Variability and Architecture

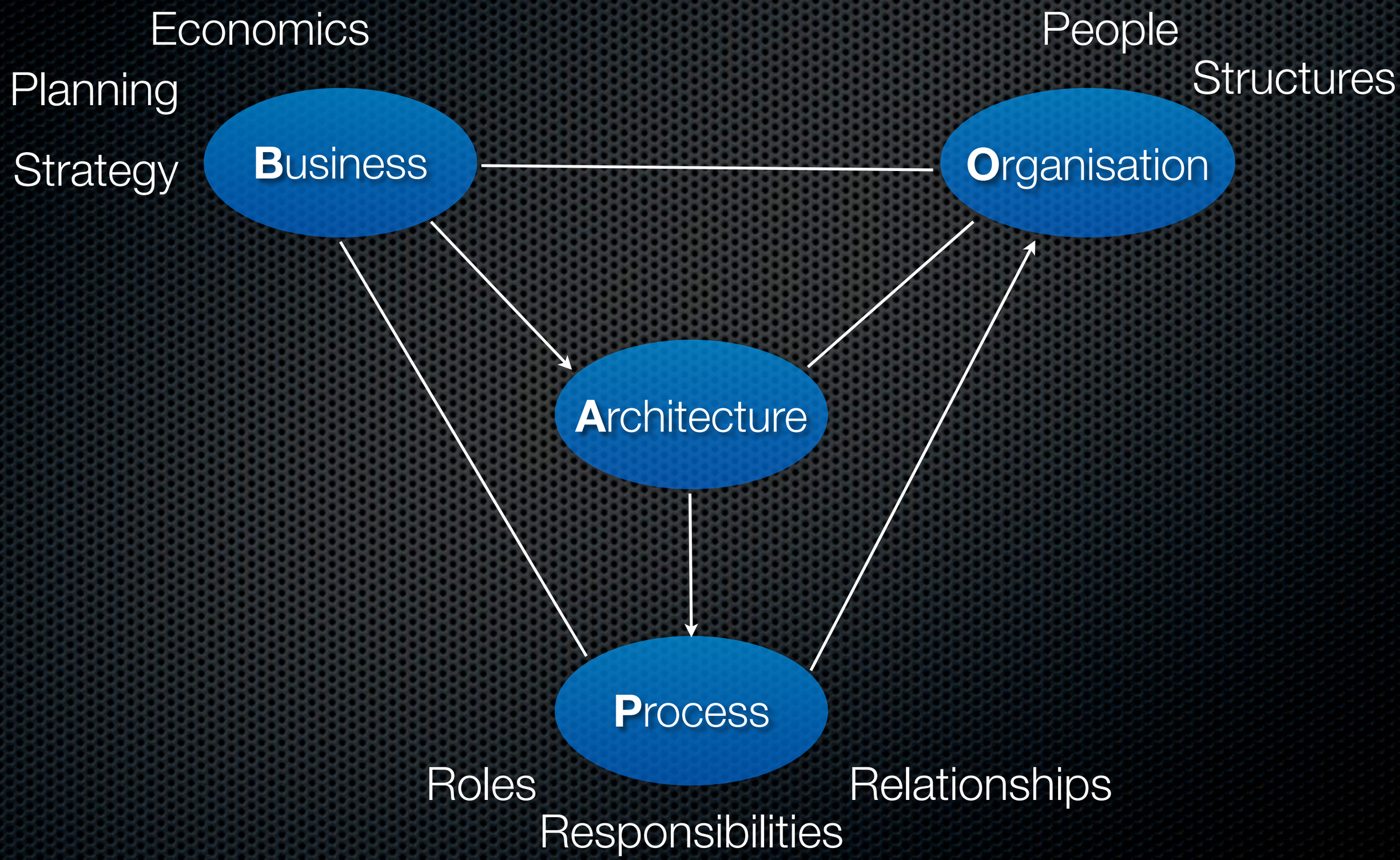
SPLE Course, DAT165, L2 & L3

Robert Feldt - robert.feldt@gmail.com

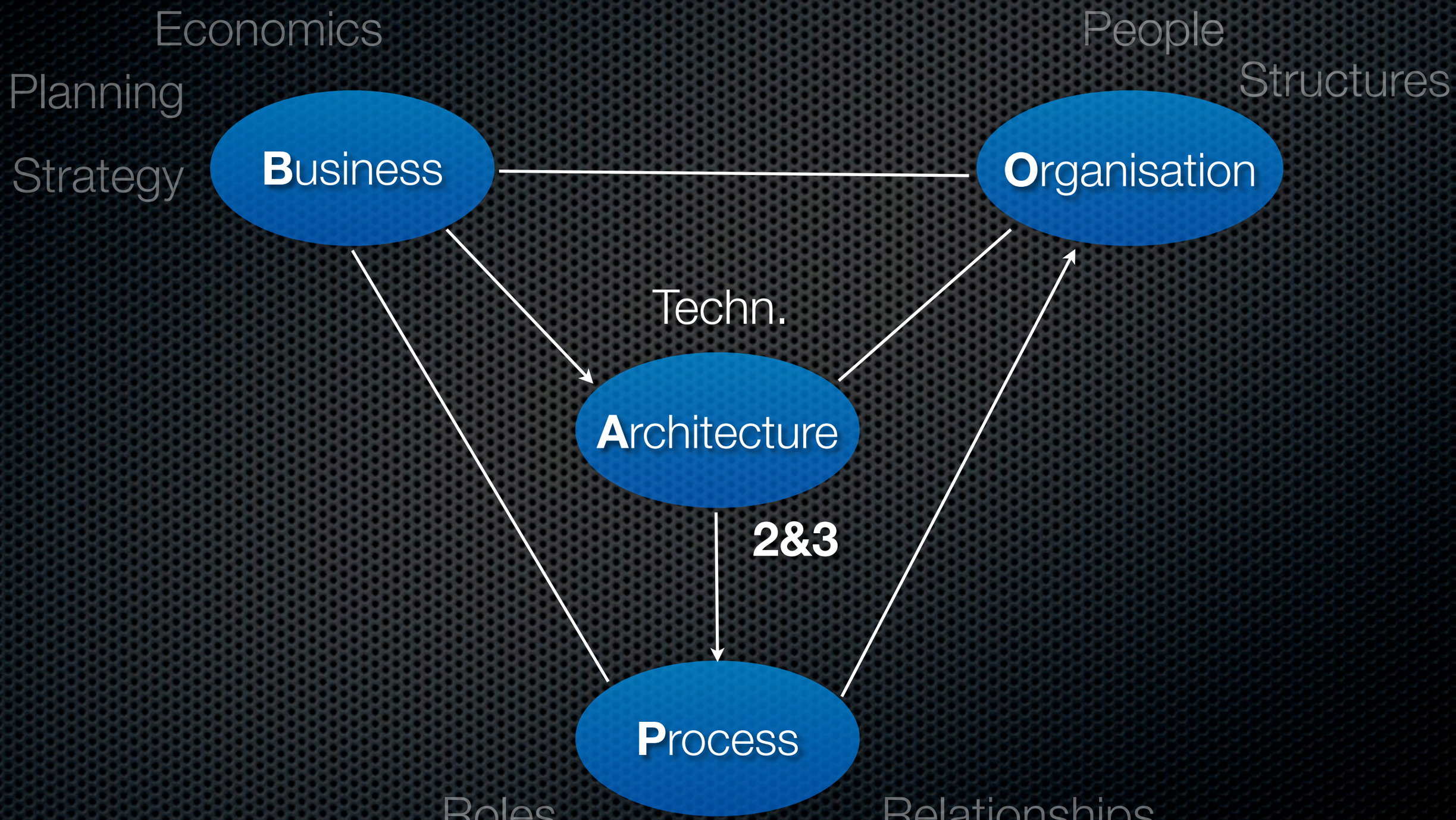
Acronyms used

- DE = Domain Engineering
- AE = Application Engineering
- RefArch = Reference Architecture
- TTM = Time To Market
- SW = Software
- SPL = Software Product Line
- SPLE = SPL Engineering (and course book!)
- Dev = Development

Lectures - Overview (BAPO Model)



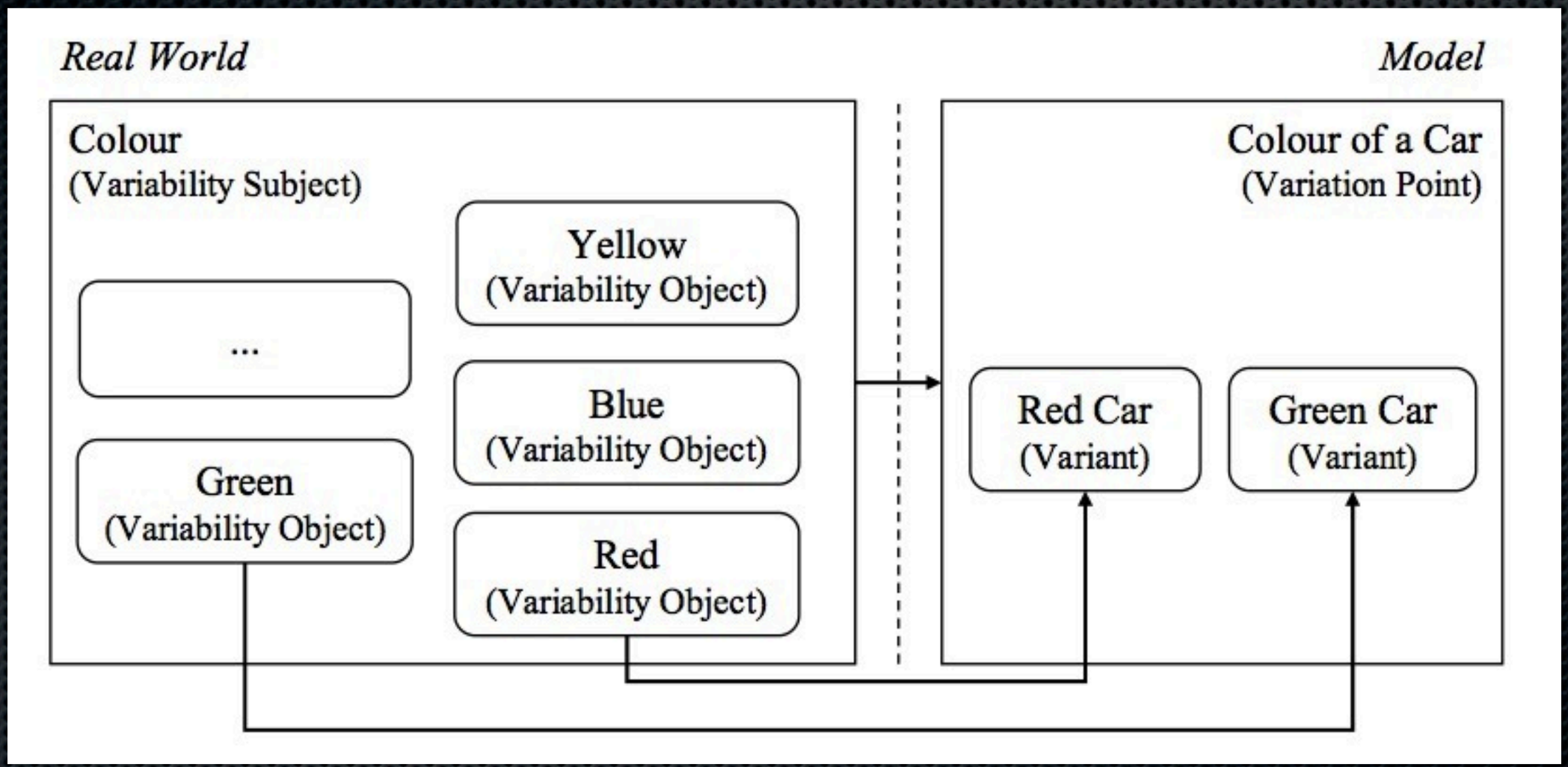
Lectures - Overview (BAPO Model)



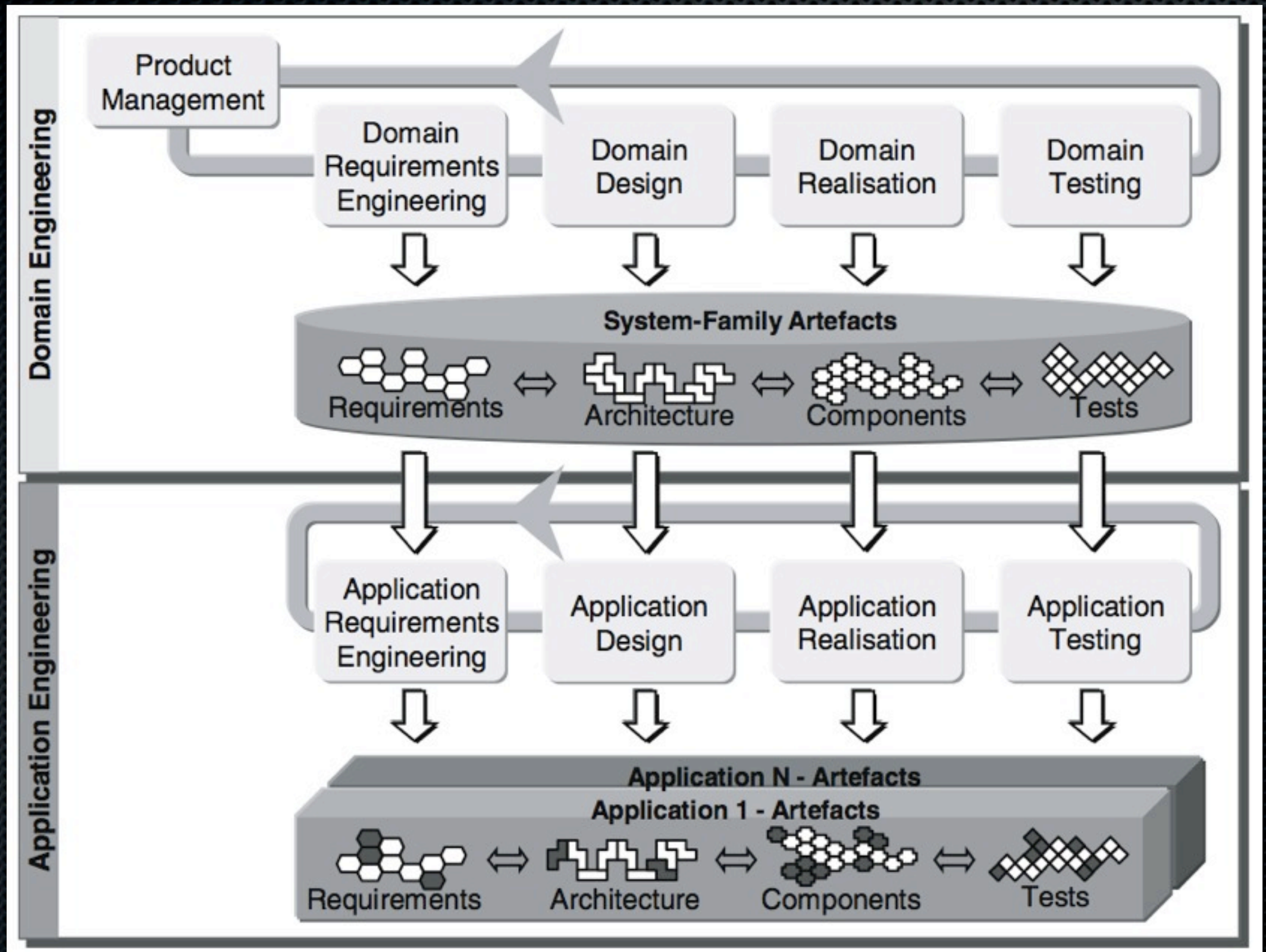
Definitions

- Variability subject - a var item of the real world
- Var object - particular instance of a subject
- Var point - represents a var subject + contextual info
- Variant - represents a var object
- For SPL, having 10 variation points with 3 possible variants, gives 3^{10} (59,049) configs

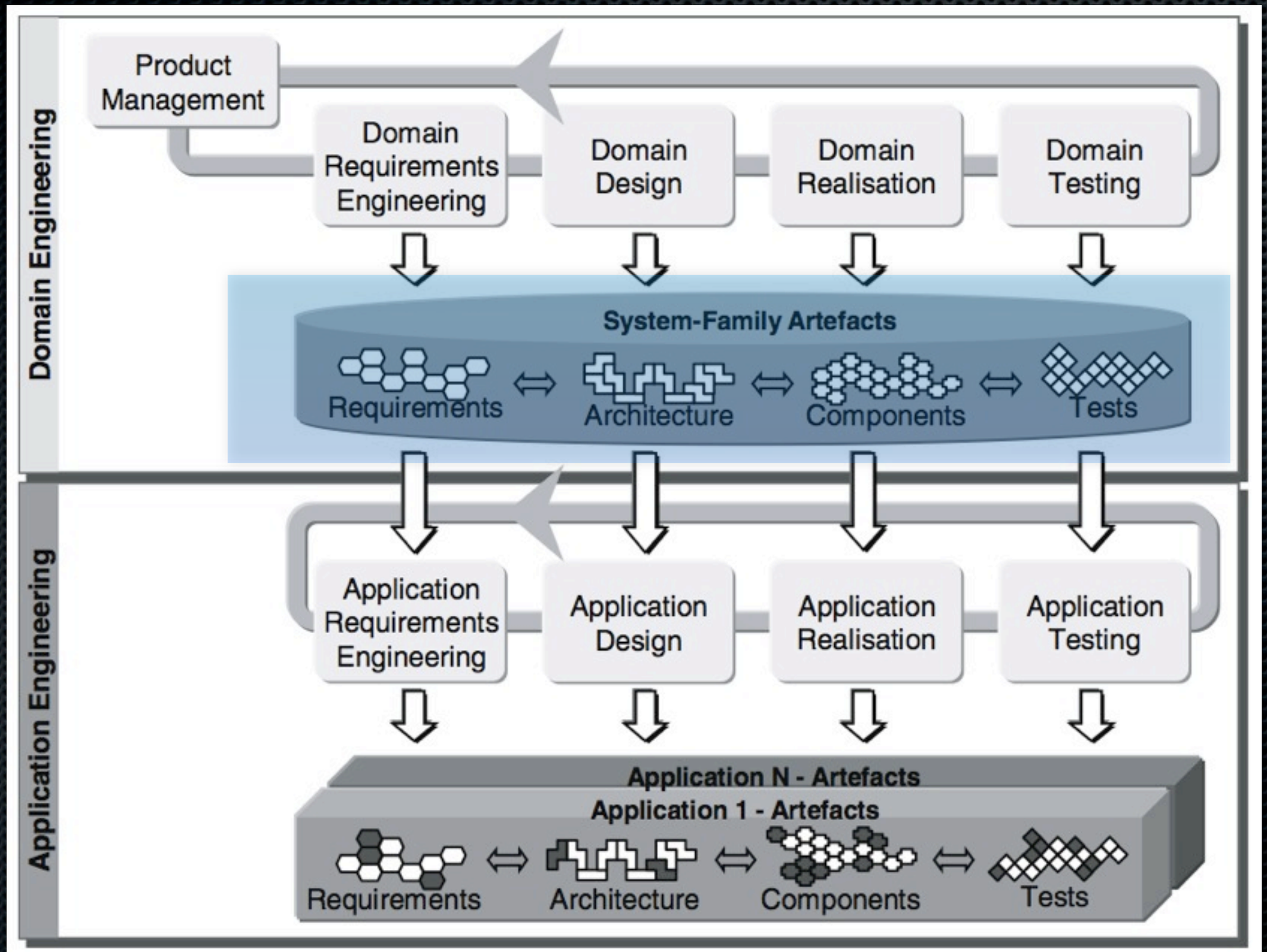
Definitions



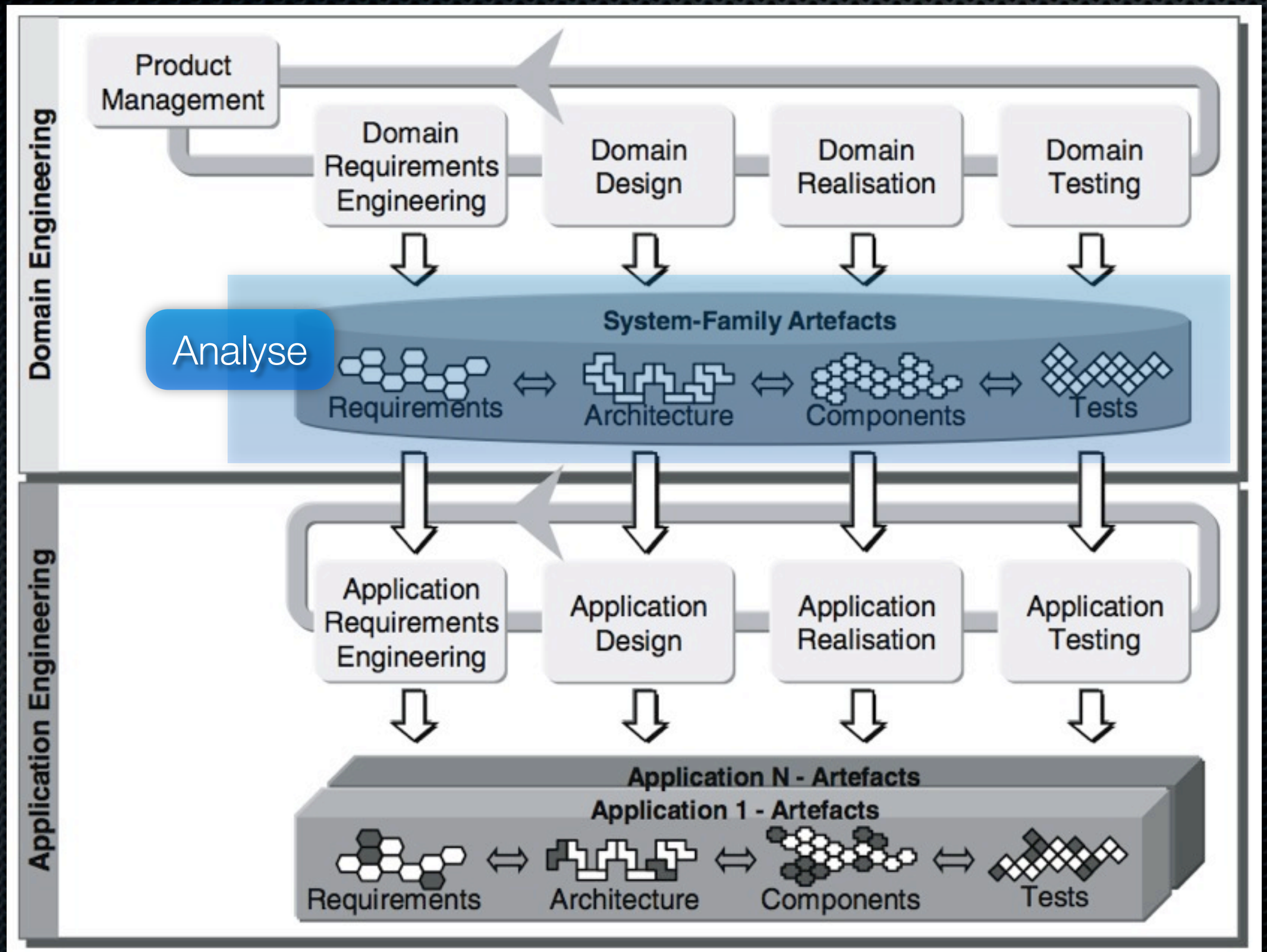
Domain and Application Engineering



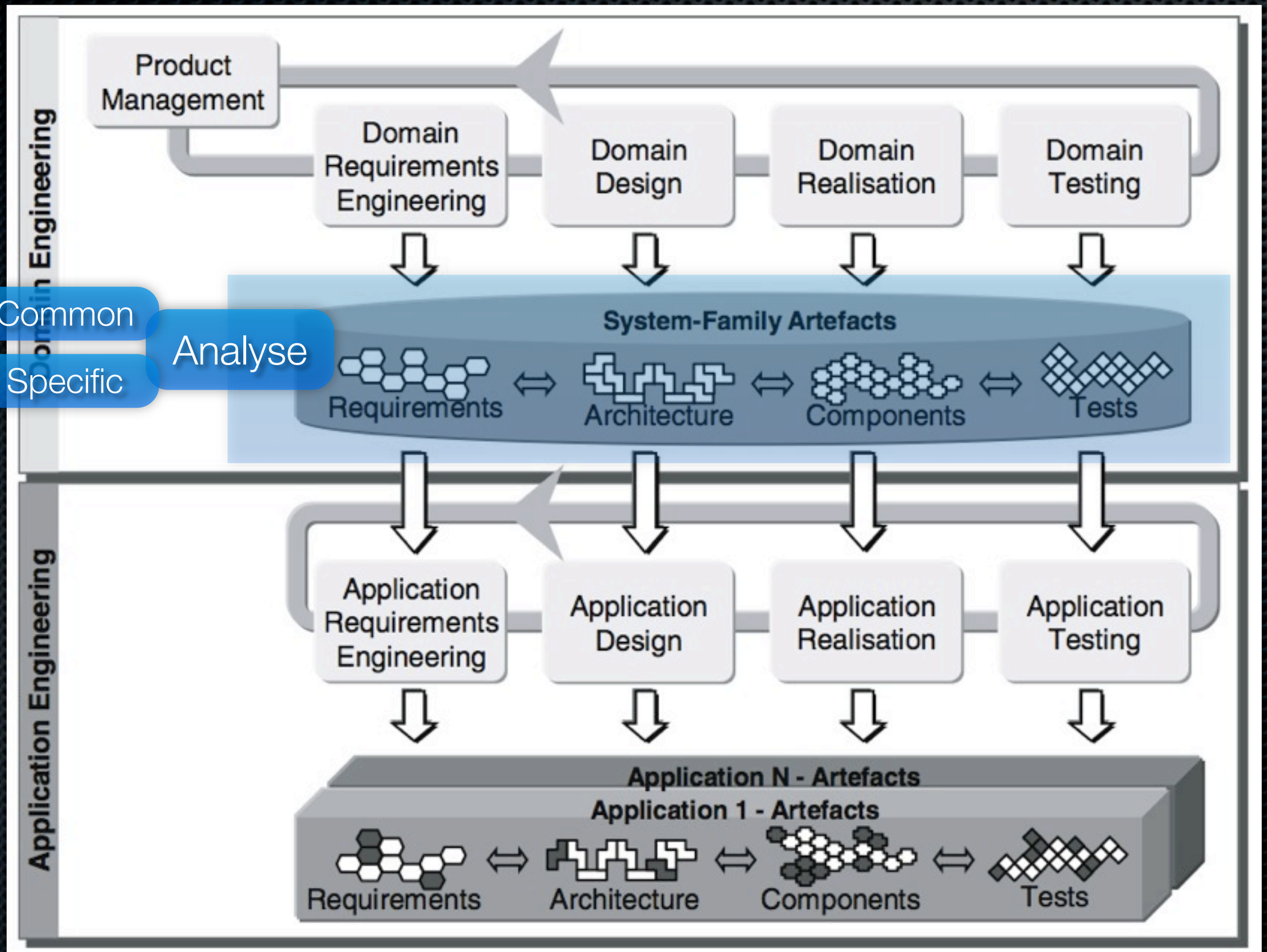
Domain and Application Engineering



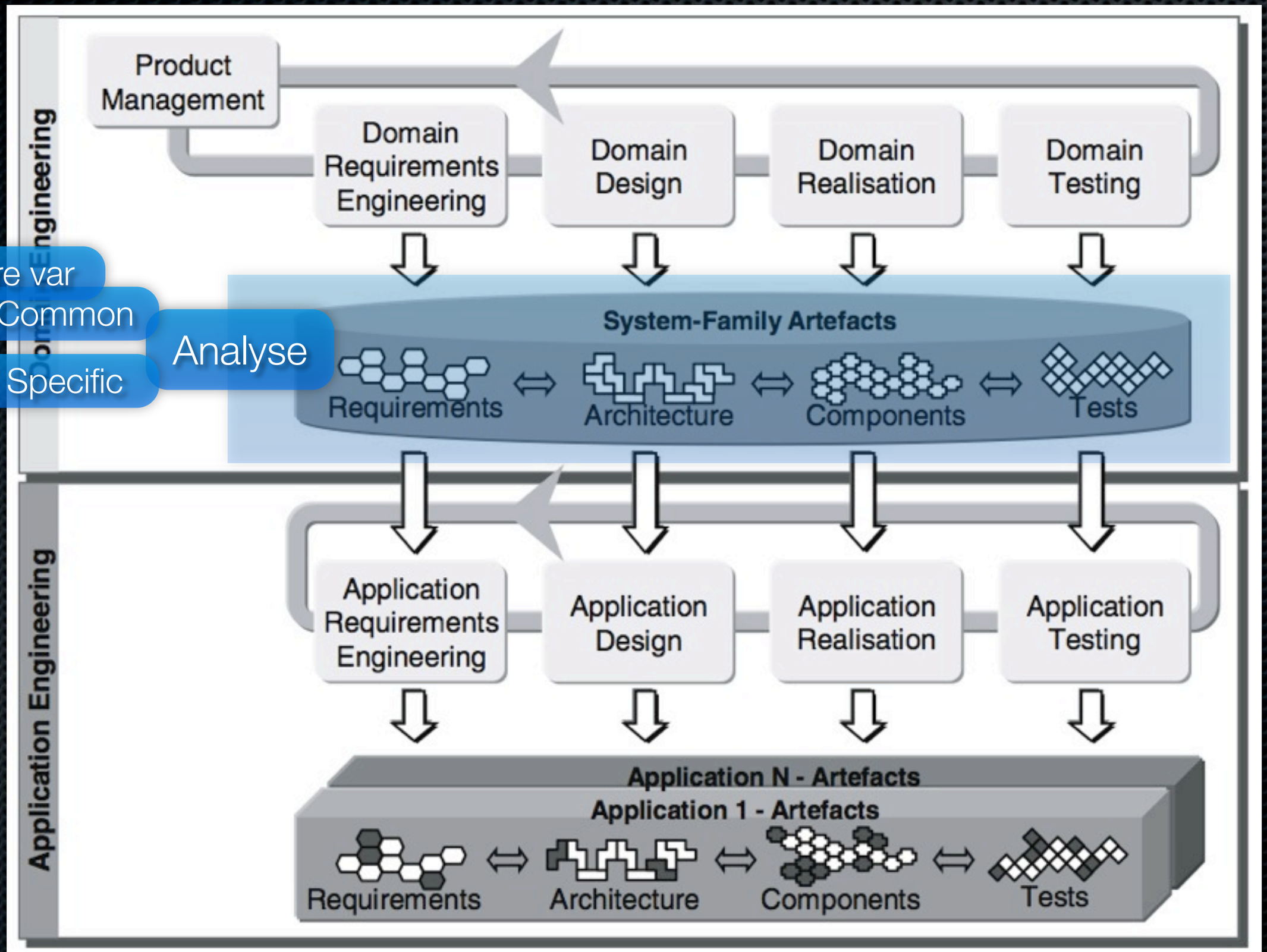
Domain and Application Engineering



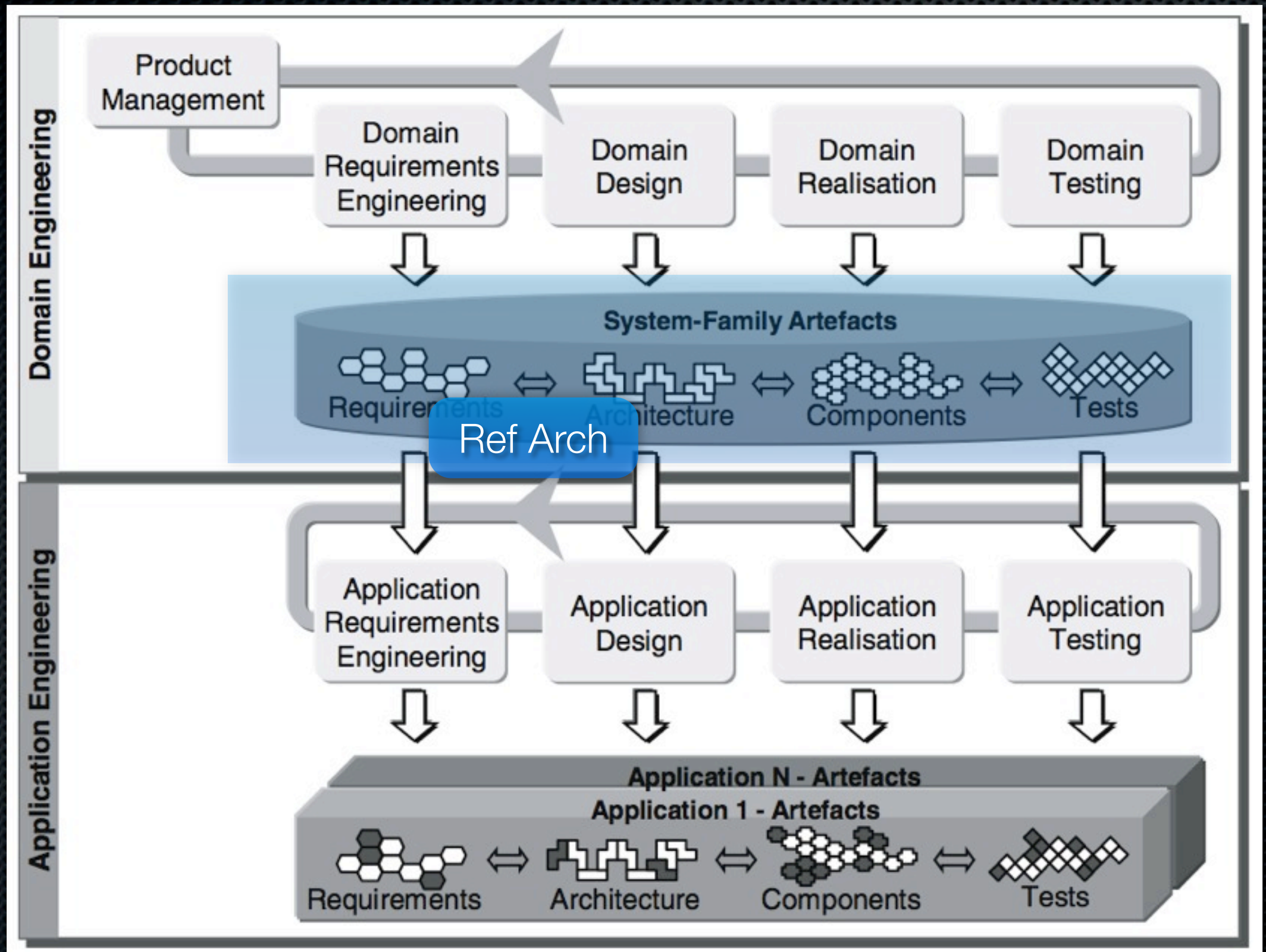
Domain and Application Engineering



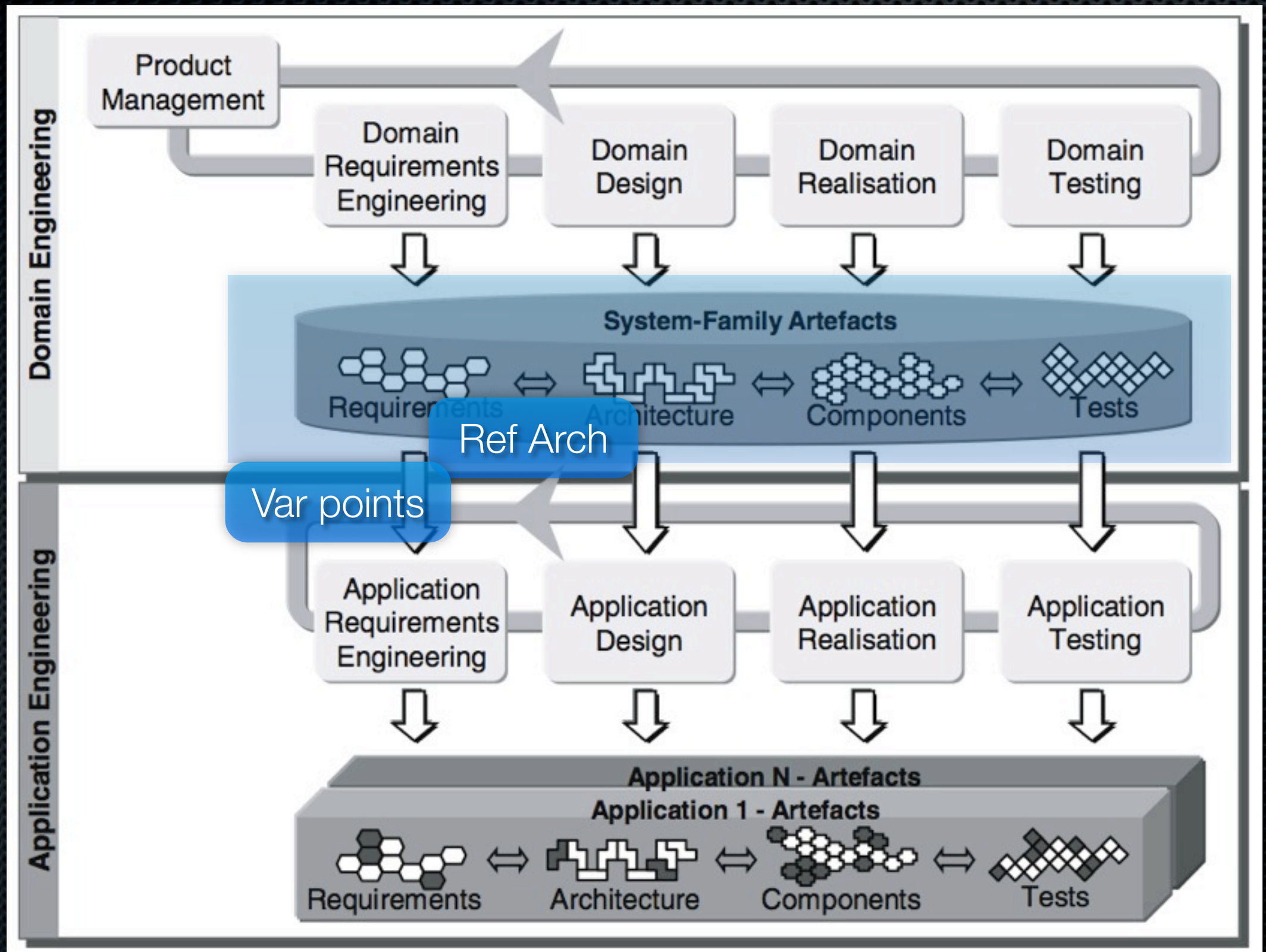
Domain and Application Engineering



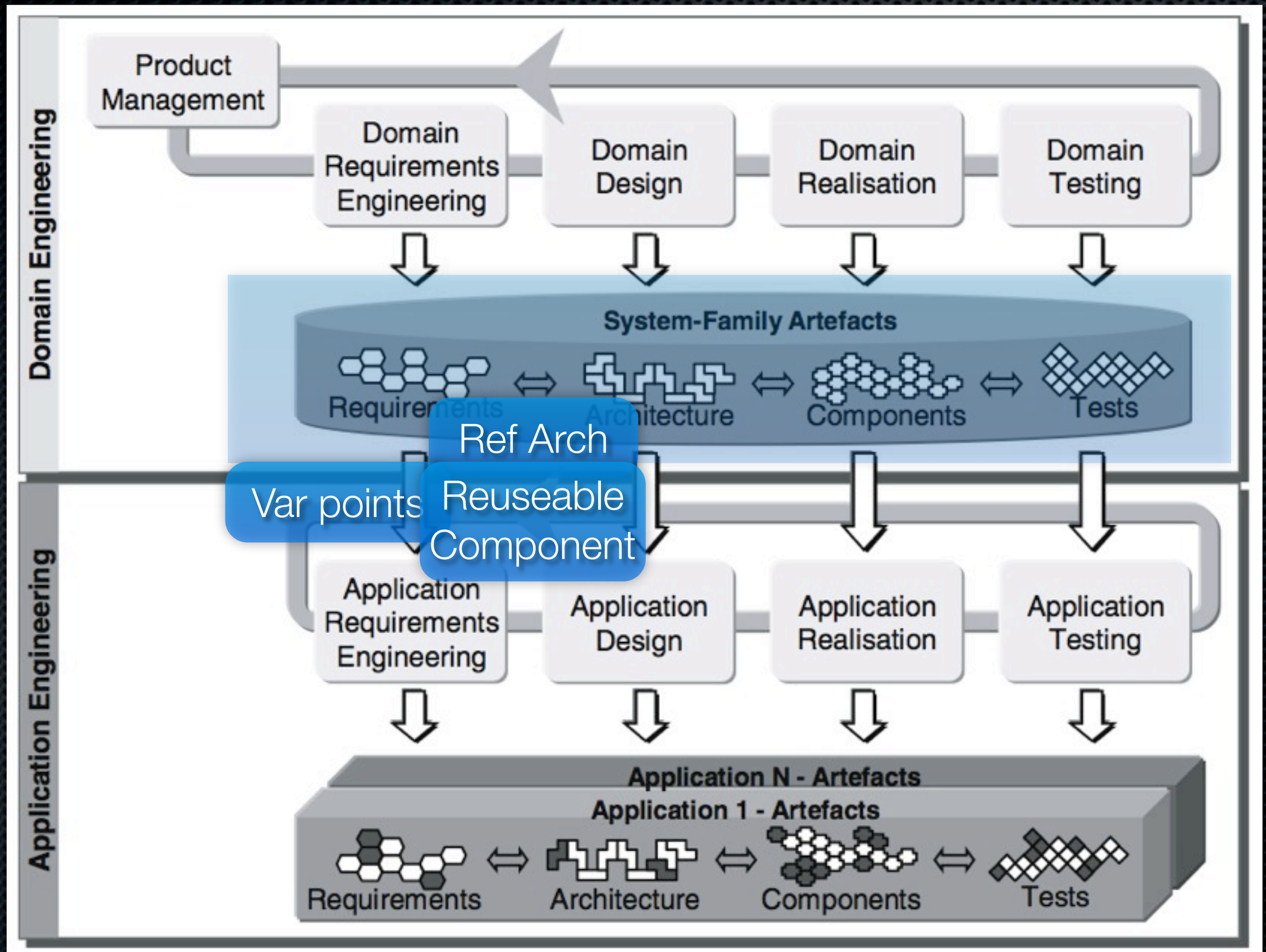
Domain and Application Engineering



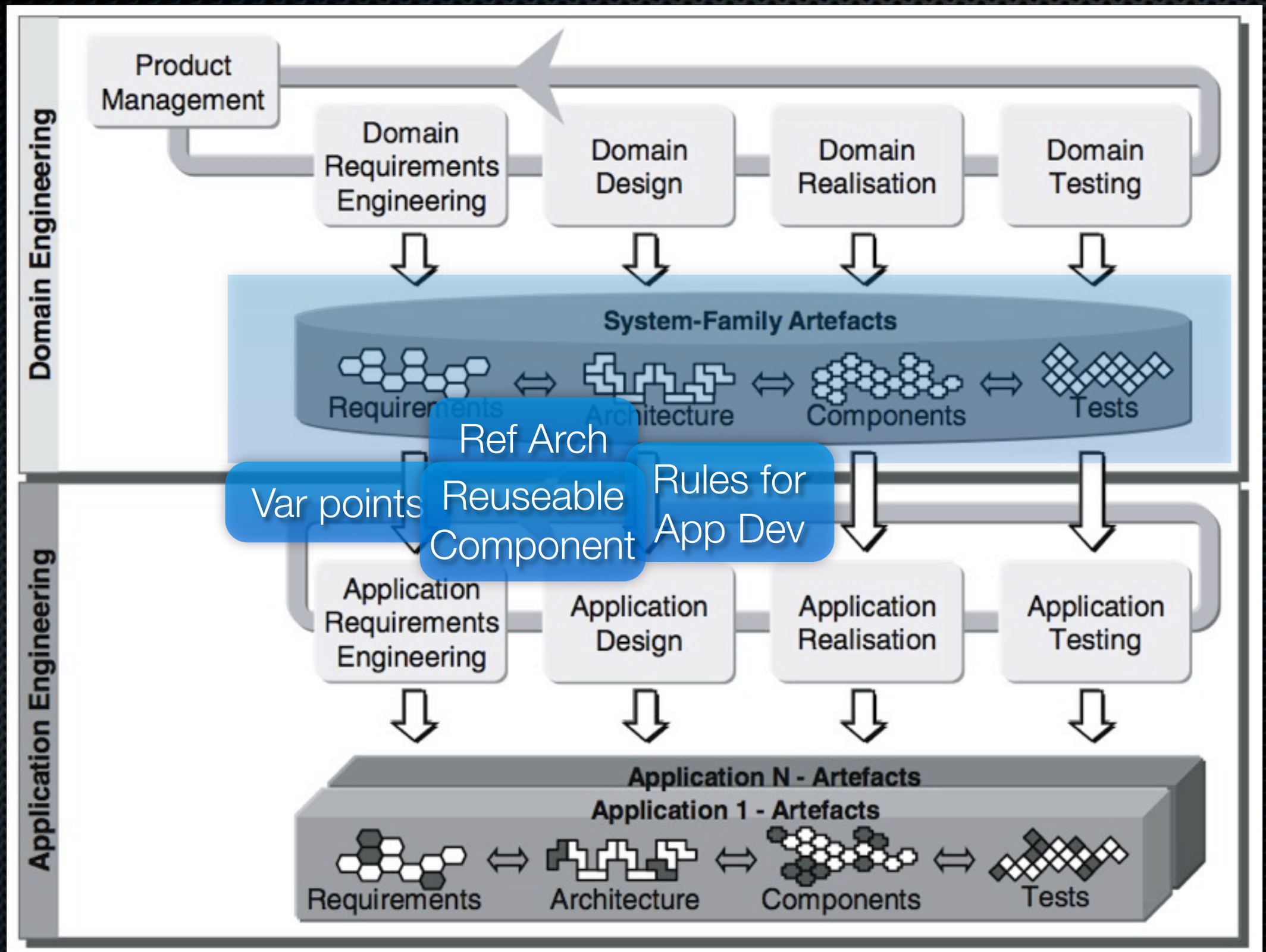
Domain and Application Engineering



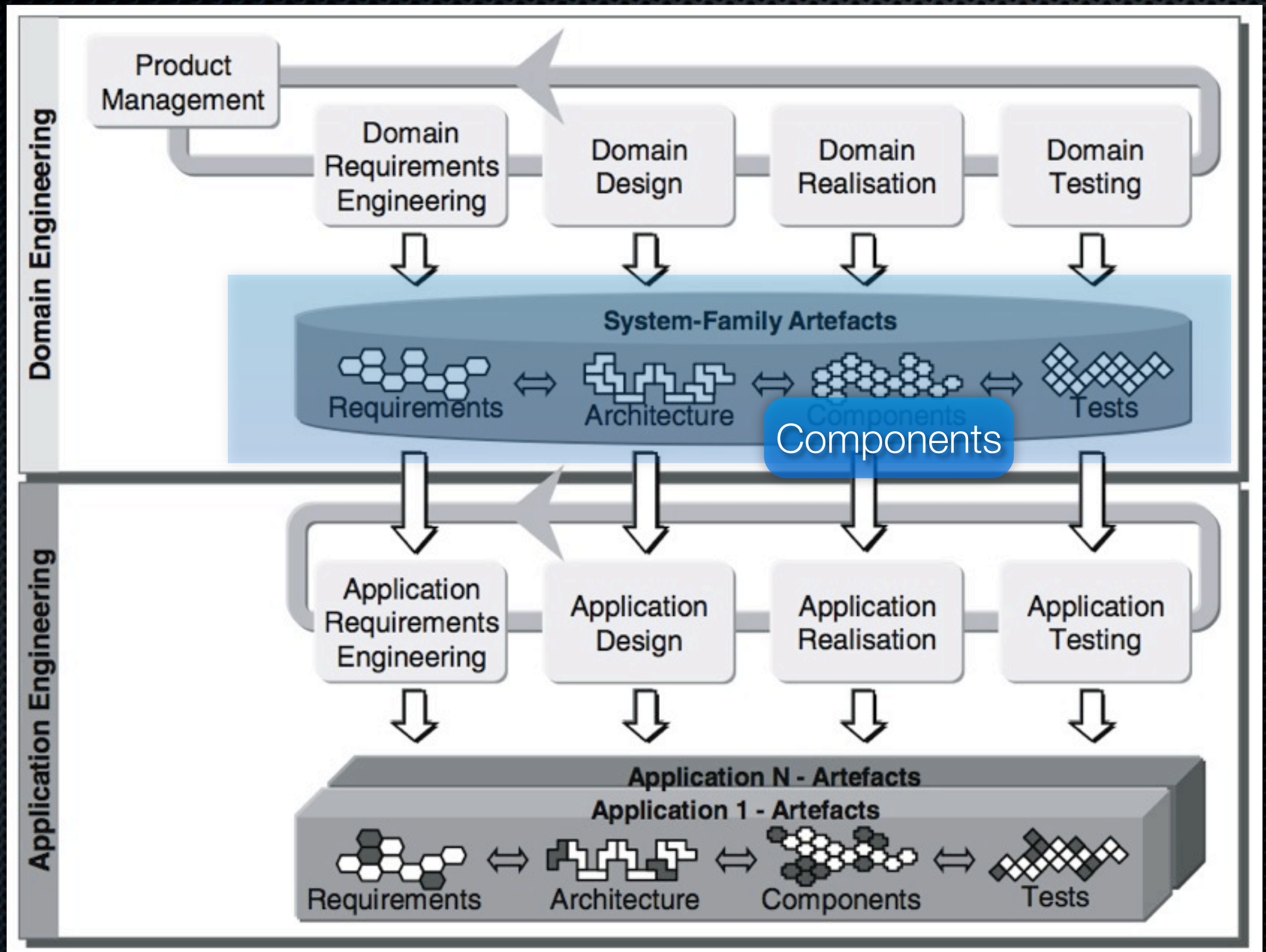
Domain and Application Engineering



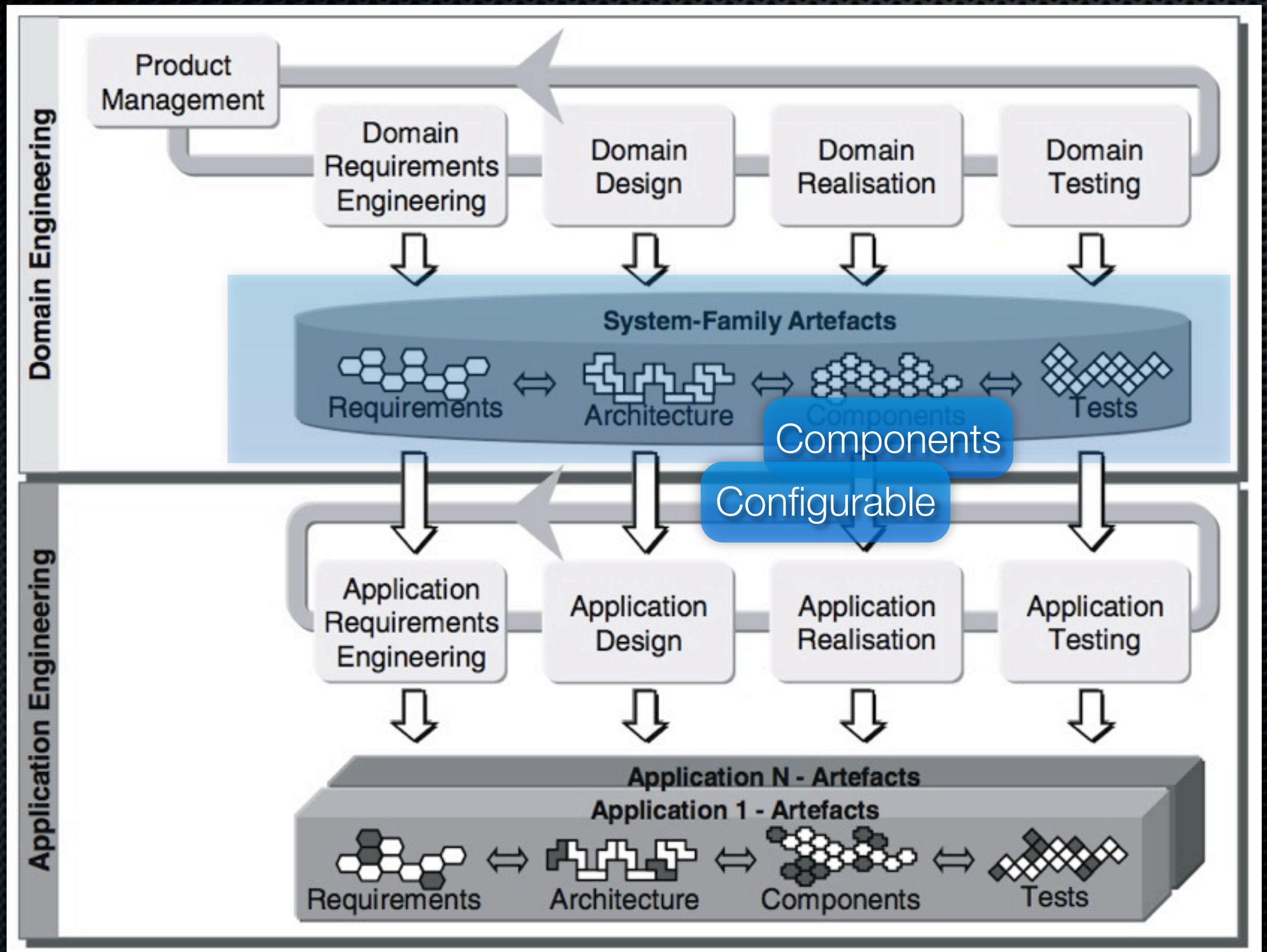
Domain and Application Engineering



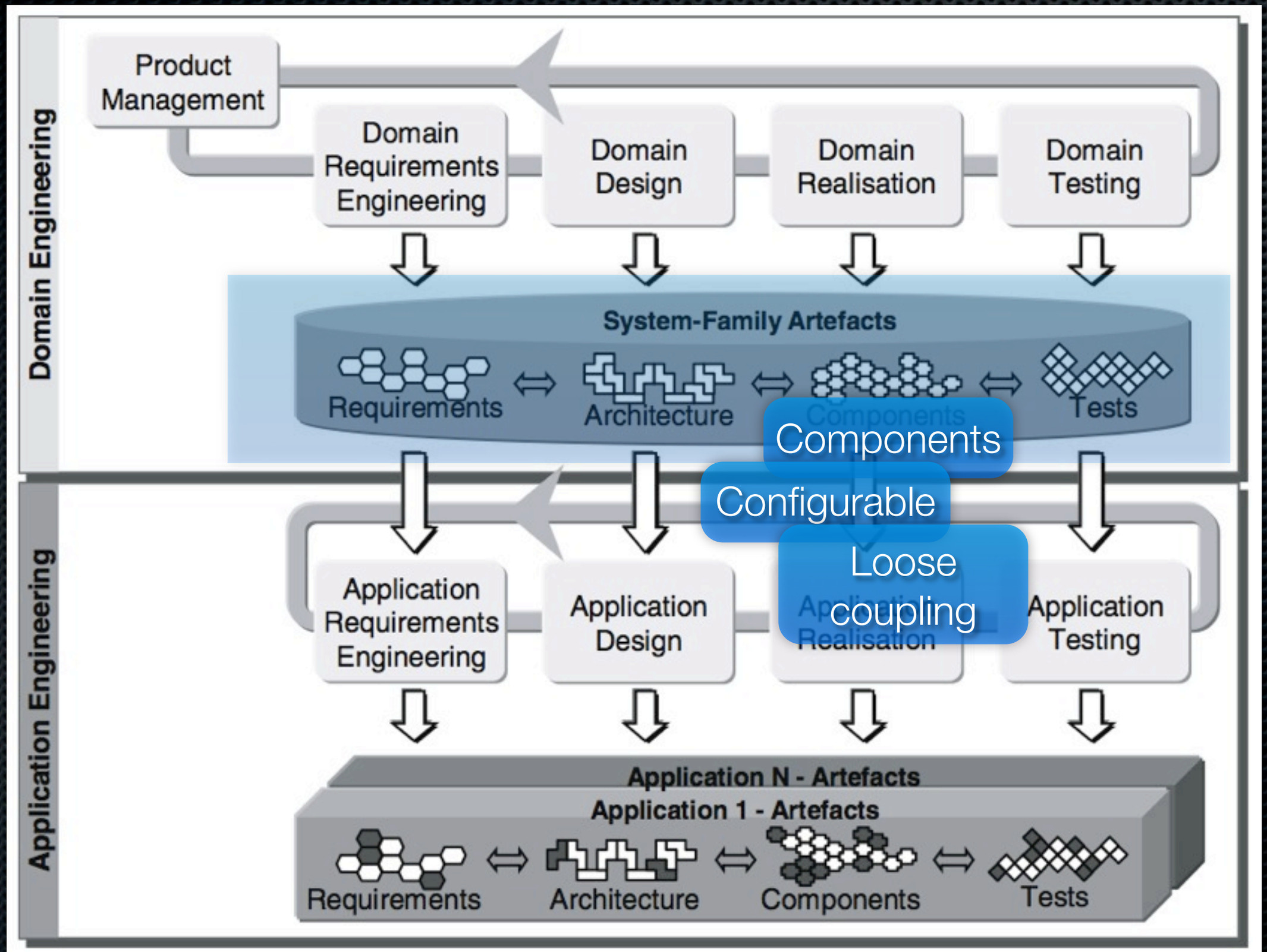
Domain and Application Engineering



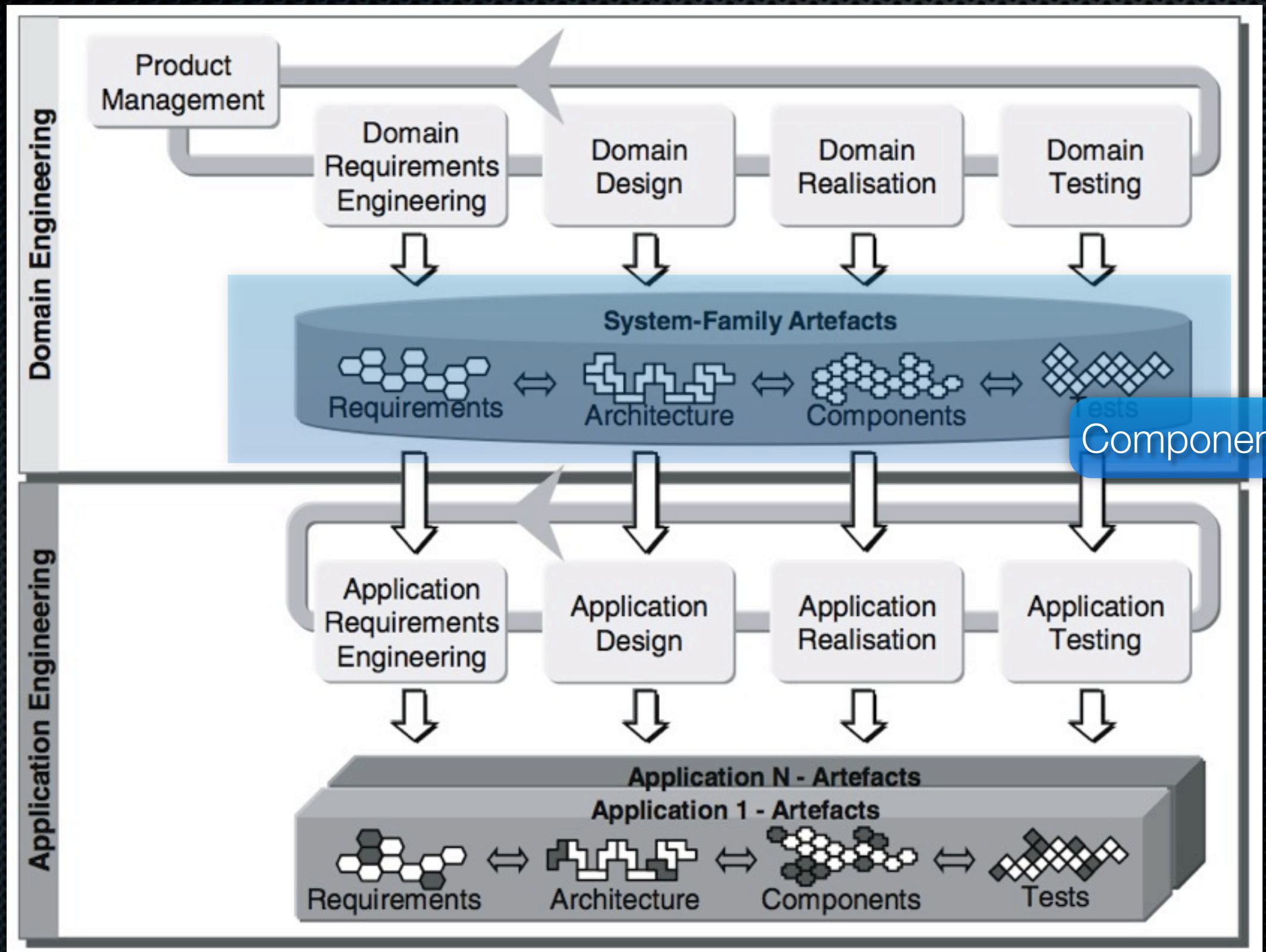
Domain and Application Engineering



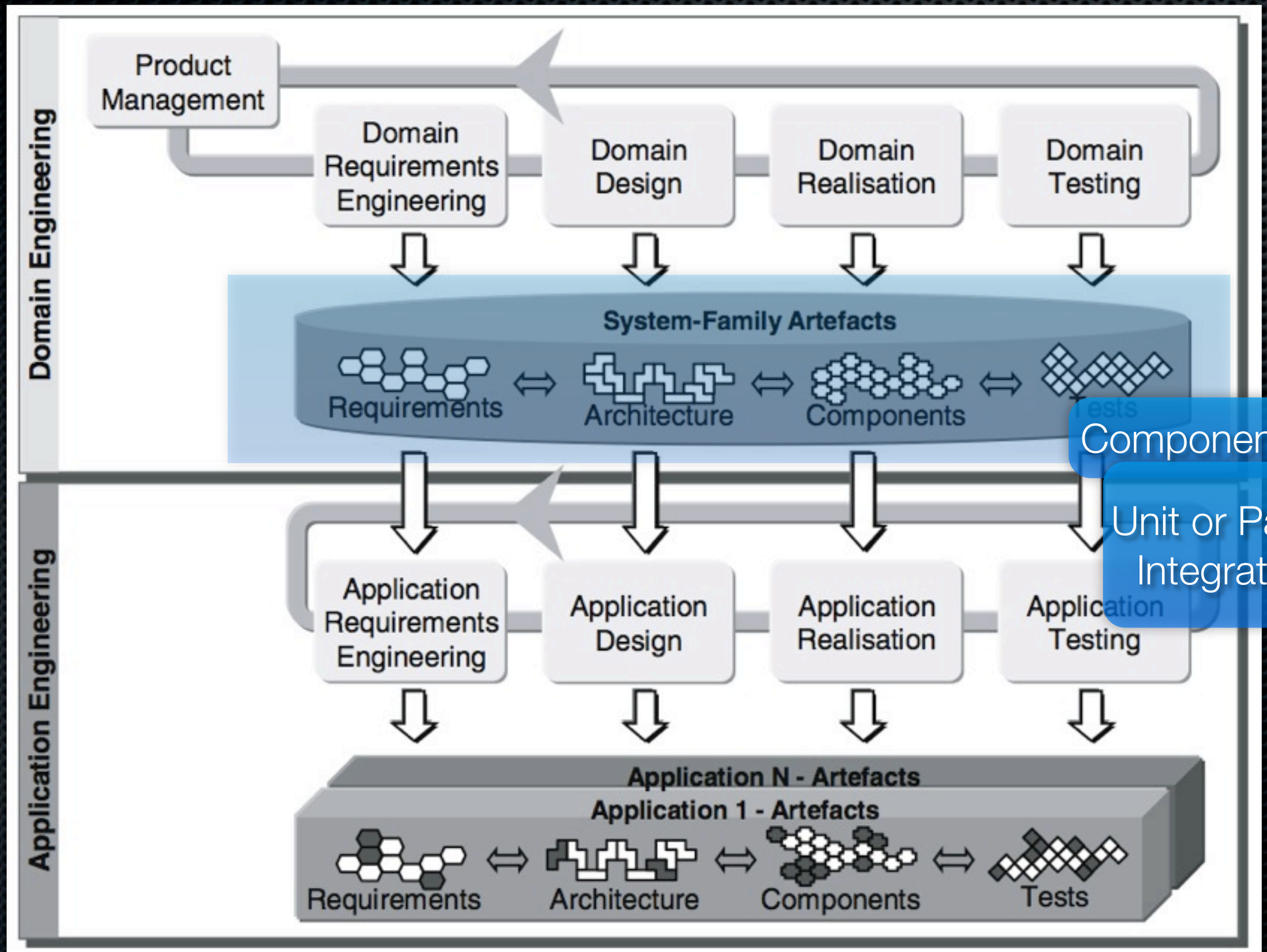
Domain and Application Engineering



Domain and Application Engineering



Domain and Application Engineering



Variability Management

- SPL = Commonality + Explicit Variability
- Variability is explicitly managed, i.e.
  - Defined, represented, discussed, exploited, implemented, evolved etc.

Feature	Prod. 1	Prod. 2	Prod. 3
Game engine	3D, C++	3D, C++	3D, C++
Score upload	No	Yes	Yes
Lead character	Mario	Ferrari	None, puzzle

Variability Management

- SPL = Commonality + Explicit Variability
- Variability is explicitly managed, i.e.
  - Defined, represented, discussed, exploited, implemented, evolved etc.

Variability is a first-class concept!

Feature	Prod. 1	Prod. 2	Prod. 3
Game engine	3D, C++	3D, C++	3D, C++
Score upload	No	Yes	Yes
Lead character	Mario	Ferrari	None, puzzle

Variability Management

- SPL = Commonality + Explicit Variability
- Variability is explicitly managed, i.e.
  - Defined, represented, discussed, exploited, implemented, evolved etc.

Variability is a first-class concept!

Feature	Prod. 1	Prod. 2	Prod. 3
Game engine	3D, C++	3D, C++	3D, C++
Score upload	No	Yes	Yes
Lead character	Mario	Ferrari	None, puzzle

Commonality, part of SPL

Variability Management

- SPL = Commonality + Explicit Variability
- Variability is explicitly managed, i.e.
  - Defined, represented, discussed, exploited, implemented, evolved etc.

Variability is a first-class concept!

Feature	Prod. 1	Prod. 2	Prod. 3
Game engine	3D, C++	3D, C++	3D, C++
Score upload	No	Yes	Yes
Lead character	Mario	Ferrari	None, puzzle

Commonality, part of SPL

Variation, supported in SPL

Variability Management

- SPL = Commonality + Explicit Variability
- Variability is explicitly managed, i.e.
  - Defined, represented, discussed, exploited, implemented, evolved etc.

Variability is a first-class concept!

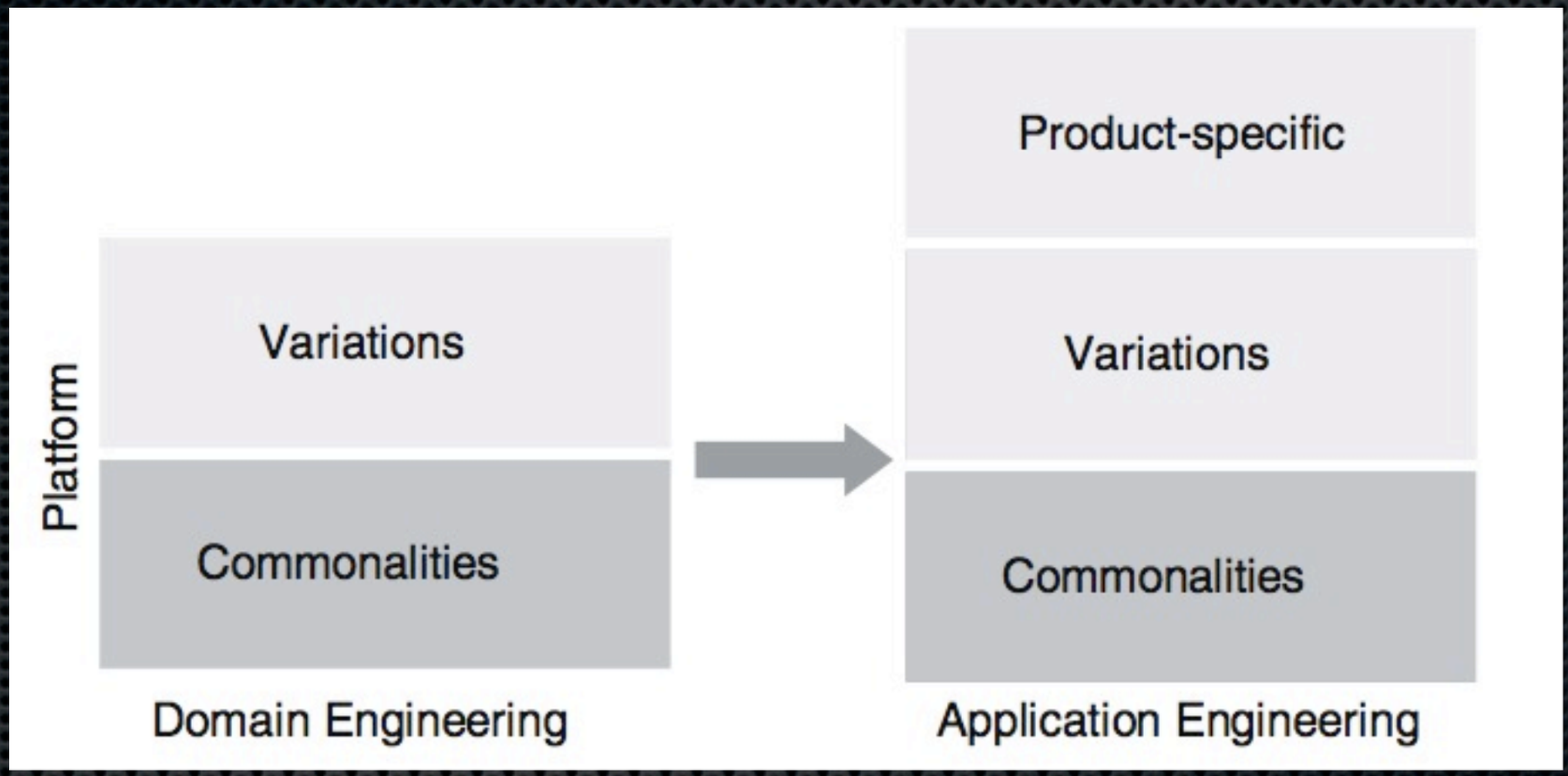
Feature	Prod. 1	Prod. 2	Prod. 3
Game engine	3D, C++	3D, C++	3D, C++
Score upload	No	Yes	Yes
Lead character	Mario	Ferrari	None, puzzle

Commonality, part of SPL

Variation, supported in SPL

Product-specific, not supported (now)

Types of Variability



Variability Documentation

- What varies?
  - Variation points
- Why does it vary?
  - Context, Reasons
- How does it vary?
  - Variants, Dependencies, Constraints
- For whom is it documented?
  - Internal & External Stakeholders
- Improves: Decision Making, Communication & Traceability

Feature-Oriented Domain Analysis [Kang98]

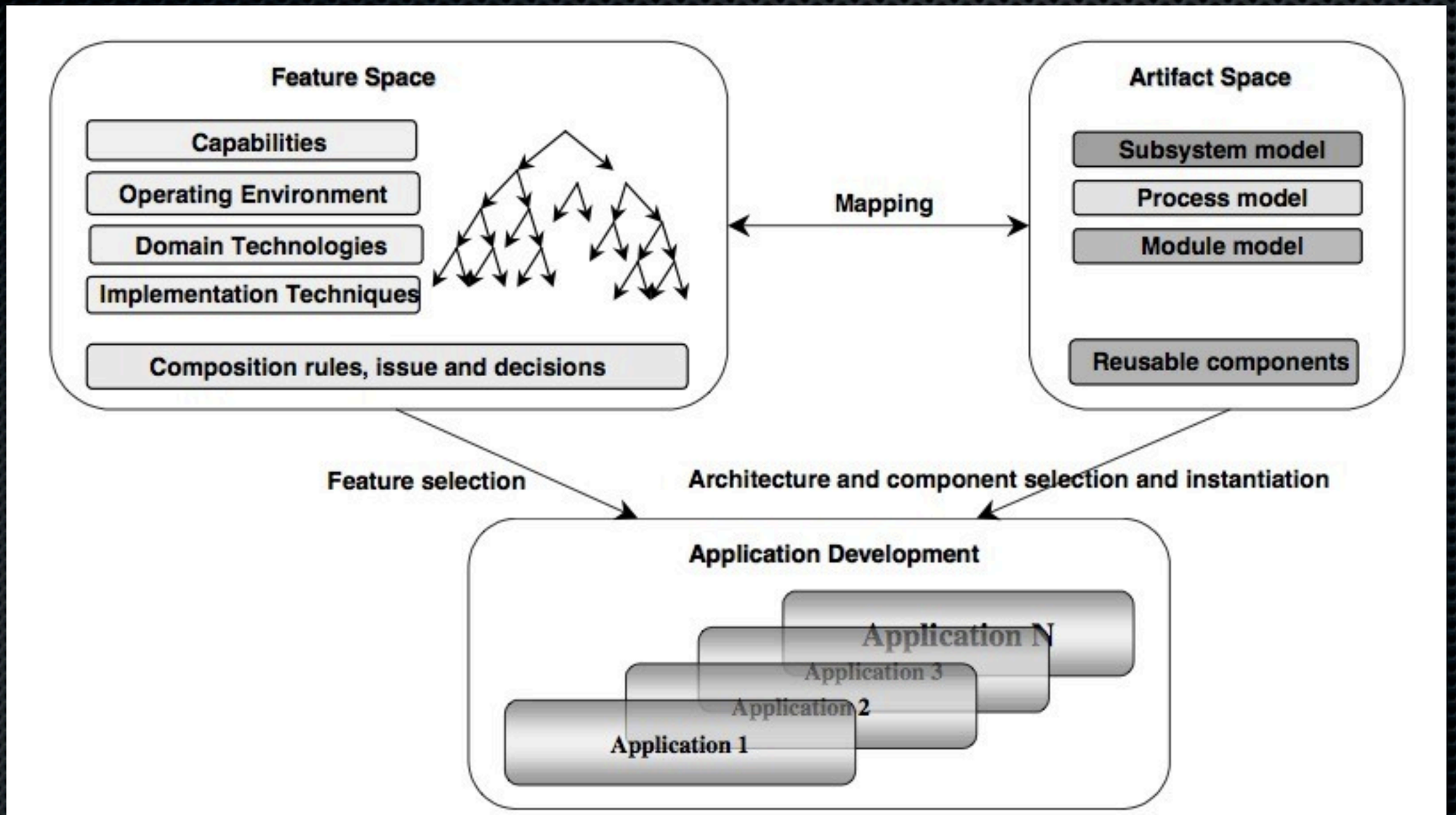


Figure 1-1: FORM's Concept of Application Development by Reuse of Domain Engineered Artifacts through Feature Selection.

Feature-Oriented Domain Analysis [Kang98]

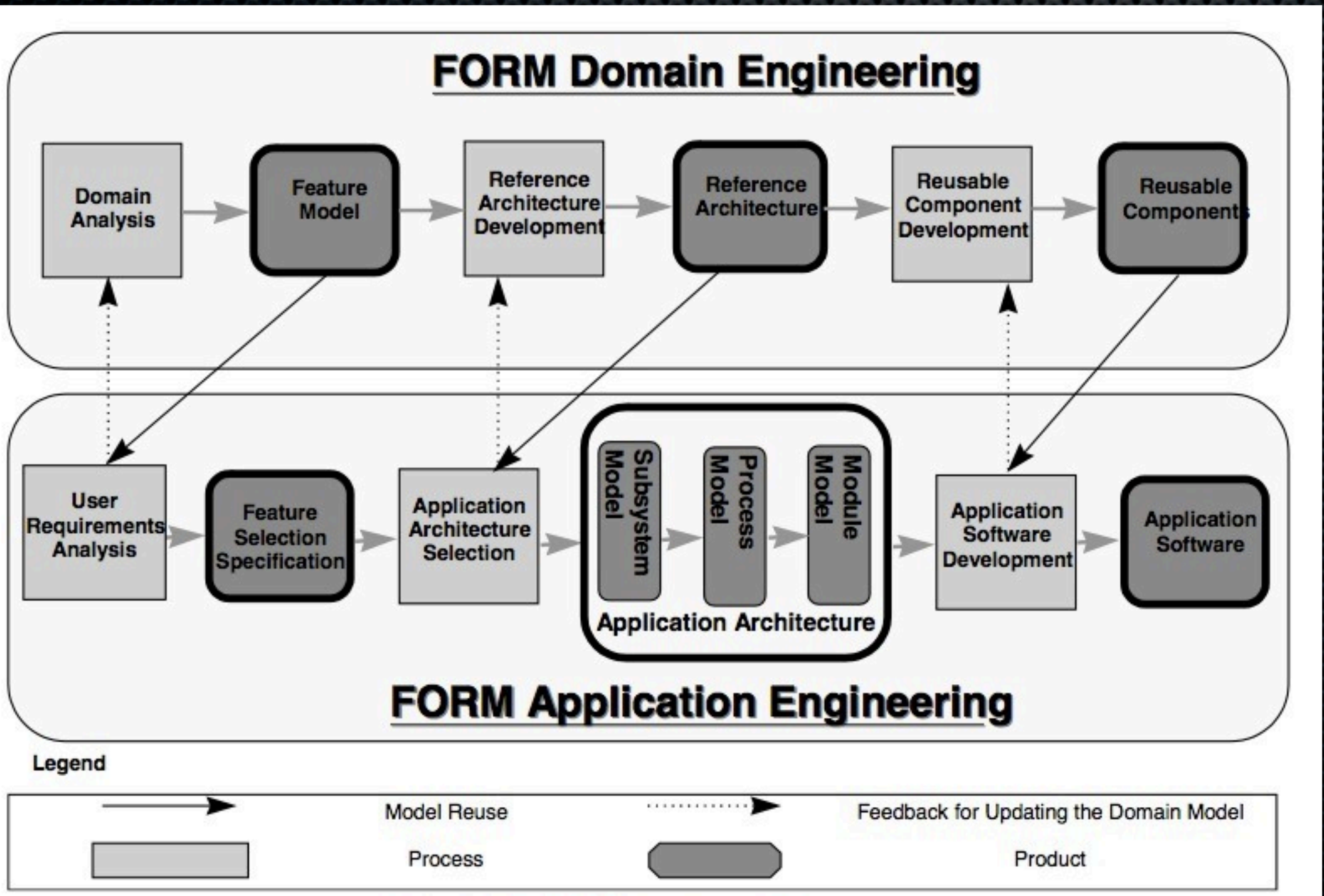
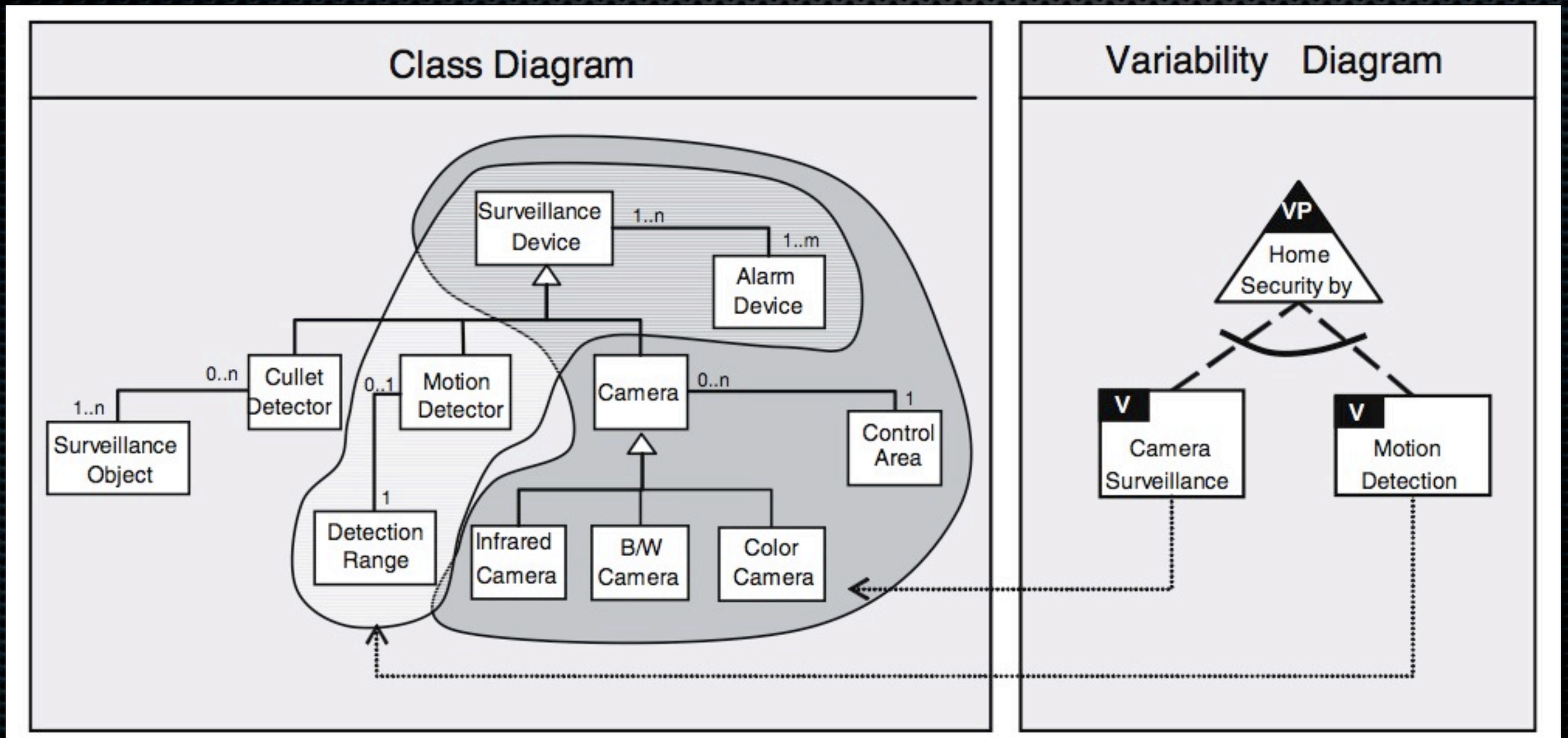


Figure 2-1: FORM Engineering Processes.

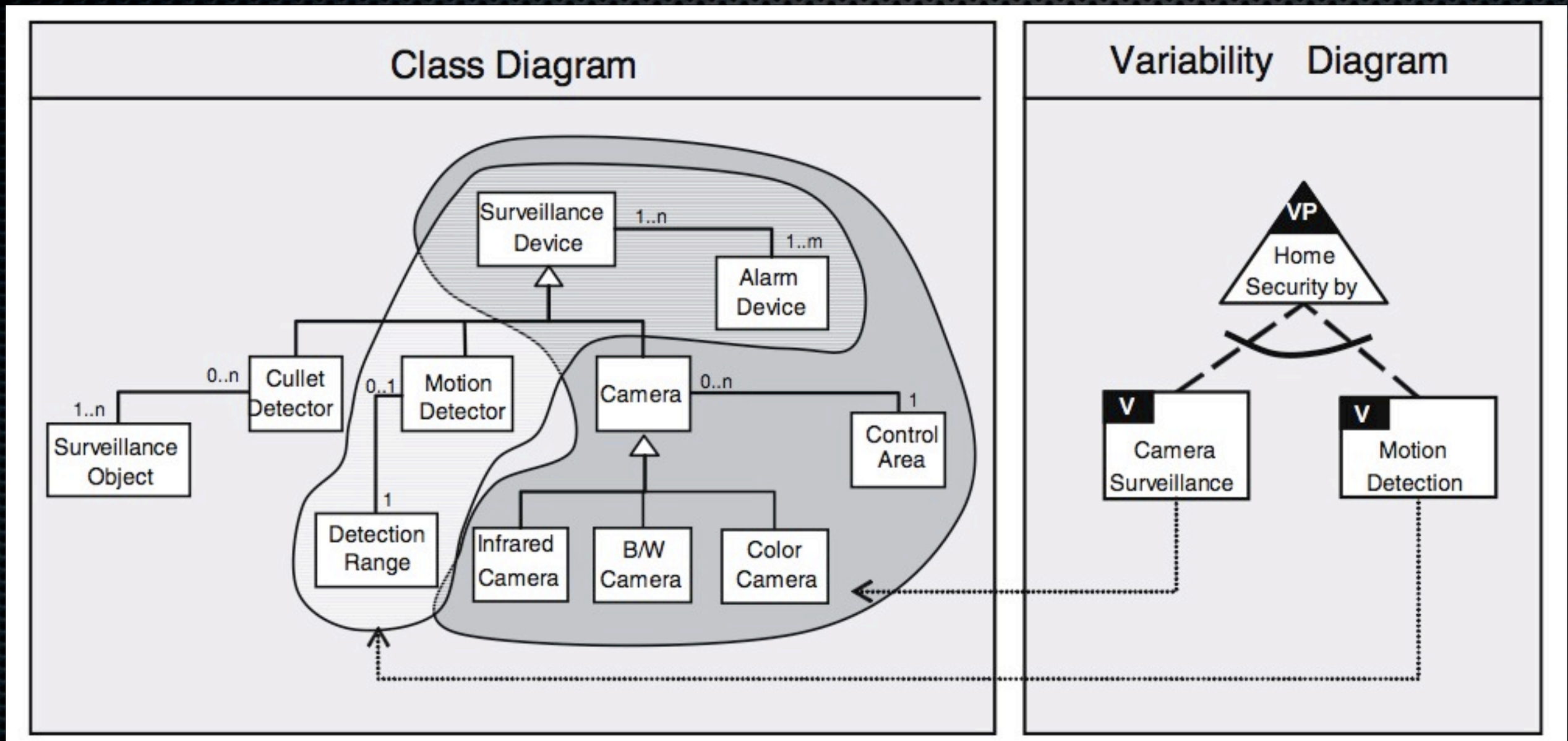
Feature-Oriented Domain Analysis [Kang98]

Graphical Variability Modeling

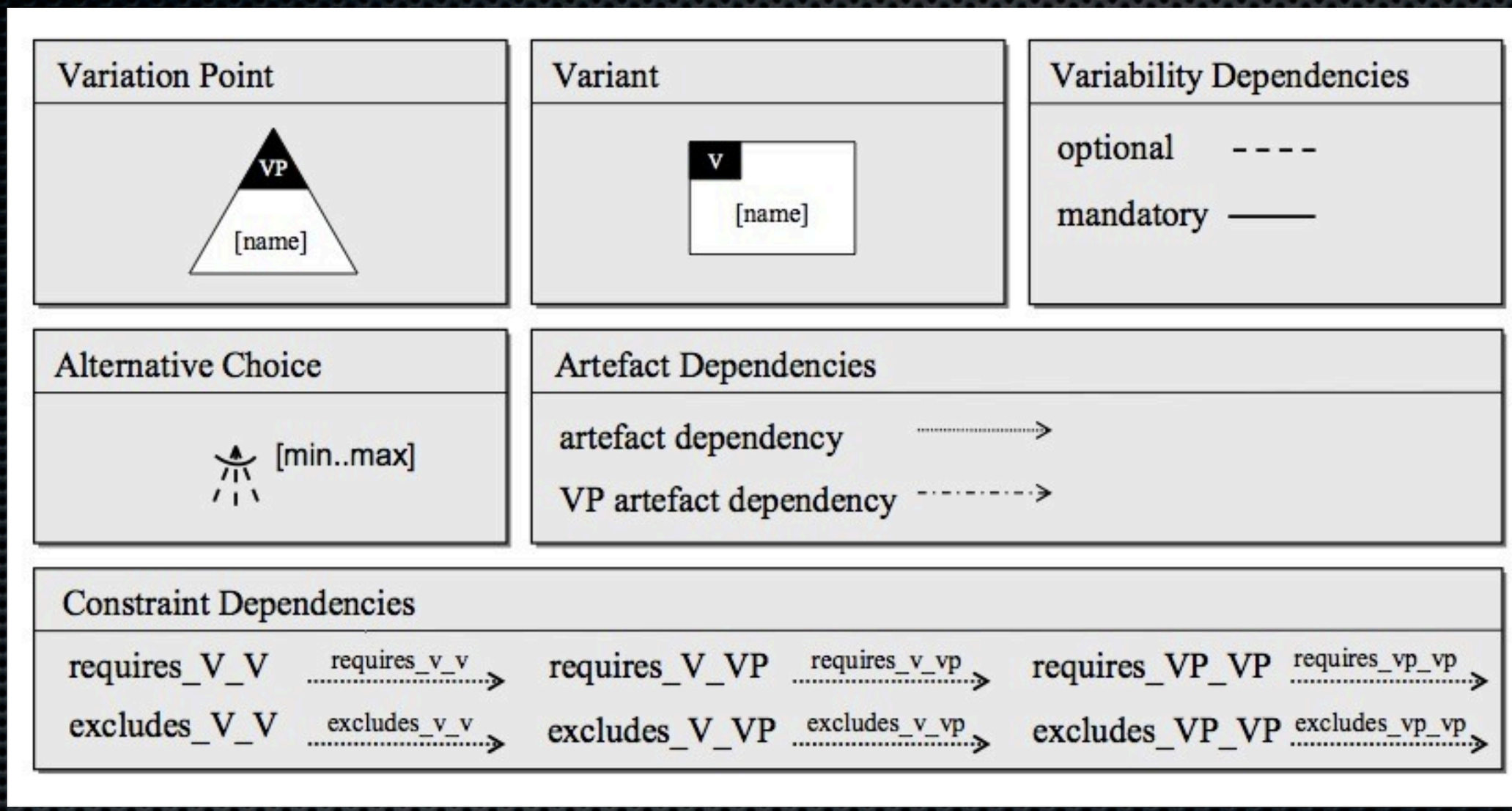


Graphical Variability Modeling

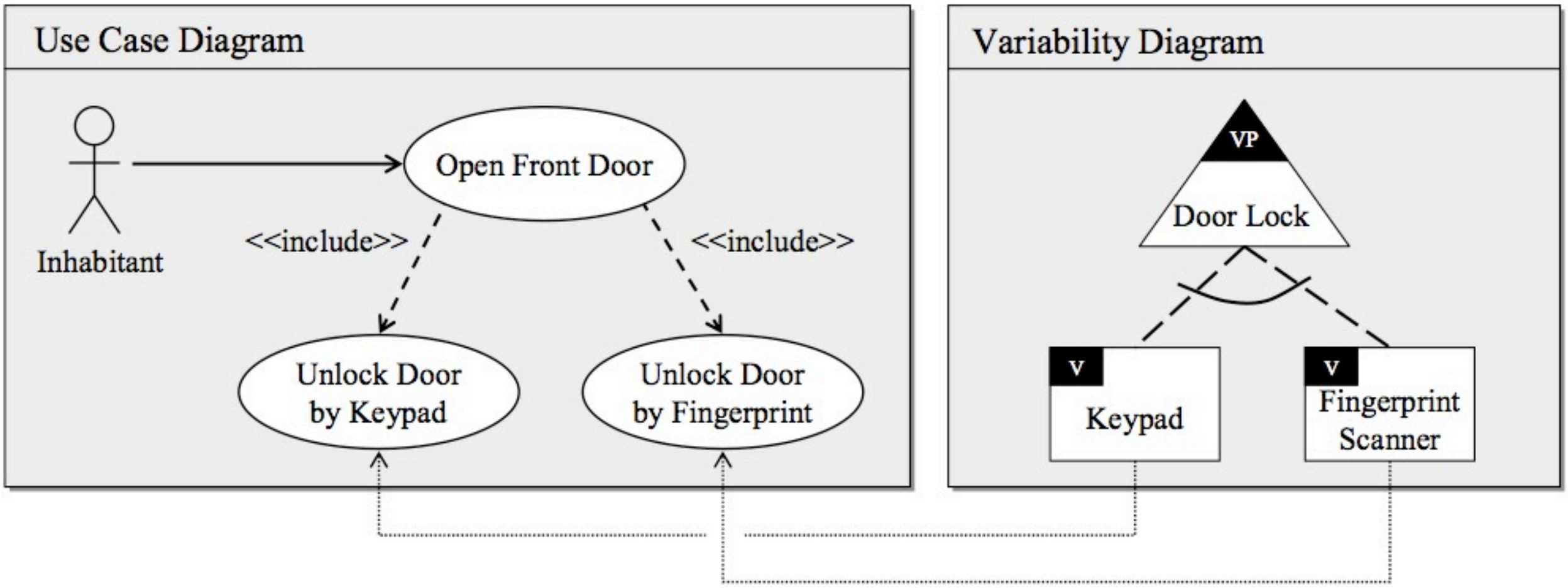
Separate Model!



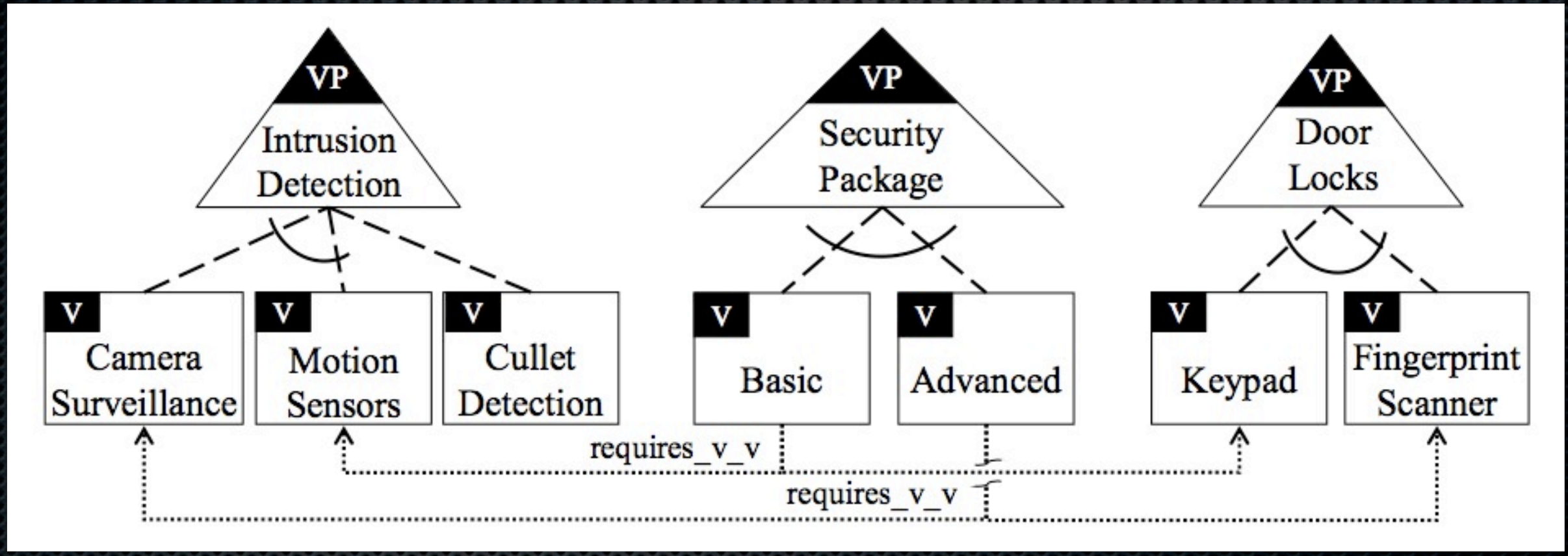
Graphical Variability Modeling (in OVM)



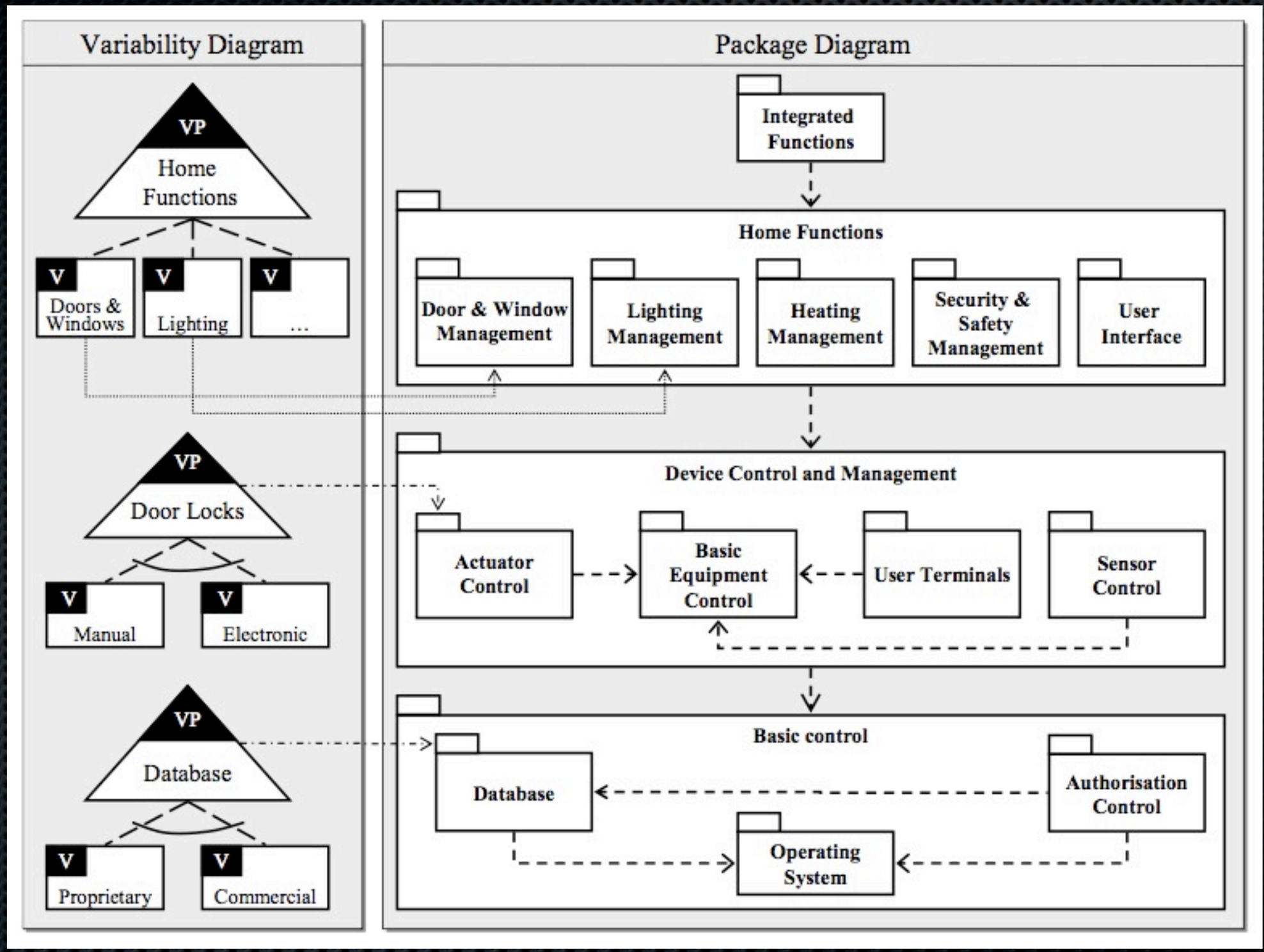
Same variability notation throughout



Packages of variants



Variability in packages/sub-systems



Architecture

Reference Architecture

- Single, shared architecture, common to all products
  - Normal architecture for commonalities
  - Variation points, variants etc for rest
- Not always there in practice, too plan-driven
  - Extract the reference architecture gradually

Time for a paper...

Experiences with Mobile Games Product Line Development at Meantime

Vander Alves
Fraunhofer IESE
vander.alves@iese.fraunhofer.de

Tarcísio Câmara
Meantime Mobile Creations
tarcisio.camara@meantime.com.br

Carina Alves
Federal University of Pernambuco
cfa@cin.ufpe.br

Abstract

Small and Medium Enterprises (SME) represent a significant percentage of software companies. Although there are some experience reports on using Software Product Line (SPL) techniques in SMEs, there is a lack of such reports on the Mobile Games domain. This domain is attractive both economically and technically, with a significant amount of variability, thus suggesting the use of a SPL approach. Accordingly, we conduct an exploratory study in a company in this domain, eliciting the main challenges, determining if and how SPL approaches help to address them, and raising hypotheses to guide future studies.

ity of game designers and developers rather than on good practices from other software domains. Nevertheless, developers in this domain have to manage paramount variability, most of which stems from a strong portability requirement, demanding games do be deployed in potentially hundreds of devices. Accordingly, from this point of view, a SPL approach seems adequate.

In this context, this paper reports an exploratory case study conducted at Meantime, a SME that is a leading mobile game development company based in Recife, Brazil. The goal of the case study was to provide insights into the following questions:

- What are the key challenges faced during the develop-

Industry example: Meantime Game Company

- Brazilian company developing mobile games
  - 60 games, 400 devices, 6 languages, 40 developers
- Critical requirement: Portability (Many mobiles)
  - User interface differences
  - CPU, memory and size constraints
  - Support API differences (J2ME, BREW & proprietary)
  - Carrier-specific requirements
  - Internationalization

Industry example: Meantime Game Company

- Developed MG2P = Meantime Game Porting Platform
  - Mobile Domain Database (MDD)
  - Meantime Base Architecture (MBA)
  - Meantime Build System (MBS)

- MDD captures basic Commonality + Variability
  - Variations: Device-specifics, Game types/APIs, Known issues, Language, Game features
  - Families of similar MobApps and Games (in porting context)
  - Typical device for each family chosen (least powerful, most issues)

Configuration knowledge in MDD

Table 2. Configuration knowledge mapping device variability to preprocessing tokens.

Category	Sub-Category	Variation	Token
Device specific	Screen Size	128x117	device_screen_128x117
		128x128	device_screen_128x118
		130x130	device_screen_130x130
		128x142	device_screen_128x142
		128x149	device_screen_128x149
Game Features	Usage of Tiled Layer API	Meantime API	game_tiledlayer_api_meantime
		MIDP 2.0 API	game_tiledlayer_api_midp2
		Siemens Game API	game_tiledlayer_api_siemens

Industry example: Meantime Game Company

- Meantime base Architecture
  - Same code base and file structure for all games
  - J2ME does not allow libraries => MBA copied for each new game
  - Pre-processing tokens from MDD handles variability
- Meantime build system
  - Built on Antenna pre-processor and Ant, more flexible

Architectural Concerns

- Architecturally significant requirements
  - Key requirements affecting the whole architecture
- Conceptual architecture
  - Key concepts of architecture
- Architectural structure
  - Decomposition into components and relations
- Architectural texture
  - Rules for using, instantiating and evolving architecture

Architecturally Significant Requirements

- Central to the purpose of the products, or,
- Technically challenging / Technical constraints
- Examples:
  - The system must encrypt all network traffic
  - The game must deploy on all mobile phones by the top 5 manufacturers that are released after 2007
  - The system must always give responses to user queries within 3 seconds
  - The system must provide a visual overview of the current flow of resources in the factory being managed
- Quality/Non-func. requirements often decisive

Conceptual Architecture

- Most important concepts + their relations
- Mental model of of domain to understand and simplify the problem
  - (Related to “System Metaphor” in Extreme Programming)

Architectural Structure

- Division into components
  - Sub-systems/units with clear interfaces
- Connections between components

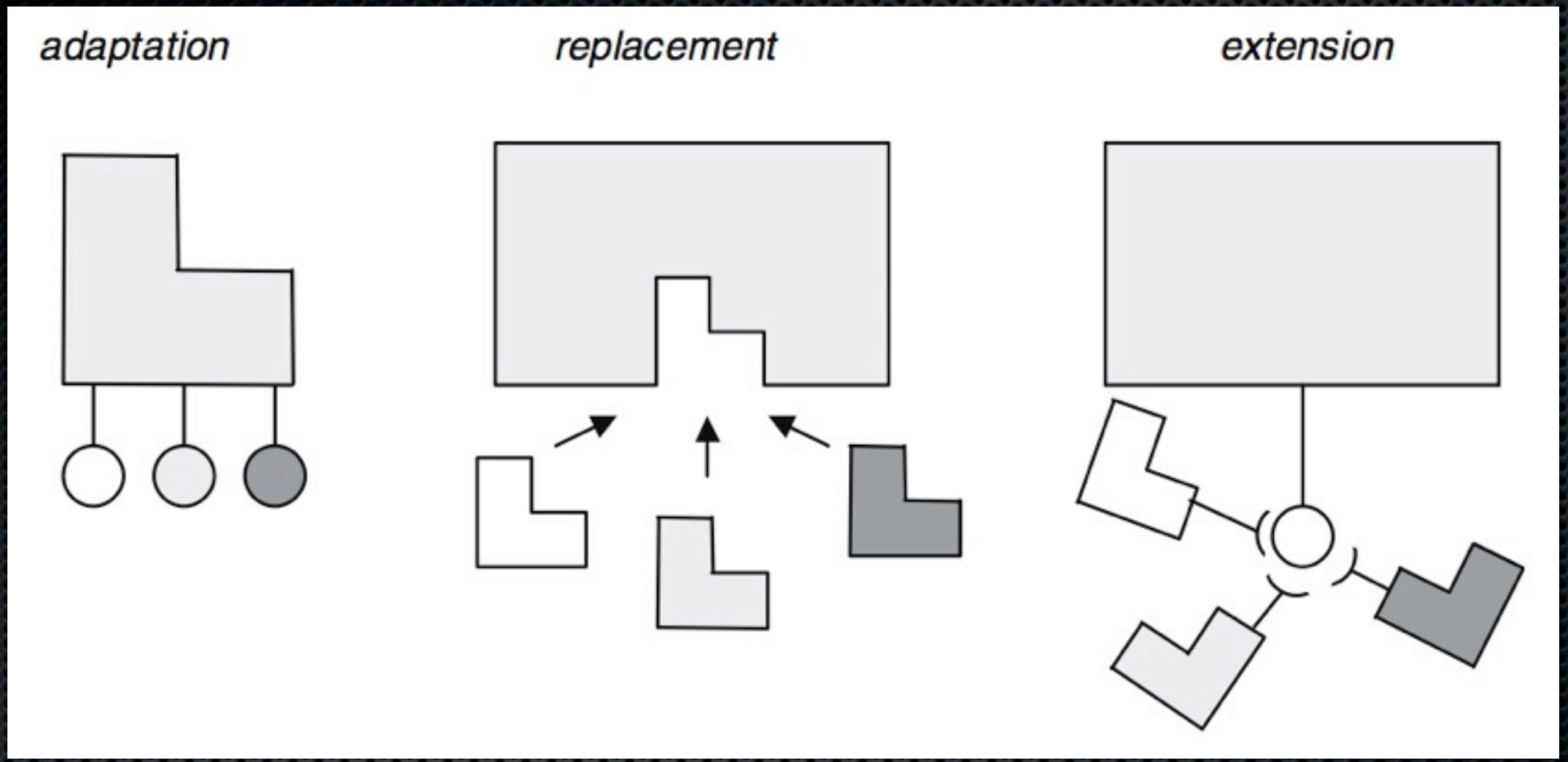
Architectural Texture

- “Manual” for the Reference Architecture
  - Guidelines, rules, “Philosophy” for
    - Using and
    - Evolving the RefArch
- Examples:
  - Coding standard
  - Design patterns
  - Architectural styles

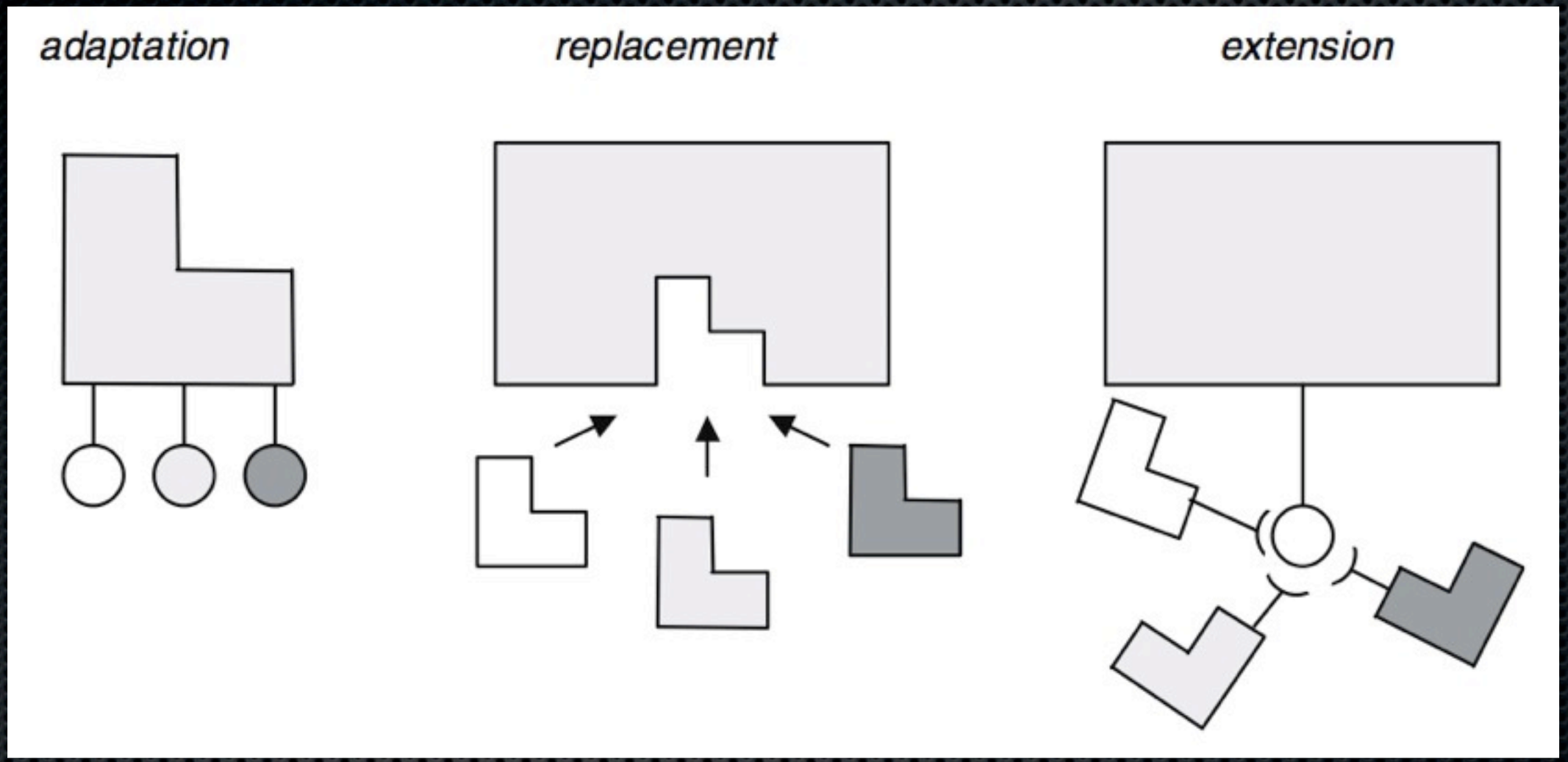
Creating a Reference Architecture

- “Normal” architecting methods can be used
  - Attribute-Driven Design, ..., OO, ..., Design Patterns, ...
- Differences:
  - More products, often more Stakeholders => Communicate
  - Also more Requirements conflicts => Resolve (elicited)
- Three basic ways to support variability:
  - Adaptation
  - Replacement
  - Extension

Variability Mechanisms

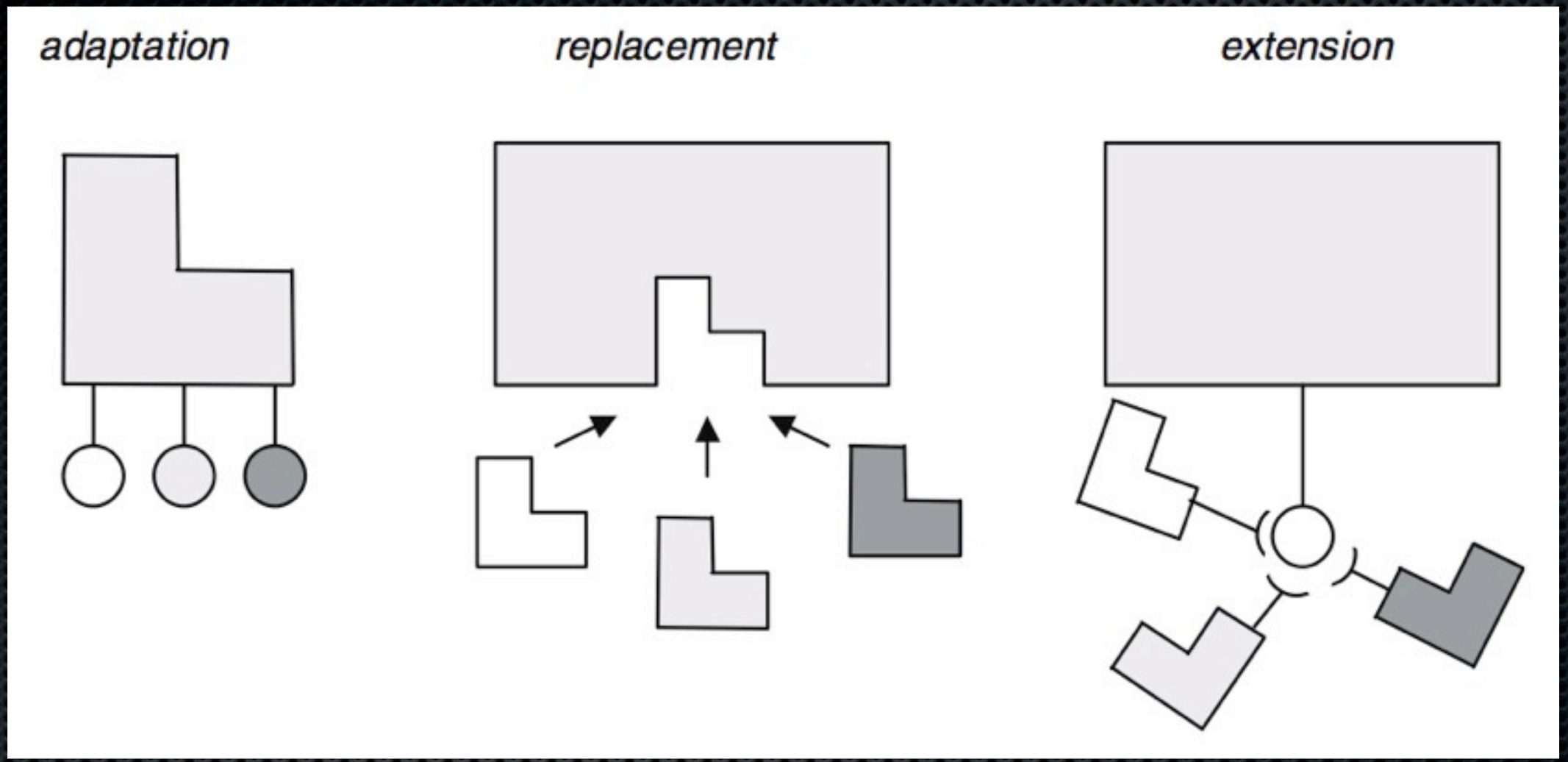


Variability Mechanisms



Only 1 component implementations
Adaptable behavior

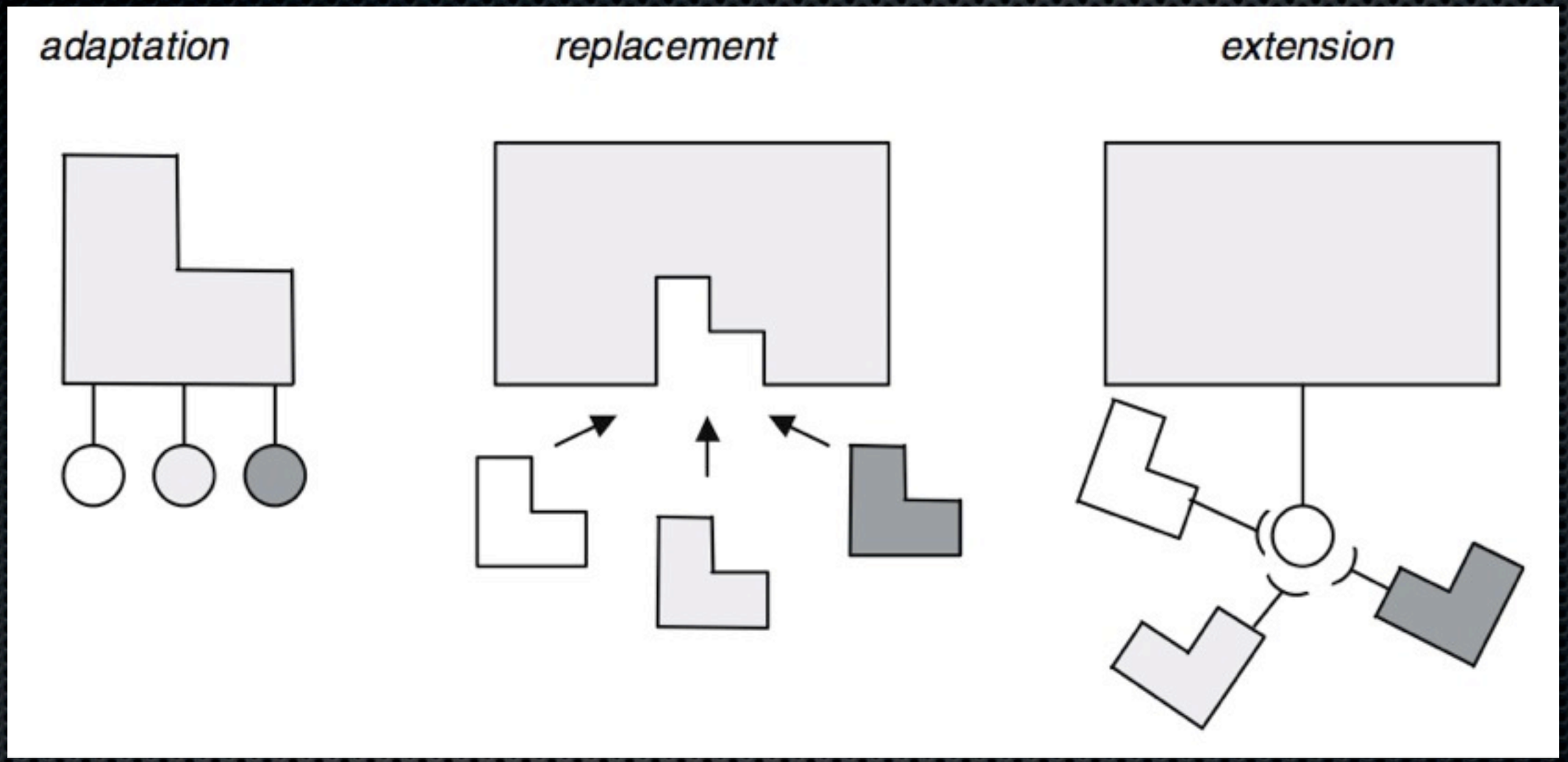
Variability Mechanisms



Only 1 component implementations
Adaptable behavior

Multiple component implementations
Choose one, or develop product-specific

Variability Mechanisms



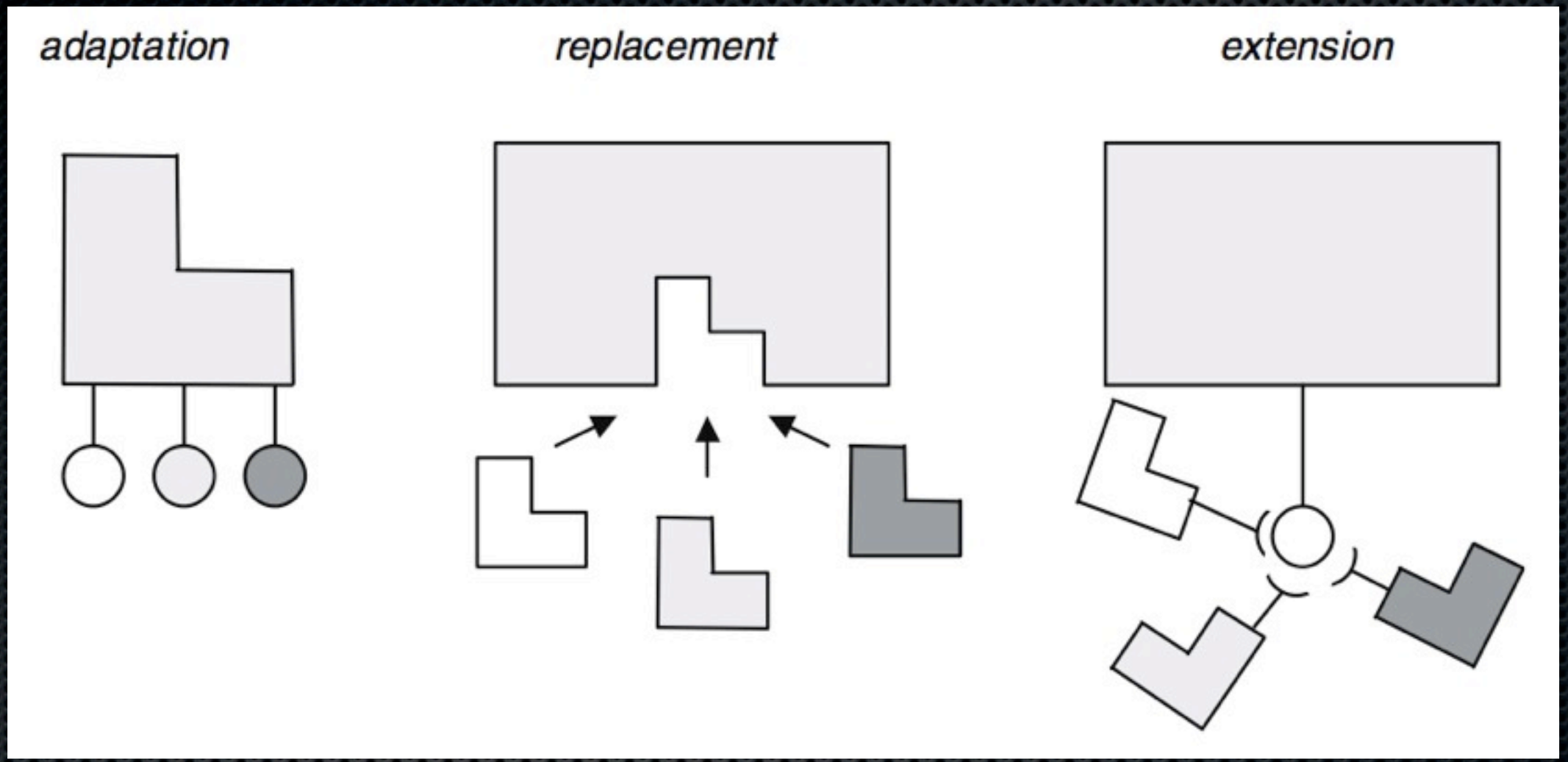
Only 1 component implementations
Adaptable behavior

Multiple component implementations
Choose one, or develop product-specific

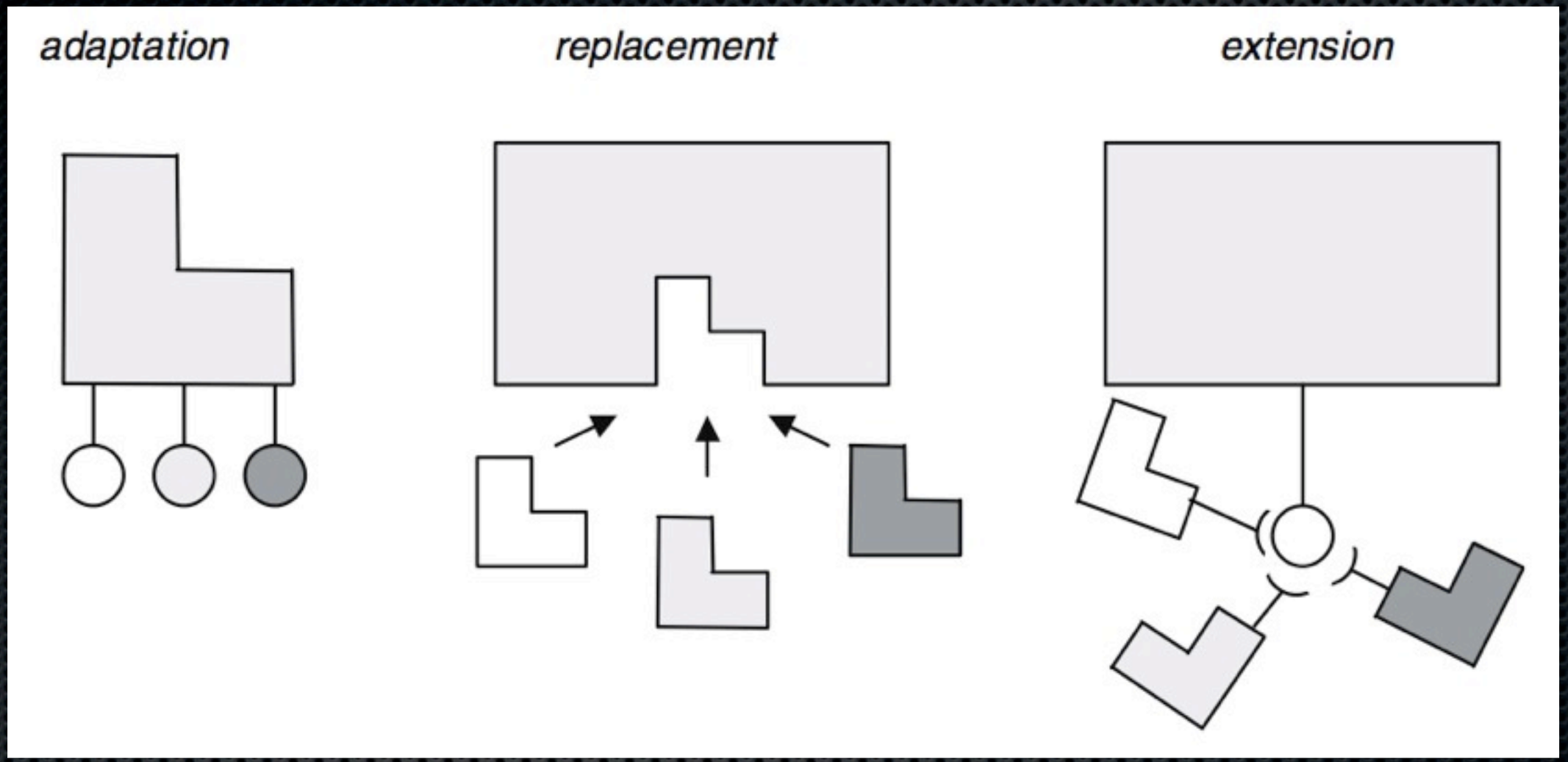
Generic interface for adding components

Variability mechanisms

Variability Mechanisms

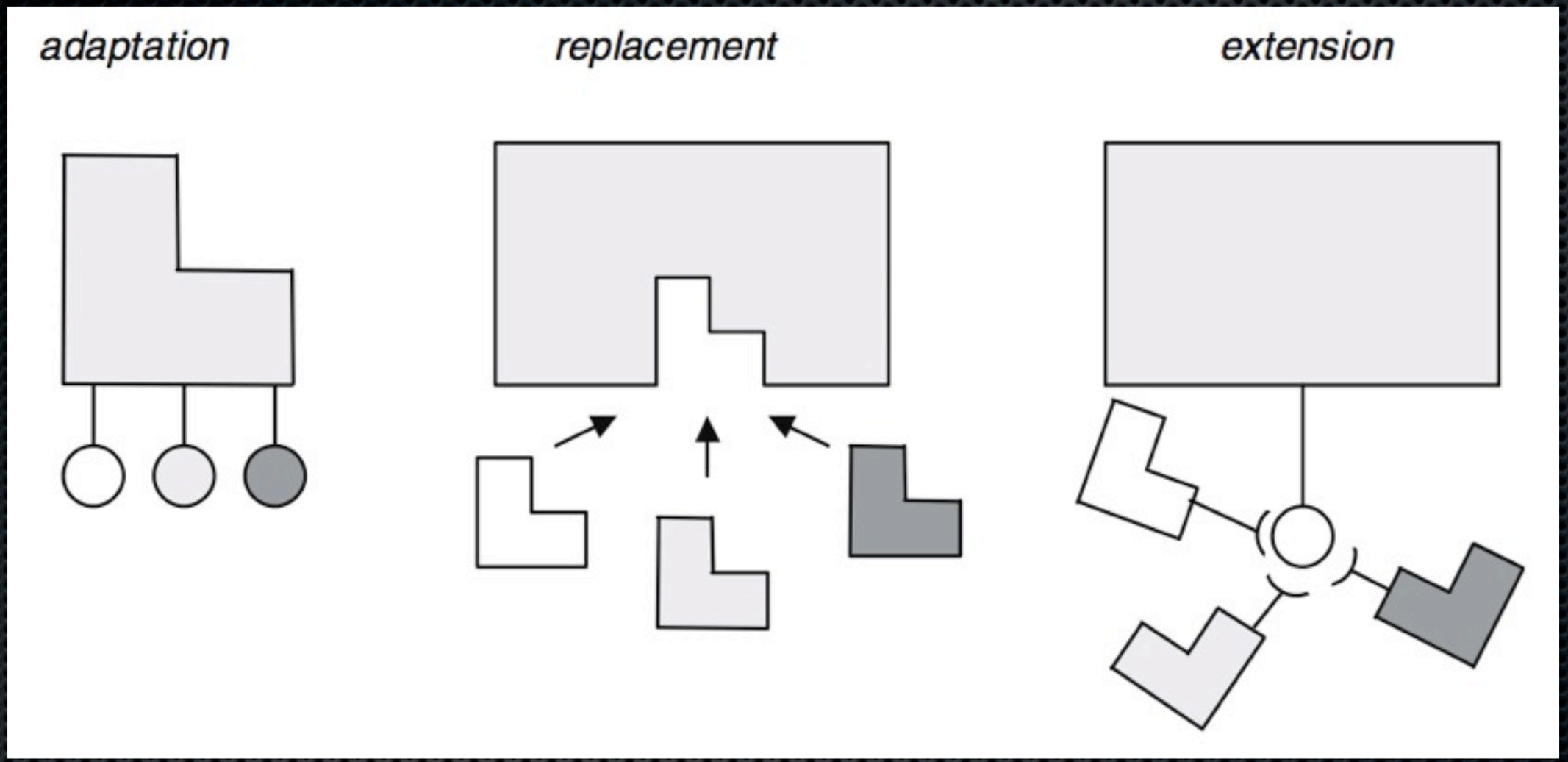


Variability Mechanisms



Only 1 component implementations
Adaptable behavior

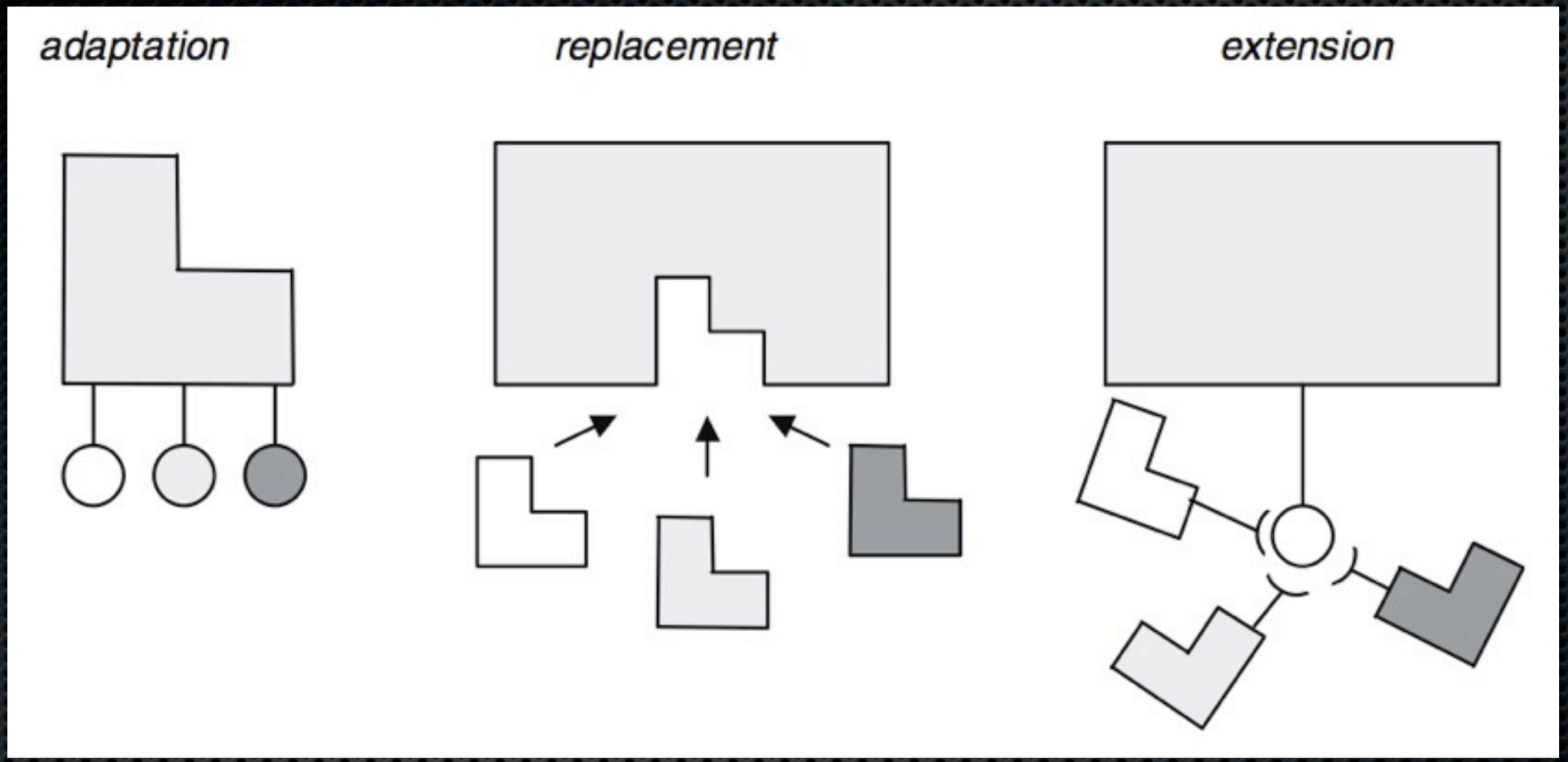
Variability Mechanisms



Only 1 component implementations
Adaptable behavior

Multiple component implementations
Choose one, or develop product-specific

Variability Mechanisms



Only 1 component implementations
Adaptable behavior

Multiple component implementations
Choose one, or develop product-specific

Generic interface for adding components

Adaptation mechanisms

- Inheritance
  - subclass changes/overrides behavior
- Patching
  - partial behavior change with little maintenance
  - DE: component, AE: patch
- Compile-time config
  - Pre-processors or macros, Makefiles
- Configuration
  - Interface to choose between multiple implementations
  - Parameters or configuration file to make choice

Replacement mechanisms

- Code generation
  - Generates code from high-level description (model, script)
  - Glue code or whole components/sub-systems
- Component replacement
  - Default component is replaced with another one
  - Often 3rd party components
  - Wrappers may be needed

Extension mechanisms

- Plug-ins
  - Architecture has interface to “plug in” components
  - Example: CORBA, COM, etc
  - Example: Strategy Design Pattern (functionality can be selected at runtime)

Variability & Commonality SPL Motivations

- Increase in the number of products that can be released
- Manage multiple, diverse products in one portfolio
- Improve product commonality
  - Not only for complexity management,
  - also for marketing (same look-and-feel)