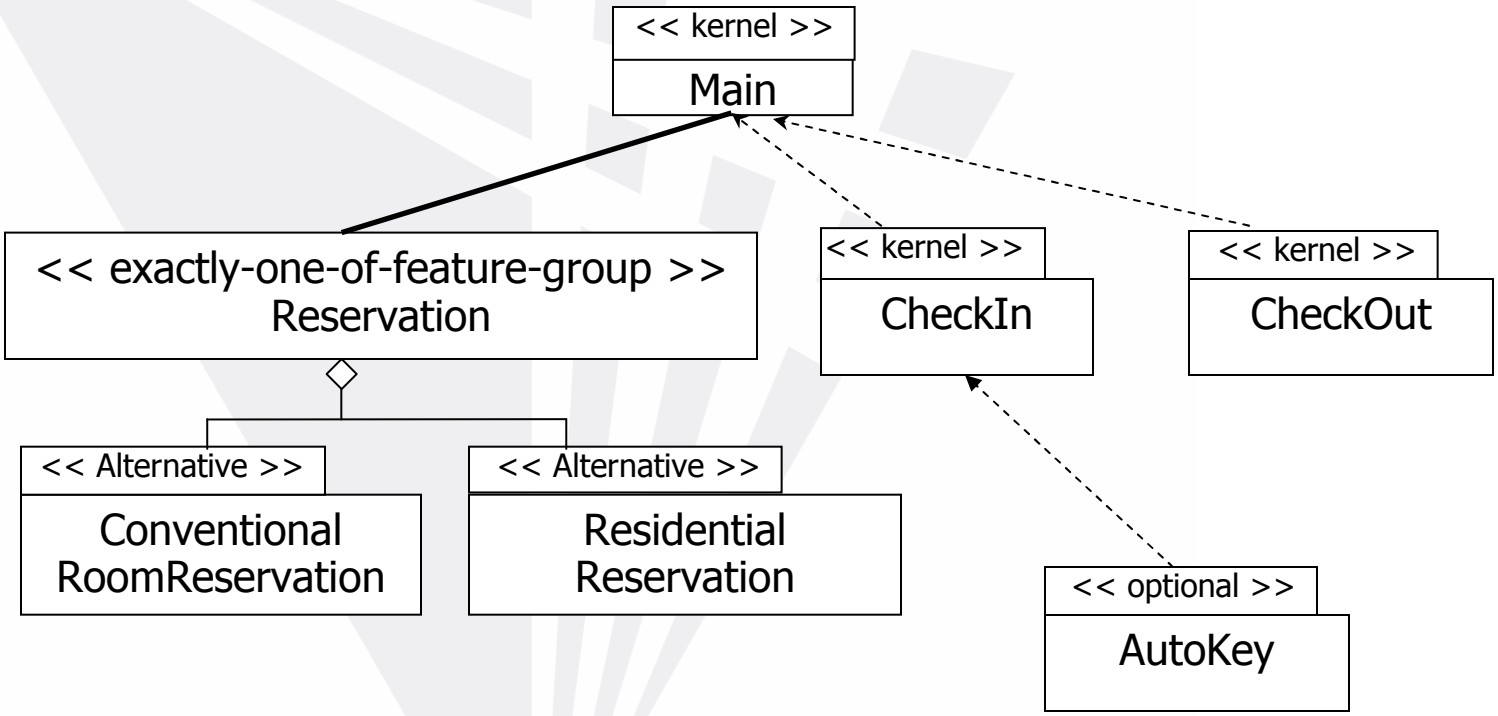


Multiple View Model

- "Multiple View Model" [e4]
  - "Considers the product line from multiple perspectives"; shotgun approach.
  - Defines common and variable characteristics of SPL using UML notation.
    - "Use Cases"
      - Used for "single systems" to find "functional features".
      - Complements "Feature Models".
    - "Feature Models"
      - <u>Main</u> sub-model
      - Primary "driver" for SOC and managing SPLs.
      - "Feature Analysis"
        - Emphasis on variable features: <u>optional</u> and <u>alternative</u>
    - Others Models, including UML
      - User Interface (UI) Navigation
      - Object Interaction.
      - Components Interfaces.
      - Activity Diagrams.
      - Software Architecture Diagrams.
      - Entity Relationship Diagrams.

Multiple View Model (cont)

- Diagram of the “Multiple View Modeling - Feature Model”. [e4]



Multiple View Model (cont)

- "Feature Description Language" (FDL) [e4]
  - Simple programming language describing the link between "kernel" code and code supplied to "Insertion Points" (Variation Points).
    - "$START ins1"
      - Inline statement where variation point code is inserted.
    - Other statements
      - Start / End a "Feature"
        - $FEATURE [X]
        - $ENDFEATURE [X]
      - Start / End selection from multiple implementations of the feature
        - $FEATUREINTERACTION [C, D]
        - $ENDFEATYREINTERACTION [C, D]
      - "if, elseif, endif" feature selection conditional statements
        - $IF FEATURE [C, D]
        - $ELSEIF FEATURE [C]
        - $ENDIF

Multiple View Model (cont)

- Diagram of “FDL” with “Kernel” and “Variable” source code. [e4]

Kernel Source Code

```
class … ()
{
.
.
.
$START ins1
.
.
.
$START ins2
.
.
.
$START ins3
.
.
.
}
```

Variable Source Code File

```
$FEATURE [A]      // Optional Feature

$START ins1
            // insertion code
$END ins1

$START ins2
            // insertion code
$END ins2

$ENDFEATURE [A]
```

```
$FEATURE [X]      // Alternative Feature

$START ins2
            // insertion code
$END ins2

$ENDFEATURE [A]
```

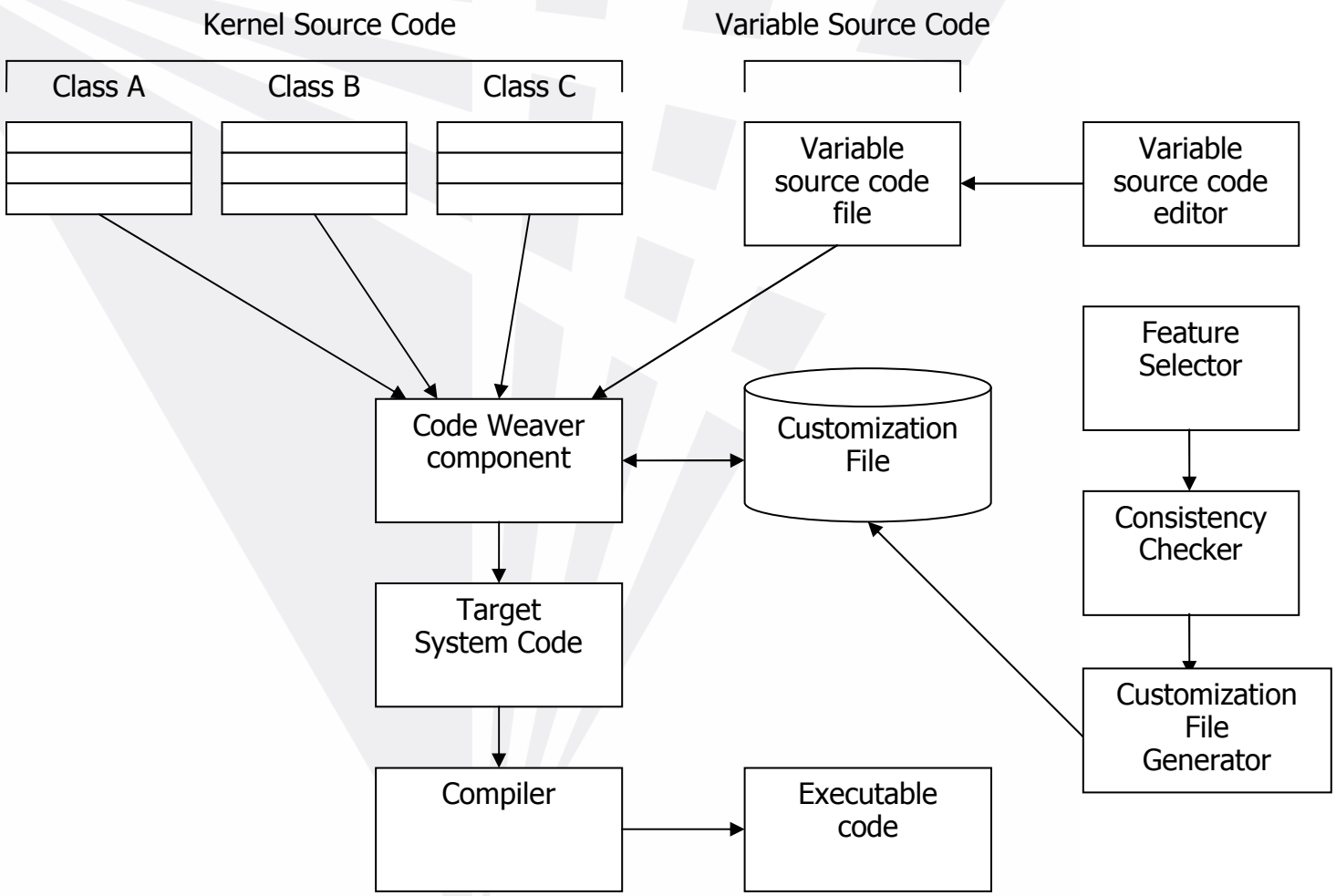
```
$FEATUREINTERACTION [C, D]

  $START ins3
    $IF FEATURE [C, D] // Both selected
      // insertion code
        $ELSEIF FEATURE [C] // C selected
           // insertion code
        $ELSEIF FEATURE [D] // D selected
           // insertion code
        $ENDIF
    $END ins3

$ENDFEATUREINTERACTION [C, D]
```

Multiple View Model (cont)

- "Feature Description Language" (FDL) (cont) [e4]
  - "Customization File".
    - Information regarding the features that are selected after the FDL statements are performed.
  - "Customization Process"
    - "Feature Selection"
      - Application Engineer selects desired features for "Target System".
    - "Generation of the Customization File"
      - Consistency check made to verify validity of selections.
      - Generates file.
    - "Code Weaving"
      - Reads the "Customization File" to get selected features.
      - Identifies selected features in the "Variable source code file".
      - Integrates "Variable source code file" into "Kernel source code file".

Multiple View Model (cont)

- Diagram of the "Customization Process". [e4]



References

- [1] "Implementing Product Line Variabilities", Anastasopoulos, Michalis, Gacek; Cristina, p 109 – 116, ACM Symposium on Software Reusability, 2001
- [2] "A Tutorial on Feature Oriented Programming and Product-Lines", Batory, Don, p 753 – 754, Proceedings of the 25th International Conference on Software Engineering, 2003
- [3] "Components and Generative Programming", Czarnecki, Krzysztof; Eisenecke, Ulrich, p 2 – 19, ACM SIGSOFT Software Engineering Notes, vol. 24, no. 6, 1999
- [4] "Achieving Extensibility Through Product-Lines and Domain-Specific Languages: A Case Study", Batory, Don; Johnson, Clay; MacDonald, Bob; Von Heeder, Dale; p 191 – 214, ACM Transactions on Software Engineering and Methodology, Vol. 11, No. 2, April 2002

References

- [5] "Software Product Lines: Organizational Alternatives", Bosch, Jan, p 91 – 100, Proceedings of the 23rd International Conference on Software Engineering, 2001
- [6] "Notes on the Second International Workshop on Development and Evolution of Software Architectures for Product Families", Clements, Paul; Weiderman, Nelson, p 39–43, Software Engineering Notes, vol. 23, no. 3, 1998
- [e1] "Variability Management in Software Product Line Engineering", Metzger, Andreas; Pohl, Klaus; p 1–2, 29th International Conference on Software Engineering (ICSE'07 Companion), IEEE, 2007
- [e2] "New Methods in Software Product Line Practice", Krueger, Charles, p 37–40, Communications of the ACM, December 2006, Vol. 49, No. 12
- [e3] "Getting There From Here A Roadmap For Software Product Line Adoption", Clements, Paul; Jones, Lawrence; McGregor, John; Northrop, Linda; p 33–36; Communications of the ACM, December 2006/Vol. 49, No. 12

References

- [e4] “Separation of Concerns in Software Product Line Engineering”, Gomaa, Hassan; Saleh, Mazen; p 1–5, ACM Workshop on Modeling and Analysis of Concerns in Software (MACS), 2005

- [e5] “Tracing Software Product Line Variability – From Problem to Solution Space”, Berg, Kathrin; Bishop, Judith; Muthig, Dirk; p 182-191, Proceedings of SAICSIT, 2005