Advanced Topics of Software Engineering (ASE)

Chapter 2. From requirements to system design

Prof. Dr. Florian Matthes, Prof. Dr. Alexander Pretschner

Chair of Software Engineering for Business Information Systems (sebis)
Faculty of Informatics
Technische Universität München
wwwmatthes.in.tum.de

From requirements to system design

What is software reuse?

Basic idea of reuse:

- Do not reinvent the wheel!
- If you or someone else already solved a problem for one project
  → reuse the solution for the same problems in other projects.

One of the key concepts of software architecture:

- Clearly defined architecture (and interfaces) support reuse.
- Separation of concerns and low coupling between components
  → allows detachment and reuse in other architectures.
- Concepts like SOA or Component-based architectures focus on reusability of components and services
  → goal: minimize customizations, maximize reuse.

Note: There are lots of commercial and open source **technical** components (e.g., XML parser, UI components, math libraries), which are based on clearly defined functional requirements. However, reusable business components (e.g., user management, billing services, reservation services) are still very rare.

Why reuse often does not work…

Technical issues:

- Reuse only done *in-the-small* or at the code level.
  - Only very limited scope.
  - No comprehensive consideration of overall system NFRs.
  - Often done unsystematically
    → results in clones that are hard to maintain.
- Inconsistent or incomplete component specifications.
  - Unclear quality characteristics
  - Unclear dependencies
  - Unclear correctness and completeness guarantees

Why reuse often does not work…

Organizational issues:

- Reuse rarely planned in advance or done too late
  - (but maybe this can’t be planned …)
- Lack of motivation and incentives for reuse
  → *not-invented-here* syndrome
- Effective reuse depends on clearly defined interfaces and good system and requirement knowledge
  → often not given for grown systems
- Lack of marketplaces and widely deployed standards
  - Component-based marketplaces like *componentsource.com* did not really take off
  - Increasing usage of web service and especially open source technology seems to change things here …

Why reuse often does not work…

- Conflict between flexibility and stability
  - Need to be flexible enough to fit multiple contexts
    → too much flexibility leads to expensive adaptations
  - Need to be stable and comprehensive enough to really generate reusability benefits
    → too little flexibility limits generic applicability

- Typical for software architecture, there exists no silver bullet and optimal trade-off
  → need to individually assess benefits and shortcomings

Later, we will discuss an approach to address this conflict within one organization
→ Product line engineering

Architectural principles and reuse

Architectural principles are important for supporting reuse:

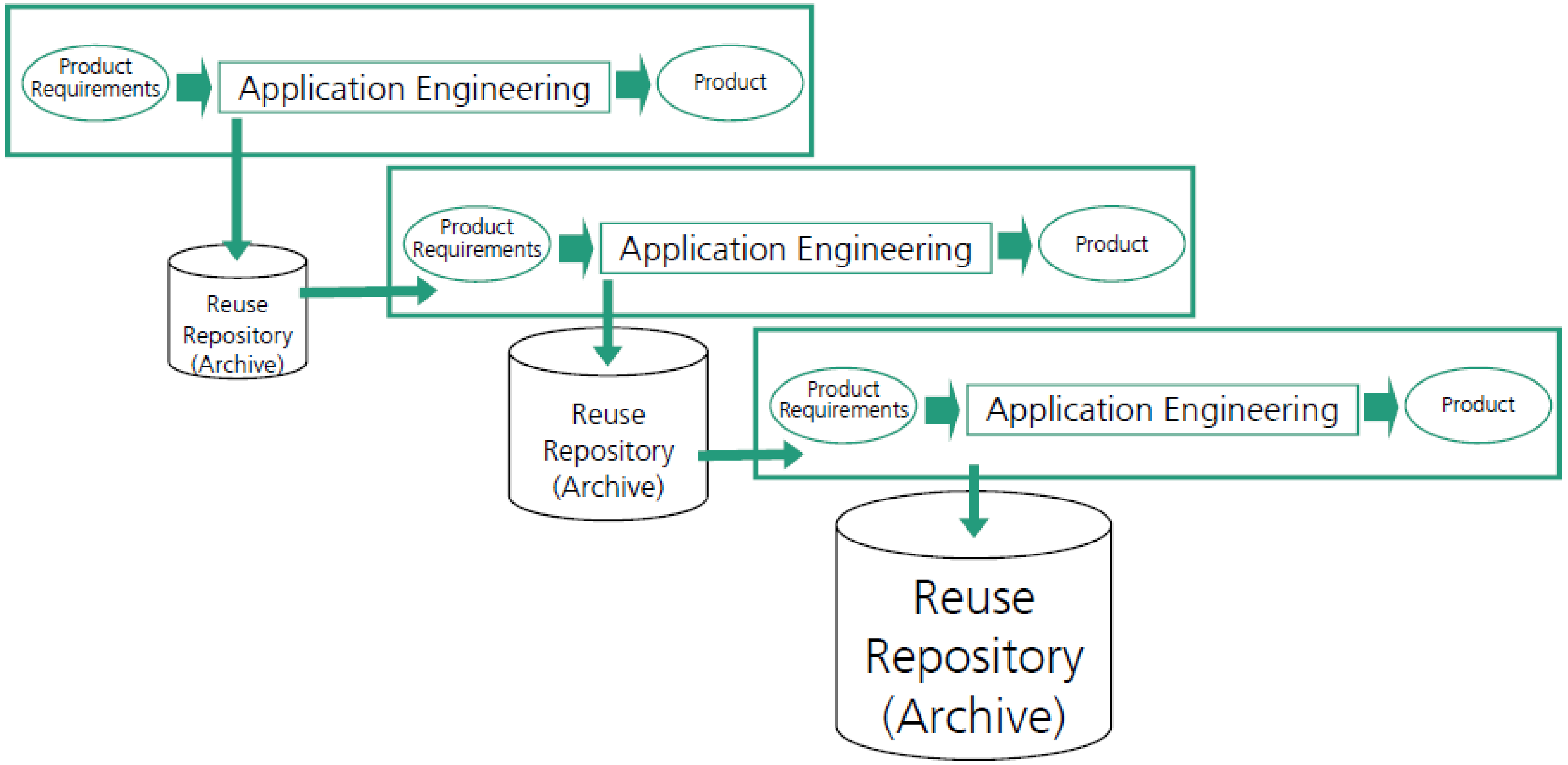
- Modularity
  → Necessary to be able to extract and insert components
- Loose coupling and high cohesion
  → Strong coupling and complex interdependencies hinder component extraction
- Information hiding
  → Hiding of internal details supports *plug-and-play* usage
- Separation of concerns
  → Clear mapping: solved problem and responsible components
  → Allows to identify and extract relevant components

Types of reuse

- Opportunistic / Ad-hoc reuse
  - Reuse of artifacts that initially were not meant and explicitly designed to be reused
  - Internal: multiple usage of own artifacts
  - External: usage of third-party artifacts
- Planned / Structured reuse
  - Systematic planning and development or reusable artifacts or adaptation of existing ones to make them reusable
  - Software product lines

In this lecture we focus on **planned / structured reuse**!

Typical form of ad-hoc reuse



Problems with ad-hoc reuse

Even though applied widely: *Clone-and-Own* paradigm

Ad-hoc reuse does not scale!

- Lack of structured means of organizing and managing artifacts
- Search effort
- Evaluation effort
- Adaptation effort (artifacts often need customization)
- Integration effort
- Clone detection and maintenance

Typical forms of structured reuse

Configurable software solutions

- Essential functionality is parameterized
- Can be customized by instantiating parameters
- Typically embedded in deployment routine
  → no code changes necessary

Software product lines

- “Software construction kits” with stable core architecture
- Managed variability modeling

(Software) Frameworks

- Planned development of reusable libraries of artifacts
- Typically, not stand-alone; meant to be integrated into other software
- Will discuss later

From requirements to system design

Variability subject and variability object

What does vary?

- A variability subject is a variable item of the real world or a variable property of such an item.

Why does it vary?

- Different stakeholder needs, different country laws, technical reasons, …

How does it vary?

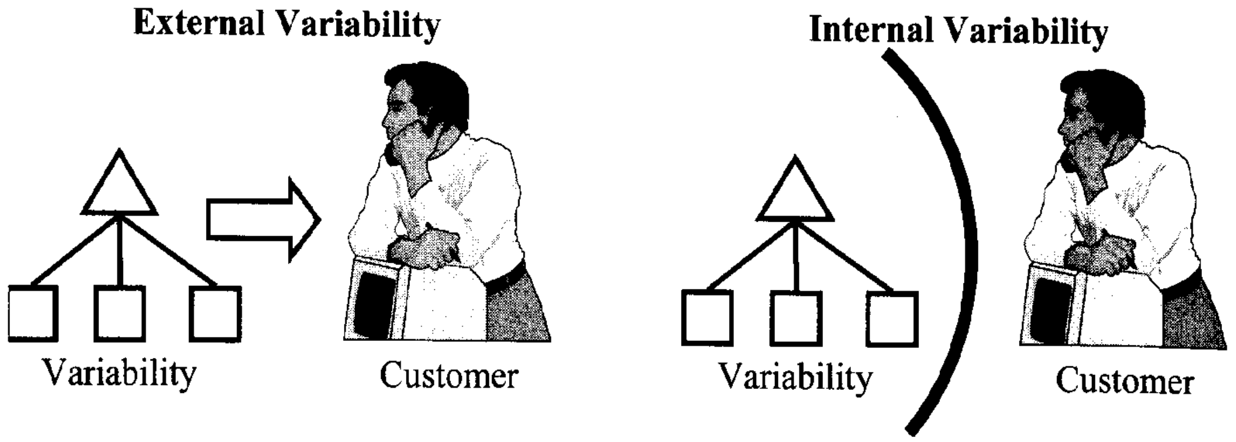
- A variability object is a particular instance of a variability subject.

Examples:

- "Color" is a variability subject of the real world. Examples of variability objects are red, green, blue, …
- "Payment method" is a variability subject and payment by credit card, payment by bill, payment by cash are examples of variability objects.

["Software Product Line Engineering." K. Pohl et al. (2005)]

Internal and external variability



External variability is the variability of domain artifacts that is visible to customers.

- Example: The customers of a home automation system can choose between different door lock identification mechanisms: keypad, magnetic card, and fingerprint scanners.
- Causes: Stakeholder needs, laws and standards

Internal variability is the variability of domain artifacts that is hidden from customers.

- Example: The communication protocol of a home automation system network offers two different modes.
- Causes: Refinement of external variability; technical reasons

["Software Product Line Engineering." K. Pohl et al. (2005)]

Managed variability

Variability is defined during *domain* engineering and is exploited during *application* engineering.

Defining and exploiting variability is supported by the concept of **managed variability**:

- Supporting variability concerned with defining variability
- Managing variable artifacts
- Supporting activities concerned with resolving variability
- Collecting, storing and managing trace information is necessary to fulfill these tasks

The moment of variability resolution is called the **binding time**.

["Software Product Line Engineering." K. Pohl et al. (2005)]

Motivation: Variability should be modeled explicitly

Variability can be defined either

- as an **integral part of development artifacts** or
- in a **separate variability model**.

Shortcomings of modeling variability within the traditional software development models:

- If variability is spread, it is almost impossible to keep the information consistent.
- It is hard to determine, which variability information in requirements has influenced which variability information in design, realization, or test artifacts.
- The software development models are already complex and get overloaded by adding the variability information.
- The concepts used to define variability differ between the different kinds of software development models.

The variability defined in different models does not integrate well into an overall picture of the software variability.

["Software Product Line Engineering." K. Pohl et al. (2005)]

Orthogonal variability

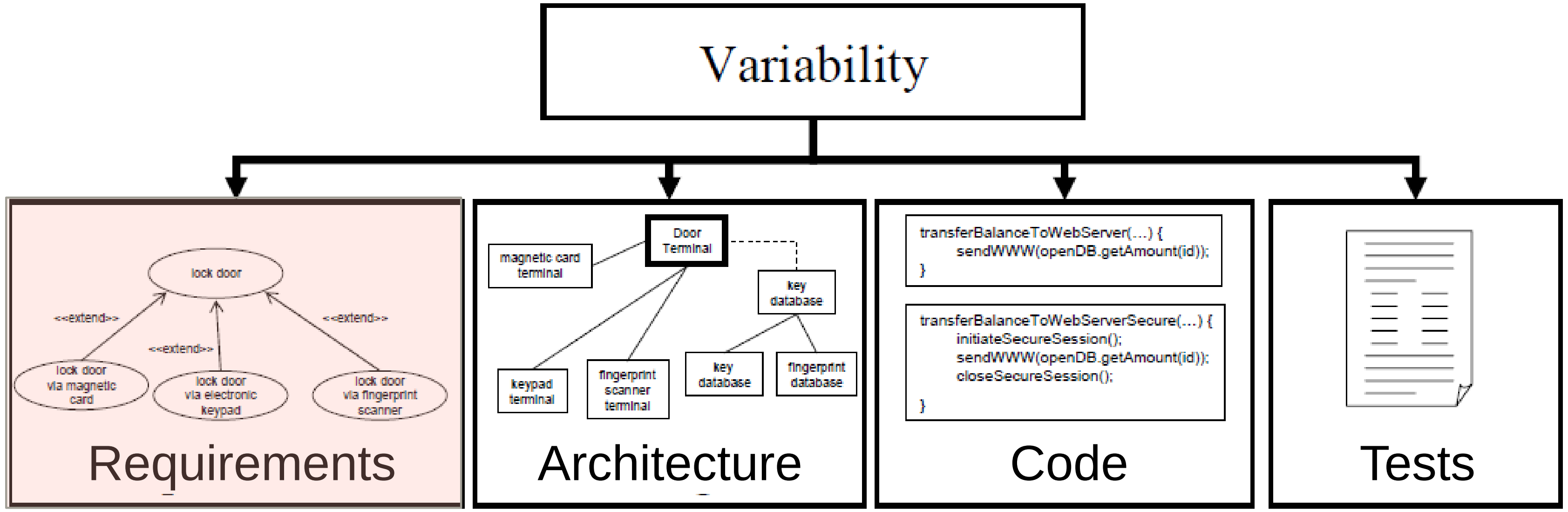
Define the variability information in a separate “orthogonal variability model”.

An **orthogonal** variability model is a model that defines the variability of a software product line. It relates the variability defined to other software development models such as feature model, use case models, design models, component models, and test models.

["Software Product Line Engineering." K. Pohl et al. (2005)]

Variability management

Variability defined in the variability model has to be related to software artifacts specified in other models, textual documents, and code.



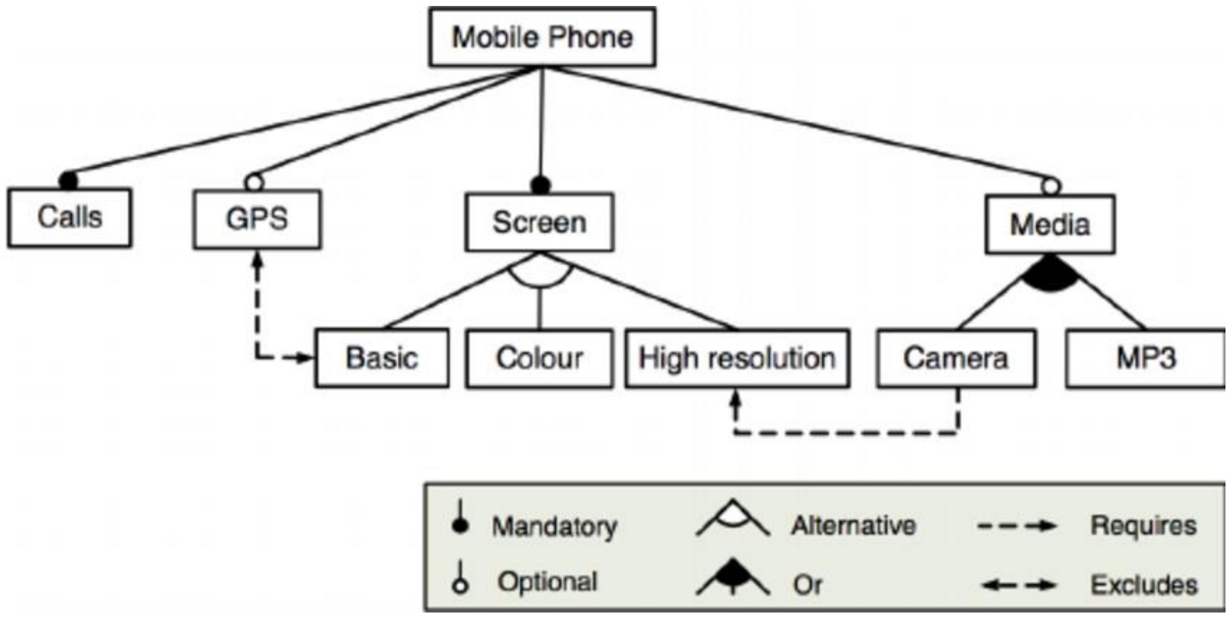
```
transferBalanceToWebServer(…) {
        sendWWW(openDB.getAmount(id));
}

transferBalanceToWebServerSecure(…) {
        initiateSecureSession();
        sendWWW(openDB.getAmount(id));
        closeSecureSession();
}
```

Feature diagrams

Variability is typically modeled as a **feature diagram:**

- Depict variation points and different variants
- Variation points can be mandatory or optional
- Variation points can have interdependencies
  - One variant requires another one to be selected
  - One variant excludes another one from being selected
- Variation points can allow a Boolean combination of variants (e.g., AND, OR, XOR)

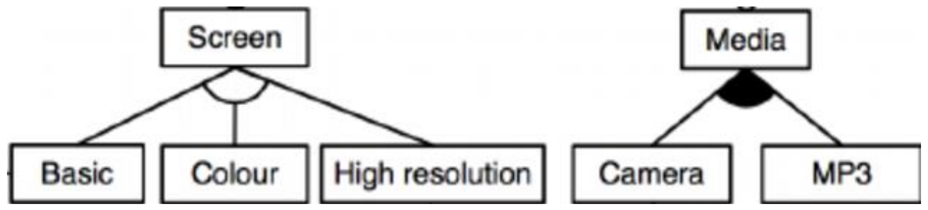


Variation points and variants

A **variation point** is a representation of a variability subject within domain artifacts enriched by contextual information.

A **variant** is a representation of a variability object within domain artifacts.

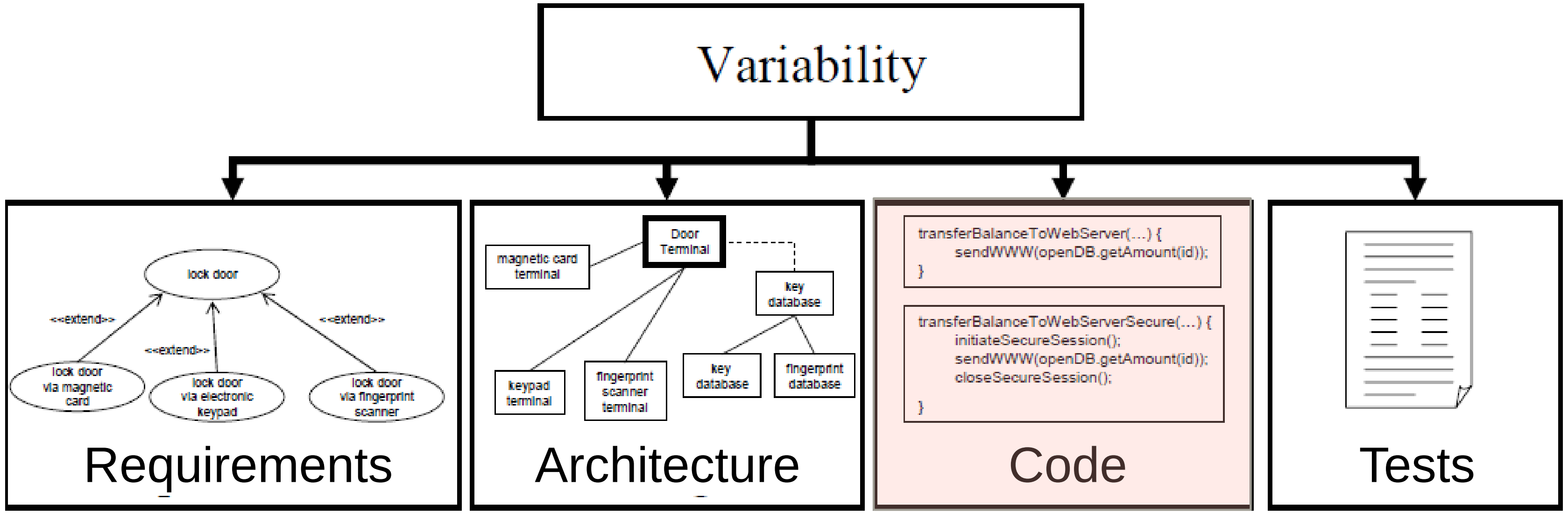
variation points



variants

Variability management

Variability defined in the variability model has to be related to software artifacts specified in other models, textual documents, and code.



```
transferBalanceToWebServer(…) {
        sendWWW(openDB.getAmount(id));
}
```

```
transferBalanceToWebServerSecure(…) {
        initiateSecureSession();
        sendWWW(openDB.getAmount(id));
        closeSecureSession();

}
```

Variability at the code level

Variability at the code level can be supported by different programming paradigms:

- **Conditional compilation**
- Polymorphism
- Aspect-oriented programming

Variability at the code level

Conditional compilation is typically done with in-code preprocessor instructions (e.g., with GPP for C/C++ code).

Basic idea:

- Decouple common from variable code
- Automatically include or exclude variable code from compilation

Advantages:

- Variable parts are emphasized → easy to identify and manage
- Variable parts may cross-cut syntactic borders (e.g., function boundaries)
  → not possible with adding new modules

Disadvantages:

- Constrains application engineers to pre-defined variants
- Variant combinations may result into inconsistencies / faulty code

Variability at code level – example

T9

```
class Message {
public:
#ifdef T9_SUPPORTED
  void checkWordList() {...}
#endif
#ifdef ATTACH_SUPPORTED
  void enableAttachButton() {...}
#endif
};
class MessageUI {
public:
  void edit(Message &msg) {
#ifdef T9_SUPPORTED
    if(t9Active) tr.checkWordList();
#endif
    // perform editing
#ifdef ATTACH_SUPPORTED
    tr.enableAttachButton();
#endif
}};
```

Attachment

One possible instantiation:

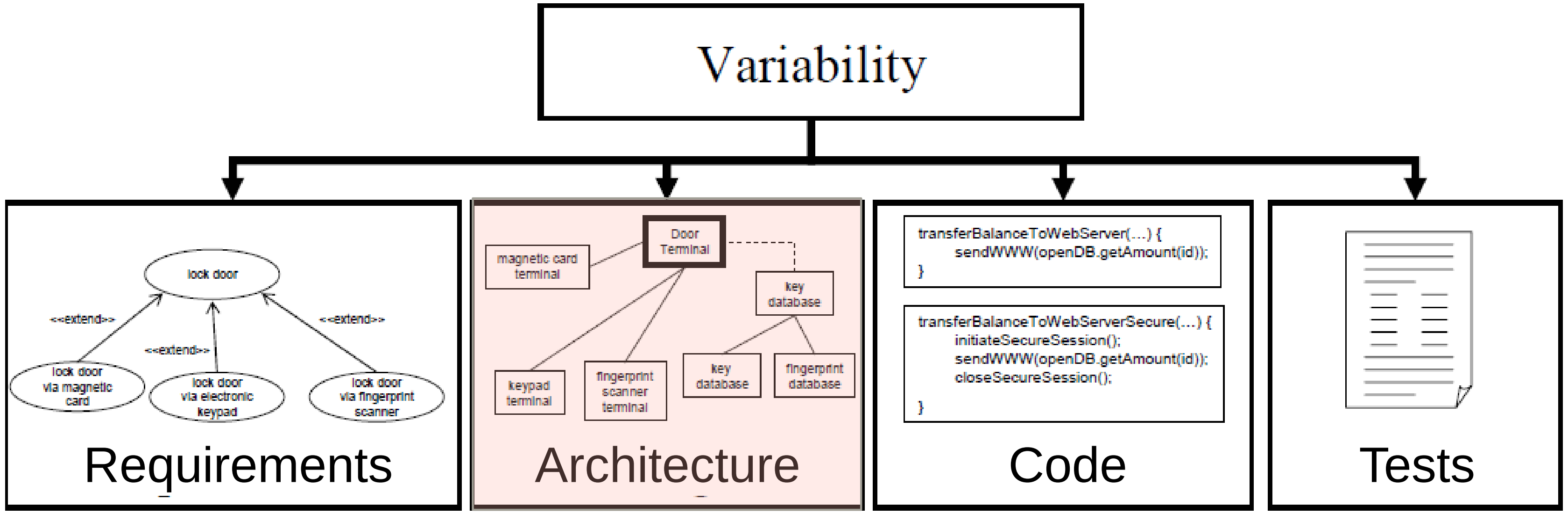
```
#define T9_SUPPORTED
#undef  ATTACH_SUPPORTED
```

```
class Message {
public:
   void checkWordList() {...}
};
class MessageUI {
public:
  void edit(Message &msg) {
    if(t9Active)
      msg.checkWordList();
}};
```

From requirements to system design

Variability management

Variability defined in the variability model has to be related to software artifacts specified in other models, textual documents, and code.



```
transferBalanceToWebServer(...) {
        sendWWW(openDB.getAmount(id));
}
```

```
transferBalanceToWebServerSecure(...) {
        initiateSecureSession();
        sendWWW(openDB.getAmount(id));
        closeSecureSession();
}
```

Variability at the architecture level

- Variability at the architecture level handled with special **Product Line Architectures** (PLA).
- A product line architecture is a generic software architecture for a product line that embodies the architectures for all product line members.
- A product line architecture differs from a single system architecture in that it has to cover additional concerns:
  - What are the common parts of the architecture?
  - What are the variable parts?
  - How does a particular instance architecture look like?
  - How can it be ensured that all intended product line members are indeed supported by the architecture?

Software product line engineering

Engineering of variant-rich software / system families

Software product lines

A software product line is a set of applications with a common architecture and shared components, with each application specialized to reflect different requirements.

Adaptation may involve:

- Component and system configuration
- Adding new components to the system
- Selecting from a library of existing components
- Modifying components to meet new requirements

"Software product family" and "software product line" are used almost synonymously.

["Software Engineering." Ian Sommerville. (2010)]

Software product line engineering

Software **P**roduct **L**ine **E**ngineering (PLE)

- Focus: proactive development for reuse
- Adoption of traditional product line approaches as used, for example, in automotive engineering.
  → Development based on customization of platforms.
- Building applications for mass customization through managed variability to systematically model commonalities and the differences in software applications.

- Separation into a *Family (Domain) Engineering* process to establish a platform with reusable assets and an *Application Engineering* process to instantiate and customize this platform to concrete products.

Specialization of a software product line

Platform specialization

- Different versions of the application are developed for different platforms.

Environment specialization

- Different versions of the application are created to handle different operating environments e.g. different types of communication equipment.

Functional specialization

- Different versions of the application are created for customers with different requirements.

Process specialization

- Different versions of the application are created to support different business processes.

["Software Engineering." Ian Sommerville. (2010)]

Why product line engineering?

- Reference architectures (next!) support reuse but mainly focus on technical aspects of software development.
- For customers, technical aspects are of minor importance
  → Functionality matters here!
- Functionality lifecycles are short (1-3 years)
  → Examples: tax management, accounting, …
- Technology lifecycles are comparably long (5-15 years)
  → E.g., many financial institutions still run COBOL software.
- PLE focusses on functionality aspects by moving a classical development approach to the development of applications based on pre-built, configurable components.

Why product line engineering?

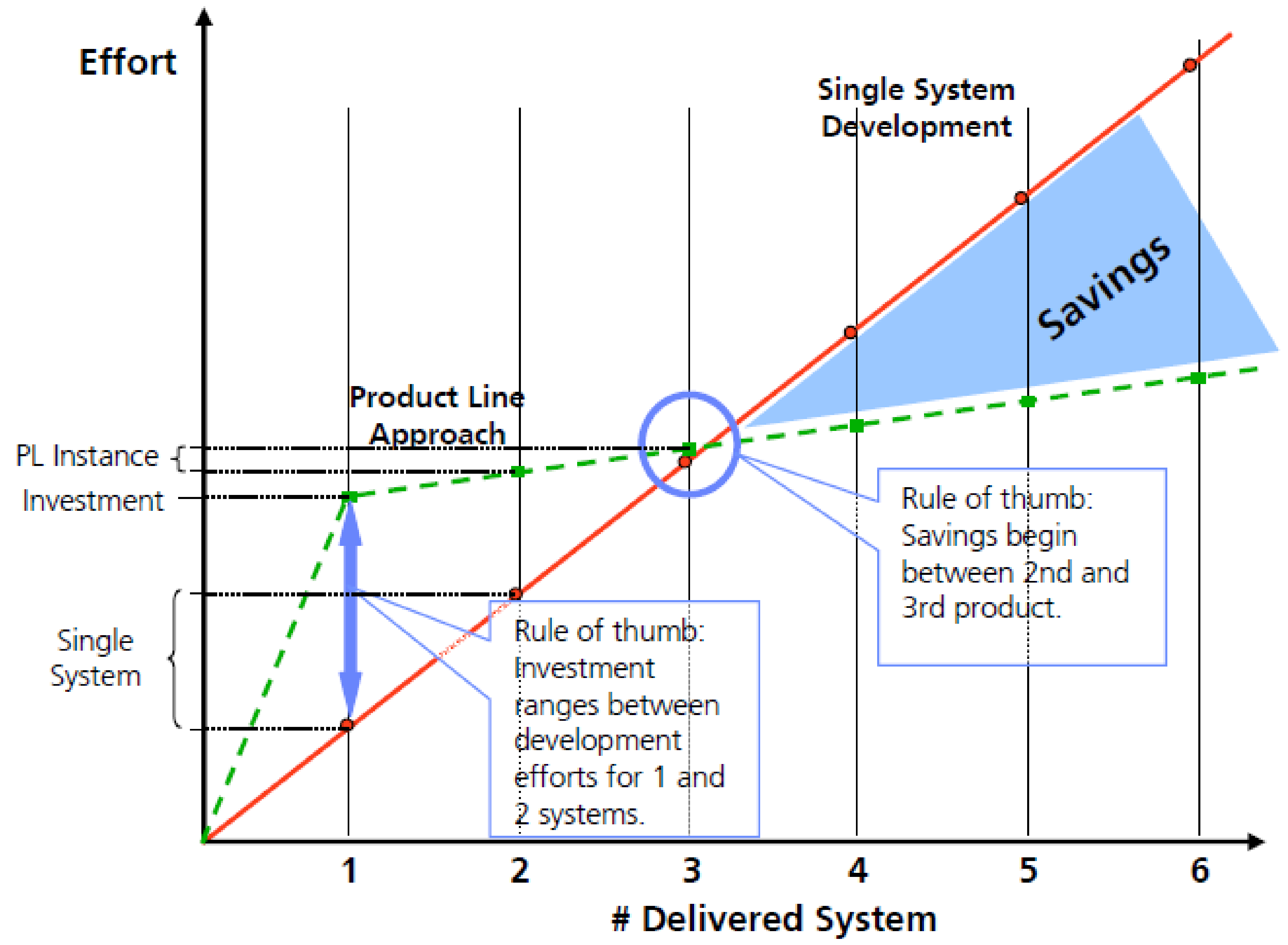
- Strategic success factors for software development:
  - Cost
  - Quality
  - Time-to-market!!!

- PLE aims at optimizing them with special emphasis on time-to-market.

- Increasing reuse means reducing necessity of re-development and customization
  → Reduces costs and time-to-market

- Repository of stable and well-tested components
  → increases quality

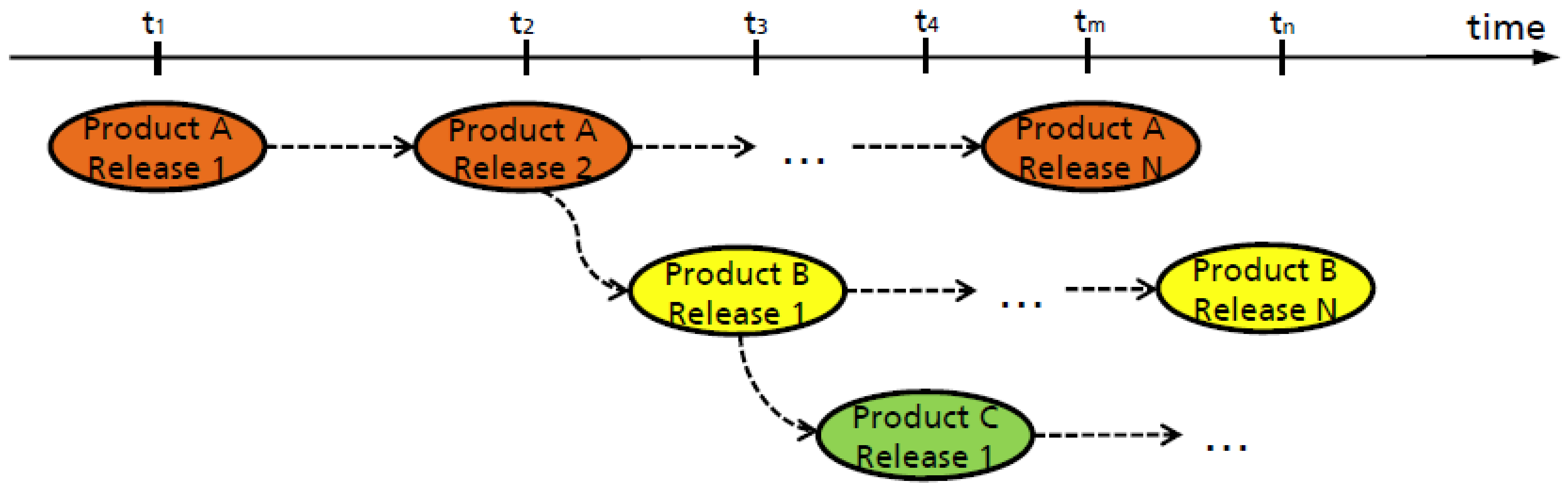
Cost reduction with PLE



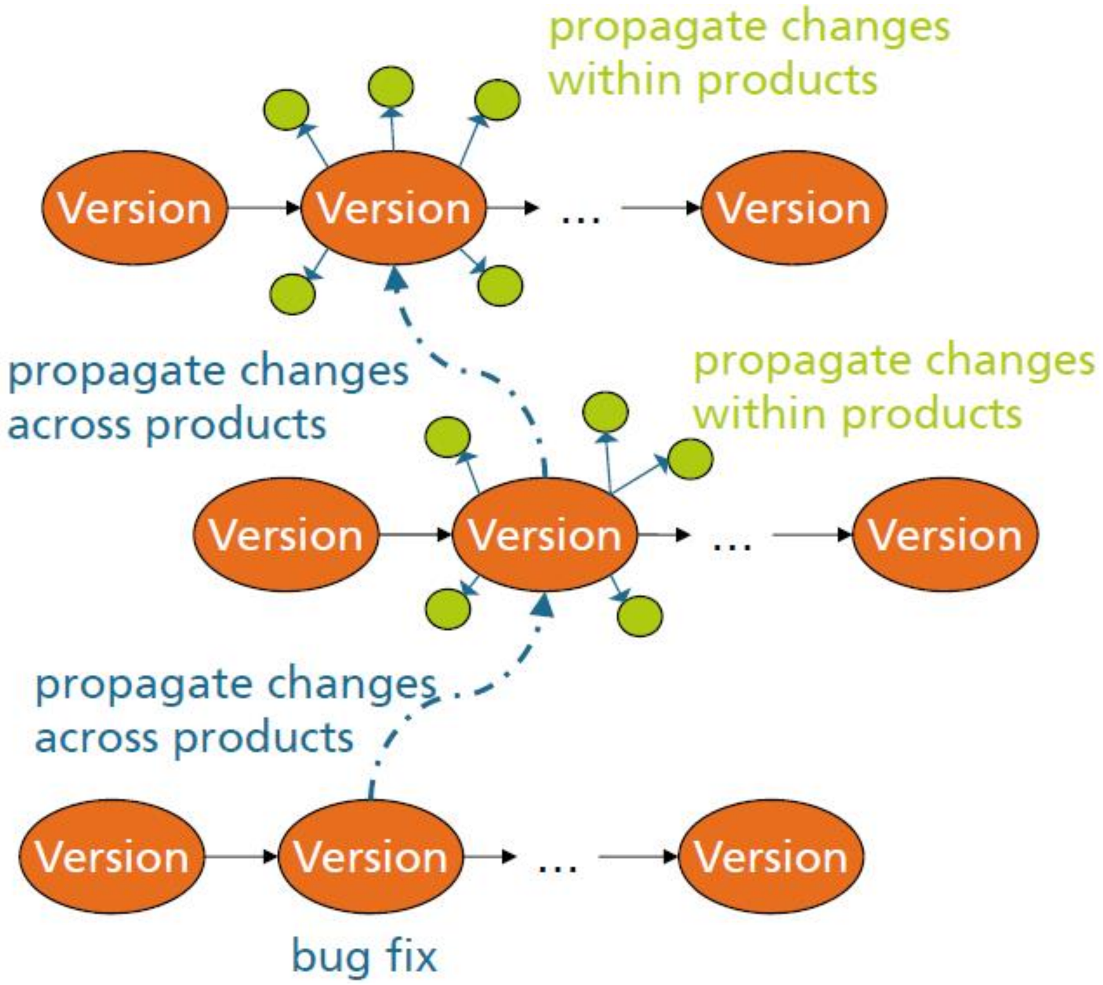
PLE: Tackled problem

Software companies rarely develop one single product

- Often develop multiple products in a certain business area
- Products “grow” over time
- Development of multiple **similar but not identical** products
- Development from scratch is unproductive

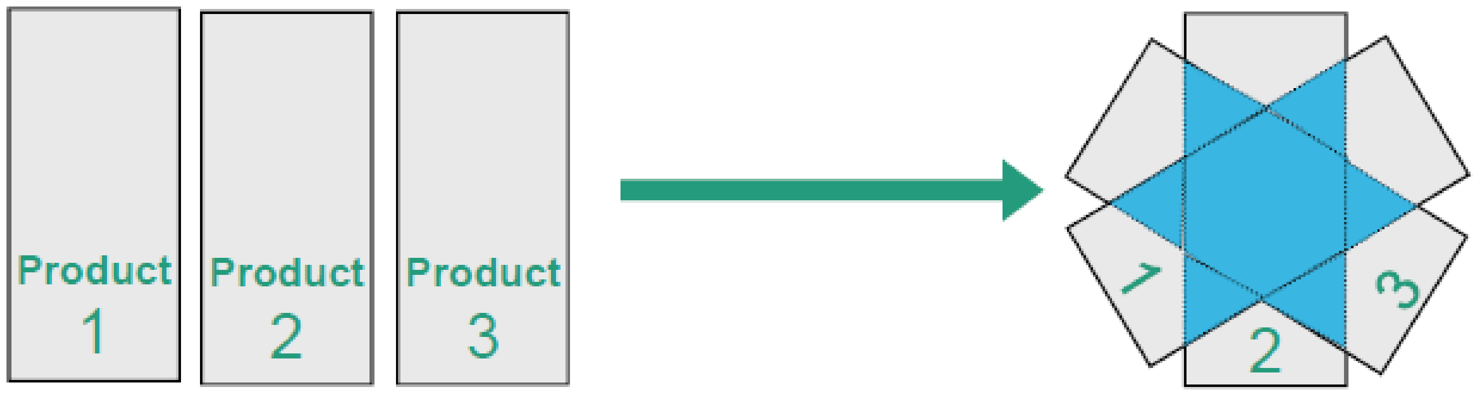


PLE: Tackled problem

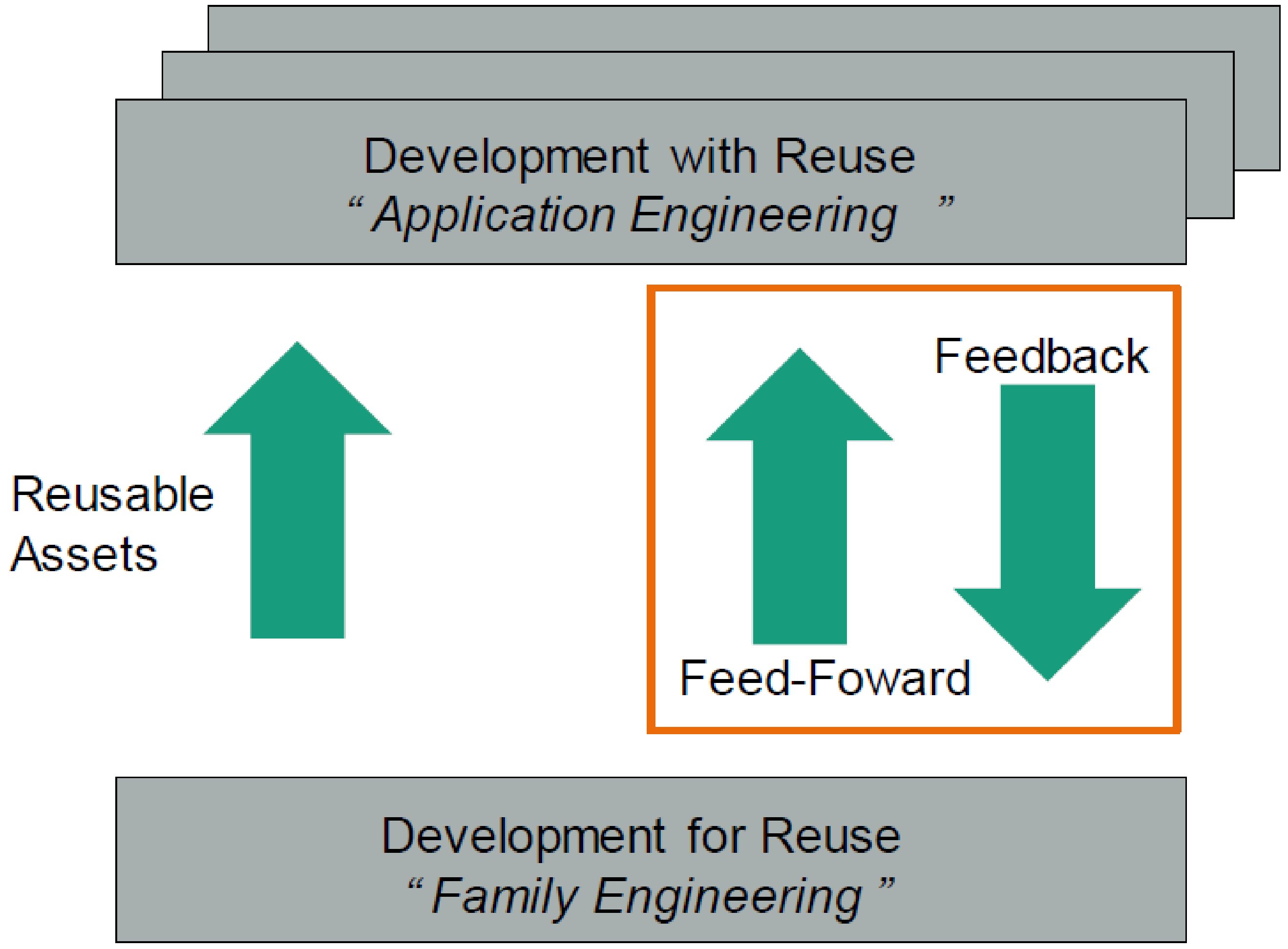

propagate changes within products
Version
Version
...
Version
propagate changes across products
propagate changes within products
Version
Version
...
Version
propagate changes across products
Version
Version
...
Version
bug fix

Optimizing reuse with PLE

- Consider all products of an organization within a specific domain as *Product Family* or *Product Line*
- Take advantage of commonalities
- Strategic planning of variability
- Goal: Support just the right amount of variability

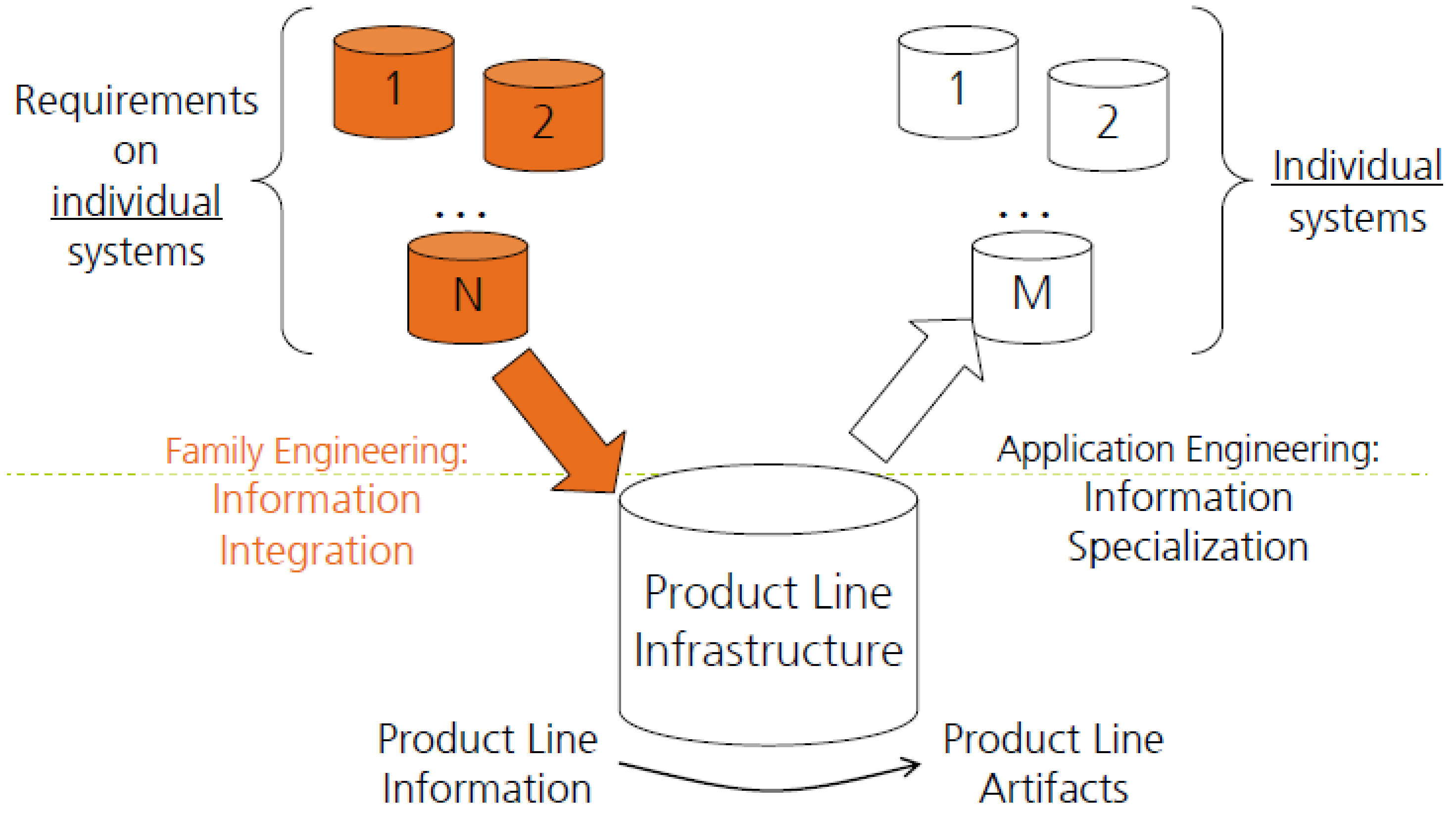


PLE: managed proactive reuse

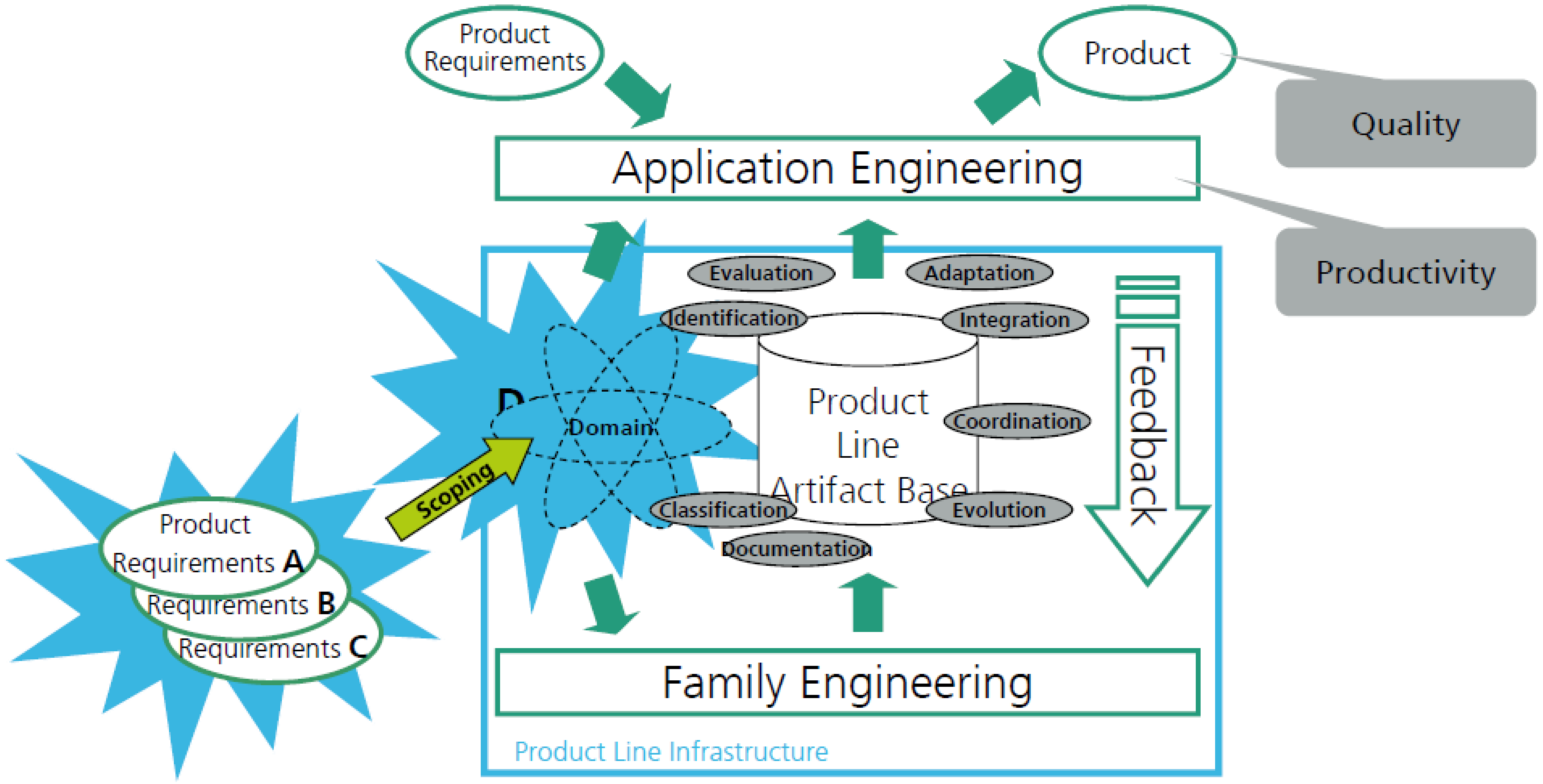


Family engineering is also called domain engineering

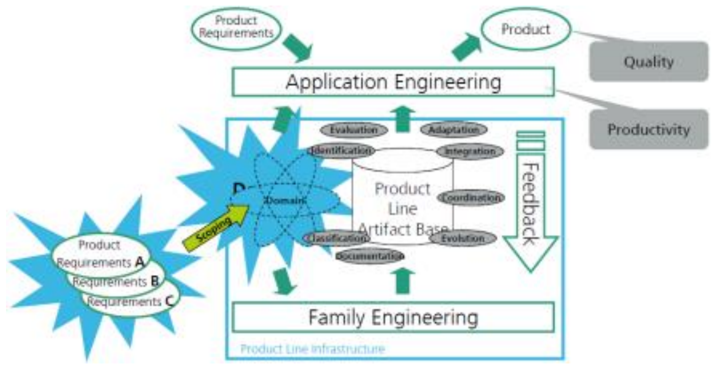
PLE: Development approach


Requirements on individual systems
1
2
...
N
1
2
...
M
Individual systems
Family Engineering: Information Integration
Application Engineering: Information Specialization
Product Line Infrastructure
Product Line Information
Product Line Artifacts

Product line's lifecycle



Product line's lifecycle



Family/Domain engineering

- Identification of product commonalities
  → Scoping
- Leads to definition of reusable (product line) artifacts
- Only consider economical reasonable variabilities

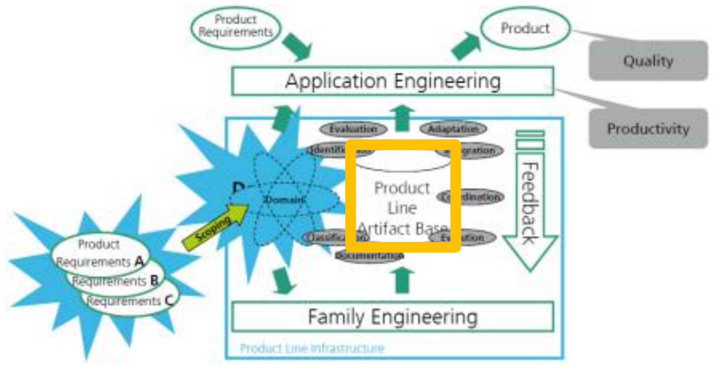
Application engineering

- Development of individual products based on the PL artifact base
- Instantiate the defined variabilities according to specific product requirements

Product line artifact base

- Acts as a repository for reusable artifact of a product line
- Product line artifacts cover both: variability and commonality

Product line artifact base



- Captures characteristics of product line artifacts

Content:

- Product model, process model, resources

Lifecycle Phase:

- Requirements, system design, unit design, code, image, data, test, integration, documentation, configuration, patch

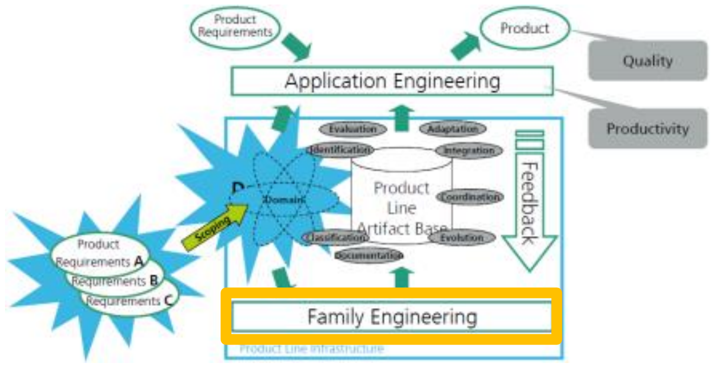
Granularity:

- Subsystem, component, folder, document, document fragment / element

Data Type:

- Model, structured text (e.g., XML), text, binary

Family/Domain engineering



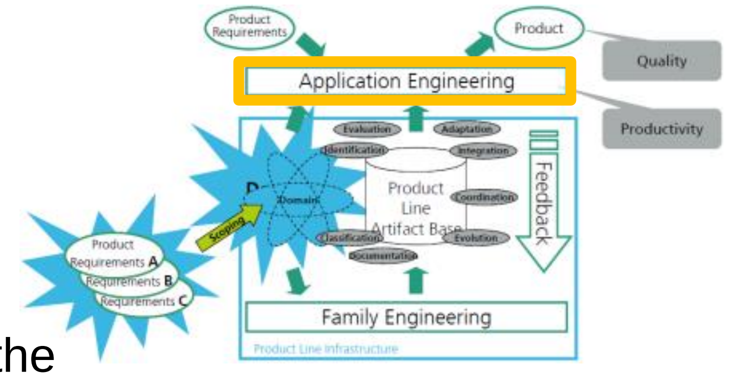
Key goals of the Family/Domain engineering process:

- Define the commonality and the variability of the software product line
- Define the set of applications the software product line is planned for, i.e., define the **scope** of the software product line
- Define and construct reusable artifacts that accomplish the desired variability

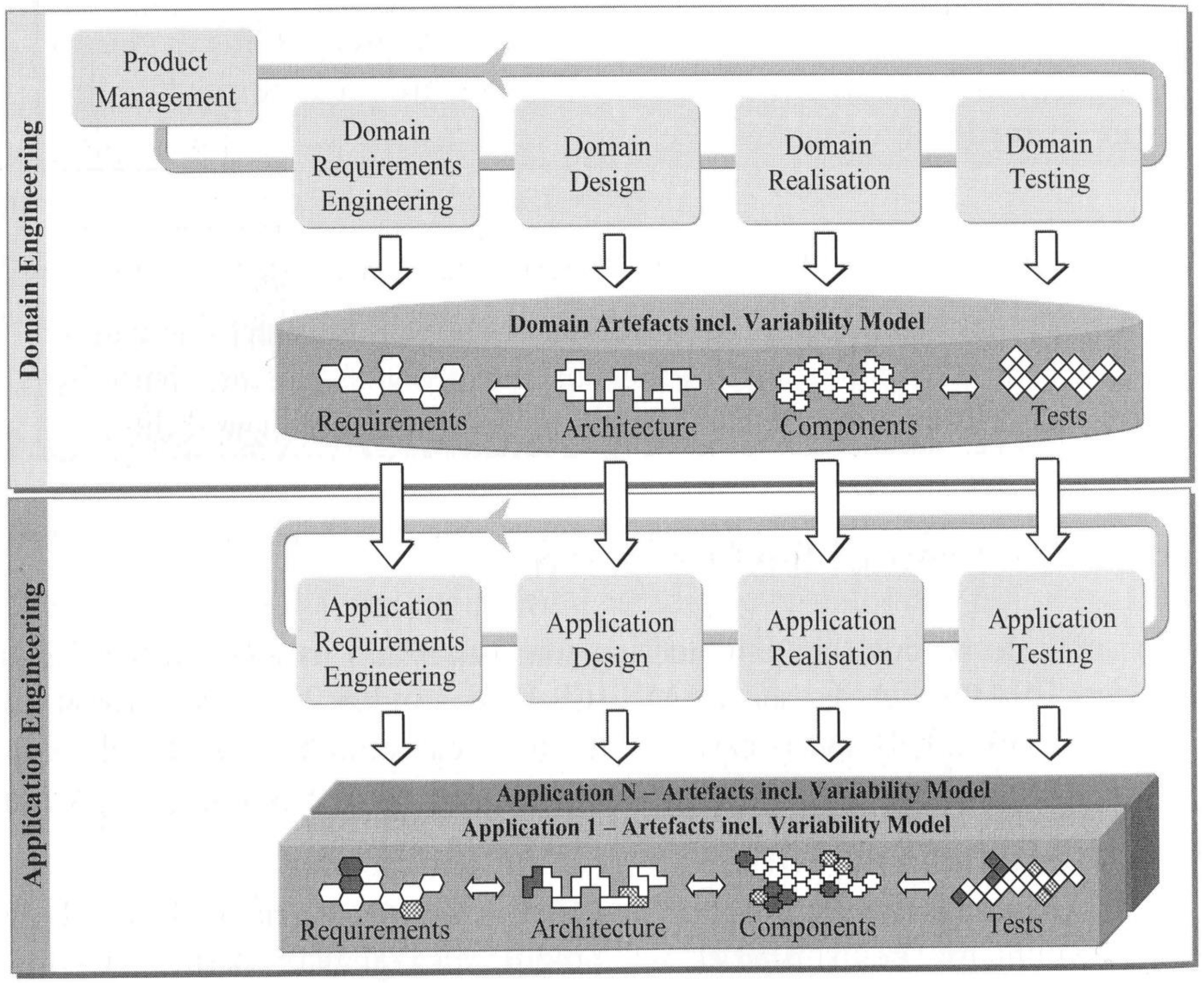
Application engineering

Key goals of the application engineering process:

- Achieve highest possible reuse of domain assets when developing product line applications.
- Exploit commonality and the variability of software product line during the product line application development.
- Document application artifacts, e.g., requirements, architecture, components, and tests, and relate them to the domain artifacts.
- Bind variability according to application needs from requirements over architecture, to components, and test cases.
- Estimate impacts of the differences between application and domain requirements on architecture, components and tests.



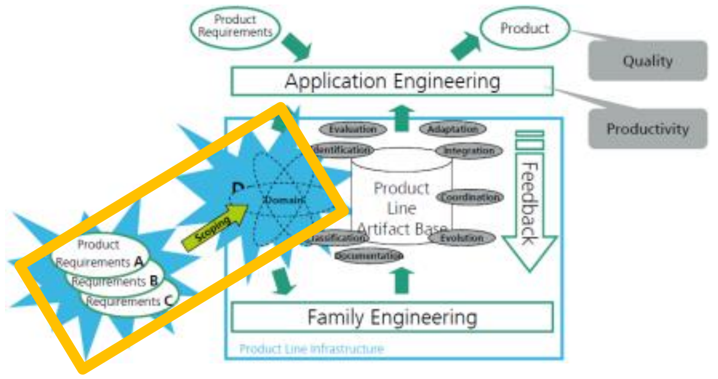
A framework for software product line engineering



["A framework for software product line engineering." G. Böckle et al. (2005)]

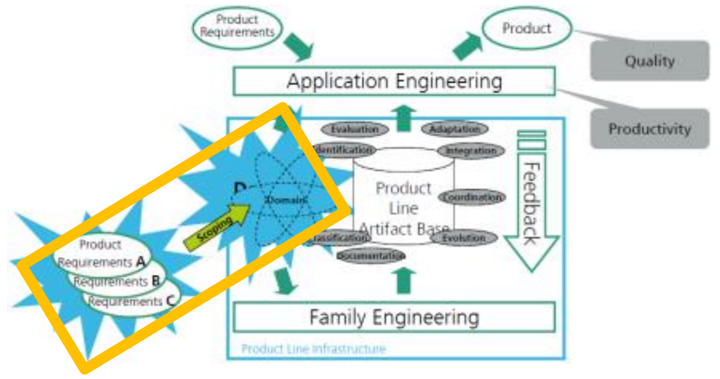
Scoping (1)

- Grouping according to domains imposes some problems
- Domains may be extremely large with fuzzy boundaries
- Problematic to identify relevant variabilities and envisioned / to be supported features
- PLE introduces the idea of **scoping** to define sharp boundaries based on concrete product requirements
- Scoping tackles
  - Existing products
  - Competitor products
  - Future or envisioned products



Scoping (2)

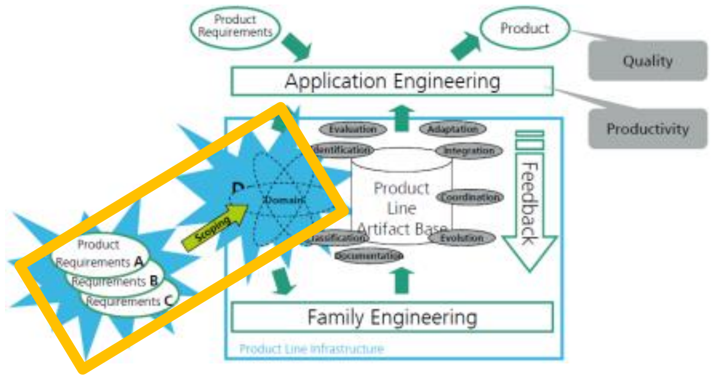
“Scoping is the process of identifying and bounding areas (subdomains, existing assets) and capabilities (features) of the product line where investment into reuse is economically useful and beneficial to product development”.



- Considering all requirements and thus variabilities of a domain is most often not feasible and in general not economical.
- Scoping helps to identify and narrow down domain specificities and commonalities to a meaningful subset.

Scoping (3)

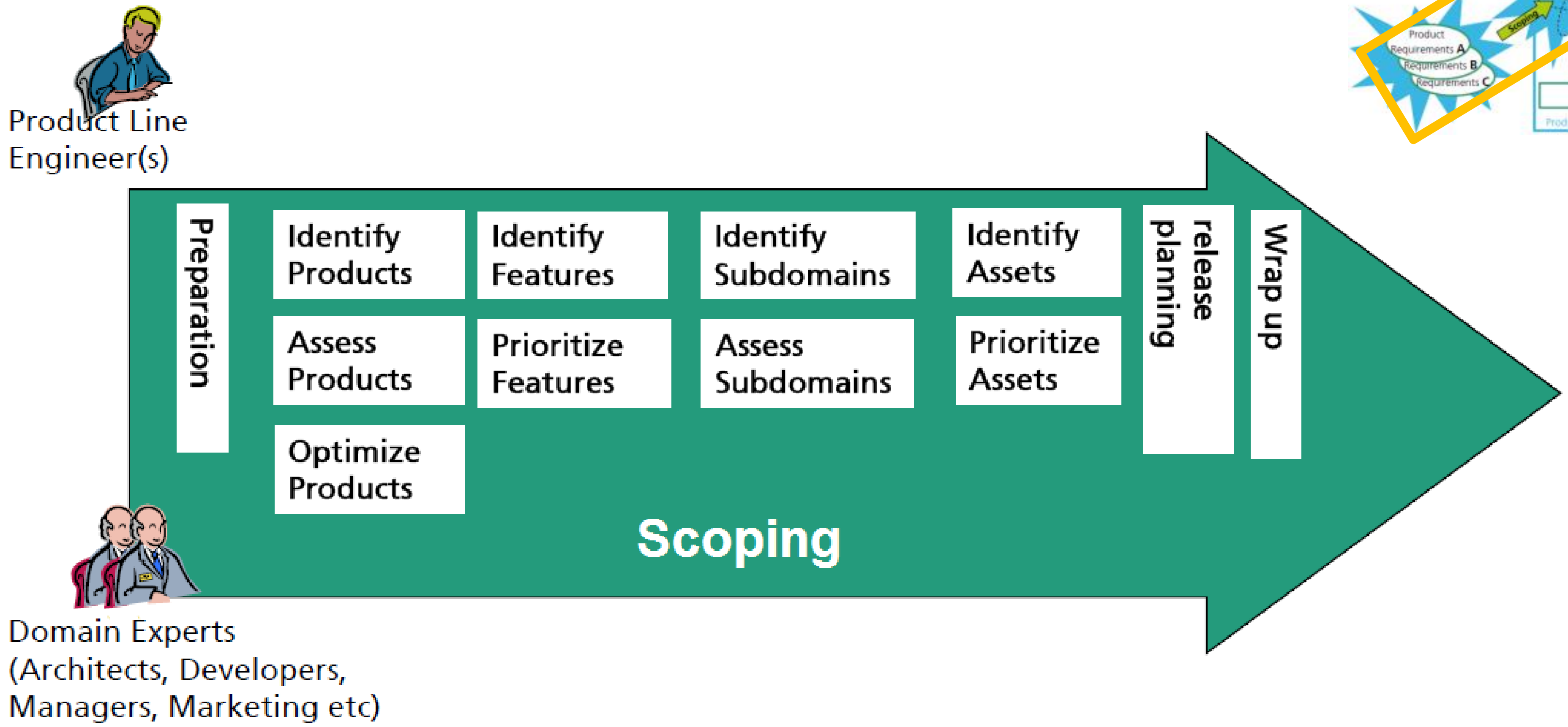
Aspects to consider

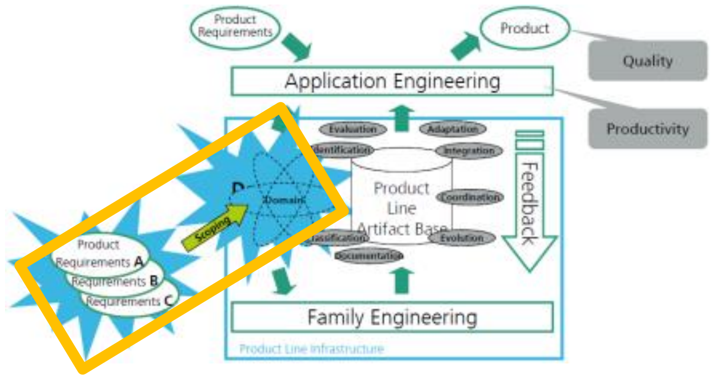


- **Products**
  - Which products do I want to have in my product line?
  - What is their market, when will they be released?
- **Domains**
  - Which subdomains will my product line have?
  - What information do they carry?
  - What are “good”, what are “bad” domains (in terms of knowledge, stability etc.)?
- **Features**
  - Which features will my product line have?
  - Which product will have what kind of features?
  - Which are easy, which are risky features?
- **Assets**
  - Which assets do I have in my product line?
  - Which reusable artifacts already exist, which ones do I have to (re-)implement?

Scoping (4)

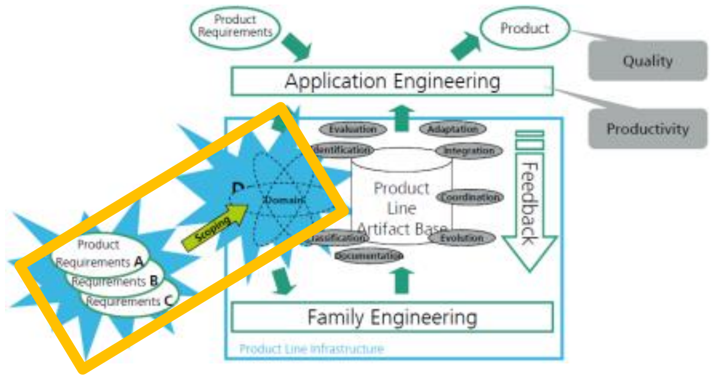
Process

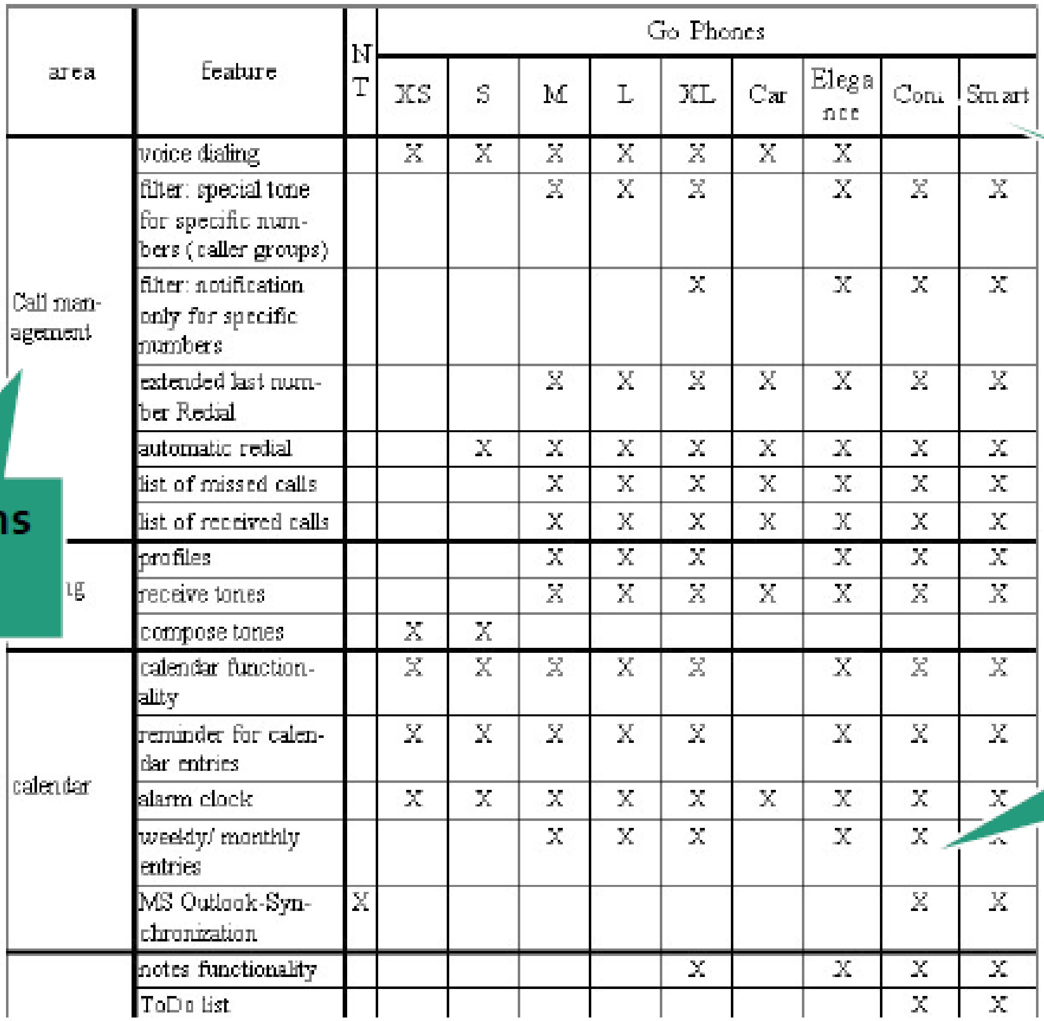




Scoping (5)

Product feature matrix





area	feature	NT	Go Phones								
			XS	S	M	L	XL	Car	Elegance	Com	Smart
Call management	voice dialing		X	X	X	X	X	X	X		
	filter: special tone for specific numbers (caller groups)				X	X	X		X	X	X
	filter: notification only for specific numbers						X		X	X	X
	extended last number Redial				X	X	X	X	X	X	X
	automatic redial			X	X	X	X	X	X	X	X
	list of missed calls				X	X	X	X	X	X	X
	list of received calls				X	X	X	X	X	X	X
ng	profiles				X	X	X		X	X	X
	receive tones				X	X	X	X	X	X	X
	compose tones		X	X							
calendar	calendar functionality		X	X	X	X	X		X	X	X
	reminder for calendar entries		X	X	X	X	X		X	X	X
	alarm clock		X	X	X	X	X	X	X	X	X
	weekly/ monthly entries				X	X	X		X	X	X
	MS Outlook-Synchronization	X								X	X
	notes functionality						X		X	X	X
	ToDo list									X	X

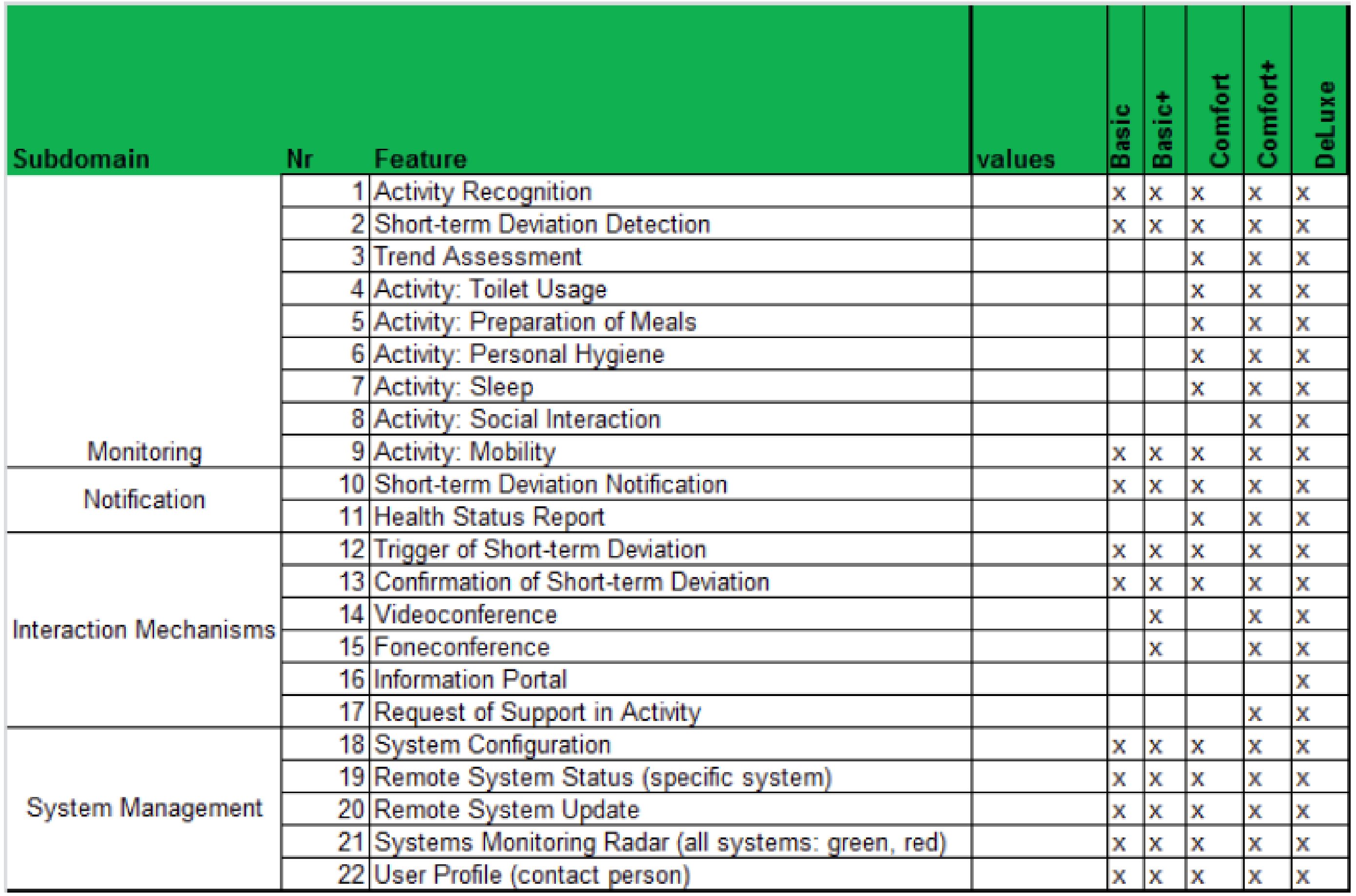
Products

Sub-domains

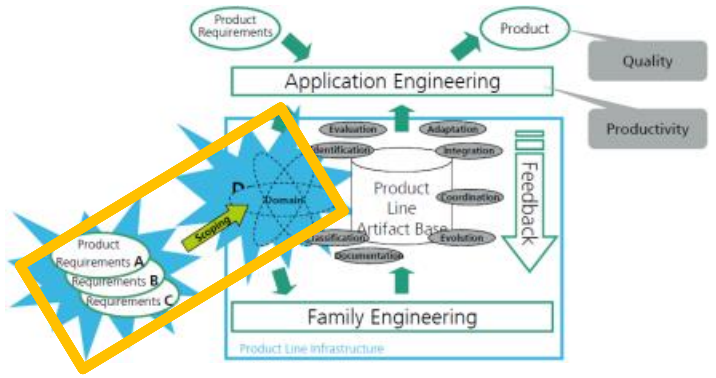
Feature is supported by a product?

Scoping (6)

Product feature matrix



Subdomain	Nr	Feature	values	Basic	Basic+	Comfort	Comfort+	DeLuxe
Monitoring	1	Activity Recognition		x	x	x	x	x
	2	Short-term Deviation Detection		x	x	x	x	x
	3	Trend Assessment				x	x	x
	4	Activity: Toilet Usage				x	x	x
	5	Activity: Preparation of Meals				x	x	x
	6	Activity: Personal Hygiene				x	x	x
	7	Activity: Sleep				x	x	x
	8	Activity: Social Interaction					x	x
	9	Activity: Mobility		x	x	x	x	x
Notification	10	Short-term Deviation Notification		x	x	x	x	x
	11	Health Status Report				x	x	x
Interaction Mechanisms	12	Trigger of Short-term Deviation		x	x	x	x	x
	13	Confirmation of Short-term Deviation		x	x	x	x	x
	14	Videoconference			x		x	x
	15	Foneconference			x		x	x
	16	Information Portal						x
	17	Request of Support in Activity					x	x
System Management	18	System Configuration		x	x	x	x	x
	19	Remote System Status (specific system)		x	x	x	x	x
	20	Remote System Update		x	x	x	x	x
	21	Systems Monitoring Radar (all systems: green, red)		x	x	x	x	x
	22	User Profile (contact person)		x	x	x	x	x



PLE: Success stories

- Nokia was able to increase the production of different phone model from 4 (without PLE) to 25 (with PLE)
- Cummins Inc. was able to reduce the software development time for new diesel engines from one year (without PLE) to one week (with PLE)
- Hewlett-Packard was able to reduce the time-to-market by a factor of 7 thanks to PLE

Case study

GoPhone Inc. hypothetical mobile phone company is about to start business in the mobile phone market
→ Decided to establish a Product Line for development

GoPhone wants to enter the market in different segments

- Basic phones
  - Only essential features (small low-res display, dial pad, …)
- Communicators
  - Combines PDA and phone (small keyboard, bigger screen, …)
- Smart phones
  - Up-to-date technology (big touch-screen, powerful CPU+RAM)

Case study

Scoping I

Analyzing and describing anticipated *Product Portfolio*

Example: GoPhone XS

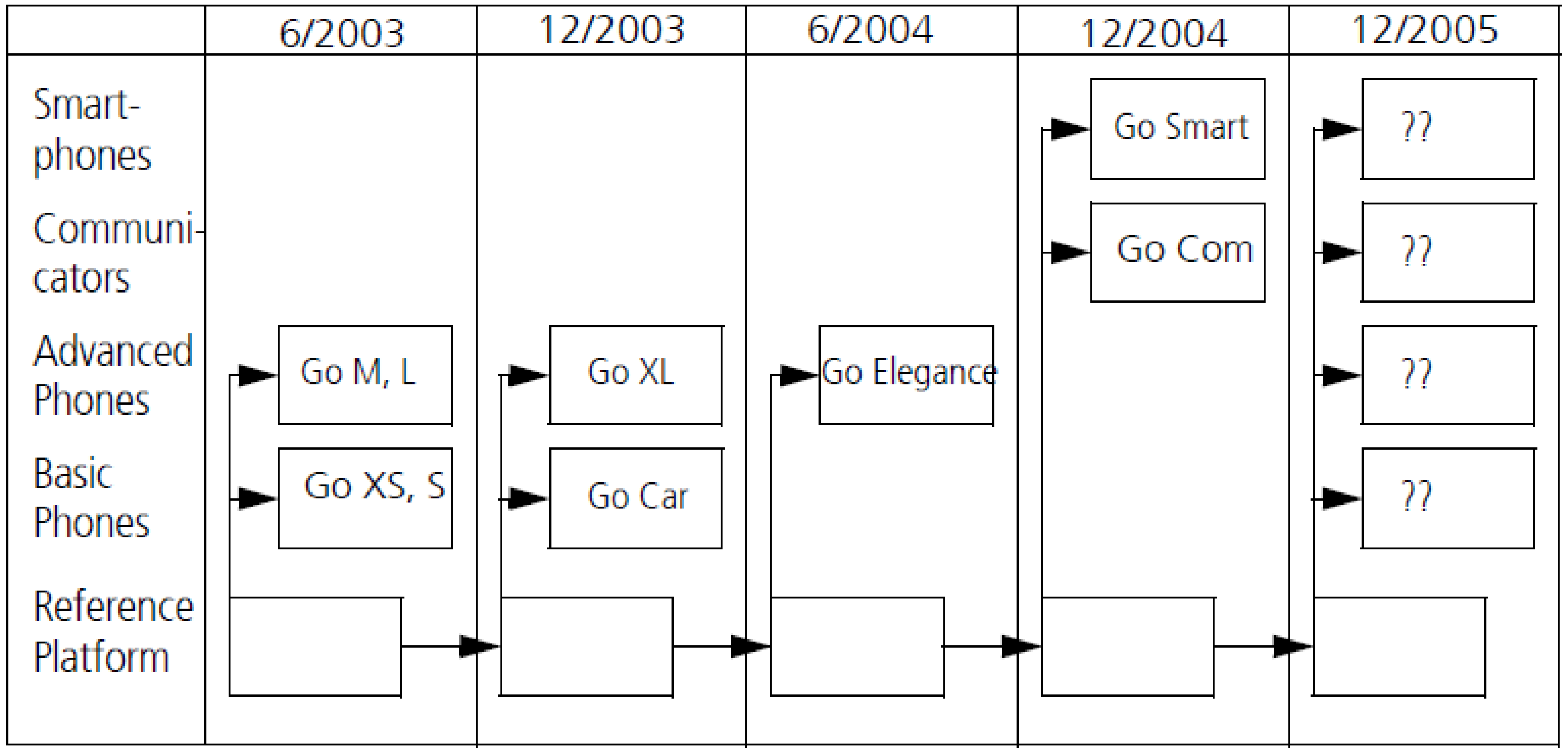
- Major functionality
  - Basic messaging + T9; basic calendar; basic voice dial
- Market segment
  - Low price; low-demand end-users; Segment: Basic phones
- System customization
  - Language (DE, EN, IT, …); network (GSM 900/1800, UMTS)
- Non-functional constraints
  - Time to menu switch 0.1 sec; MTBF < 6 month

…

Case study

Scoping II

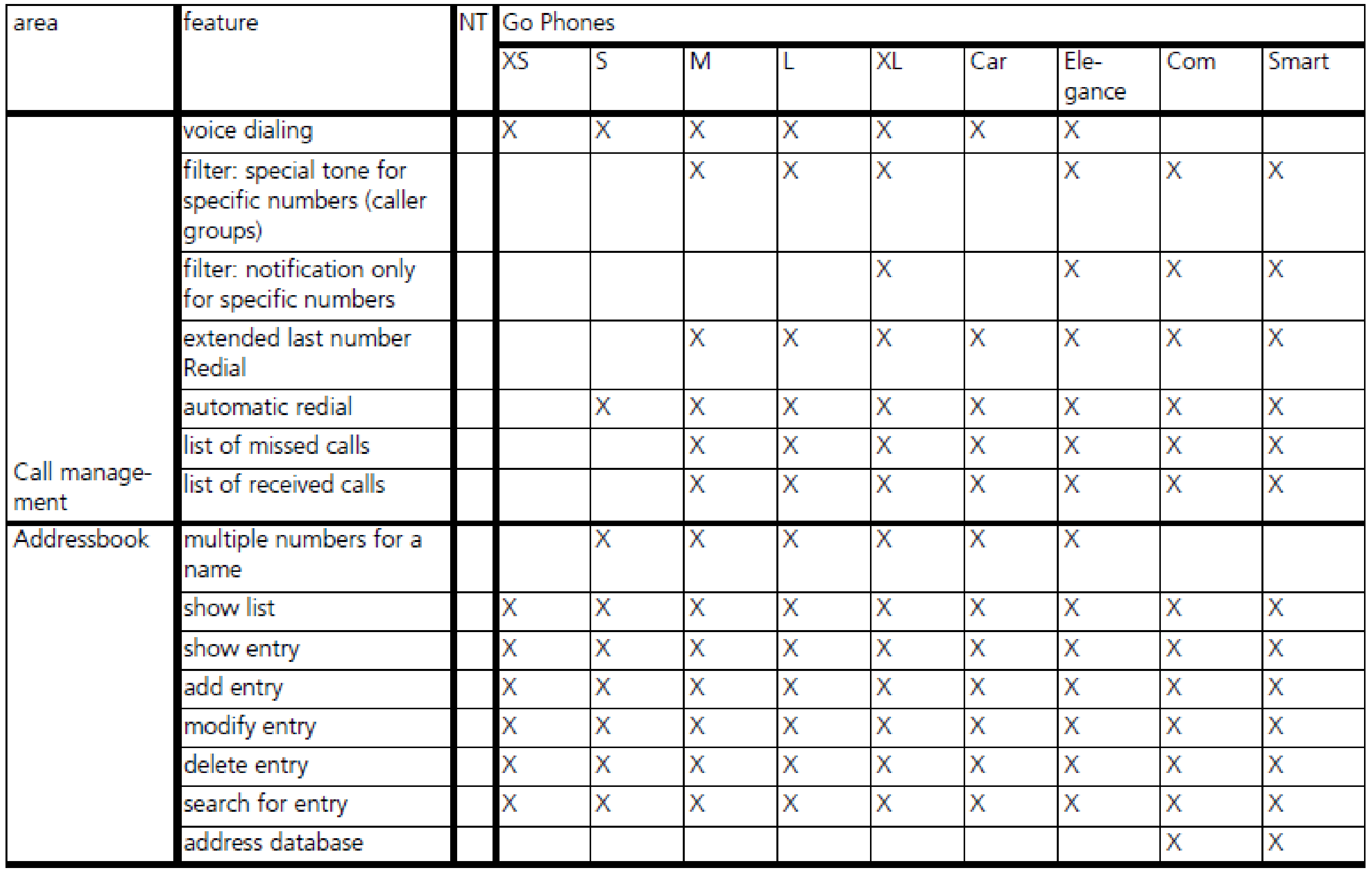
Expected *Release Plan* for different product line members



Case study

Scoping III

Based on portfolio analysis: *Product Feature Matrix*



area	feature	NT	Go Phones								
			XS	S	M	L	XL	Car	Ele-gance	Com	Smart
Call manage-ment	voice dialing		X	X	X	X	X	X	X		
	filter: special tone for specific numbers (caller groups)				X	X	X		X	X	X
	filter: notification only for specific numbers						X		X	X	X
	extended last number Redial				X	X	X	X	X	X	X
	automatic redial			X	X	X	X	X	X	X	X
	list of missed calls				X	X	X	X	X	X	X
	list of received calls				X	X	X	X	X	X	X
Addressbook	multiple numbers for a name			X	X	X	X	X	X		
	show list		X	X	X	X	X	X	X	X	X
	show entry		X	X	X	X	X	X	X	X	X
	add entry		X	X	X	X	X	X	X	X	X
	modify entry		X	X	X	X	X	X	X	X	X
	delete entry		X	X	X	X	X	X	X	X	X
	search for entry		X	X	X	X	X	X	X	X	X
	address database									X	X

Case study

Scoping IV

Describing relevant domains (based on product description)

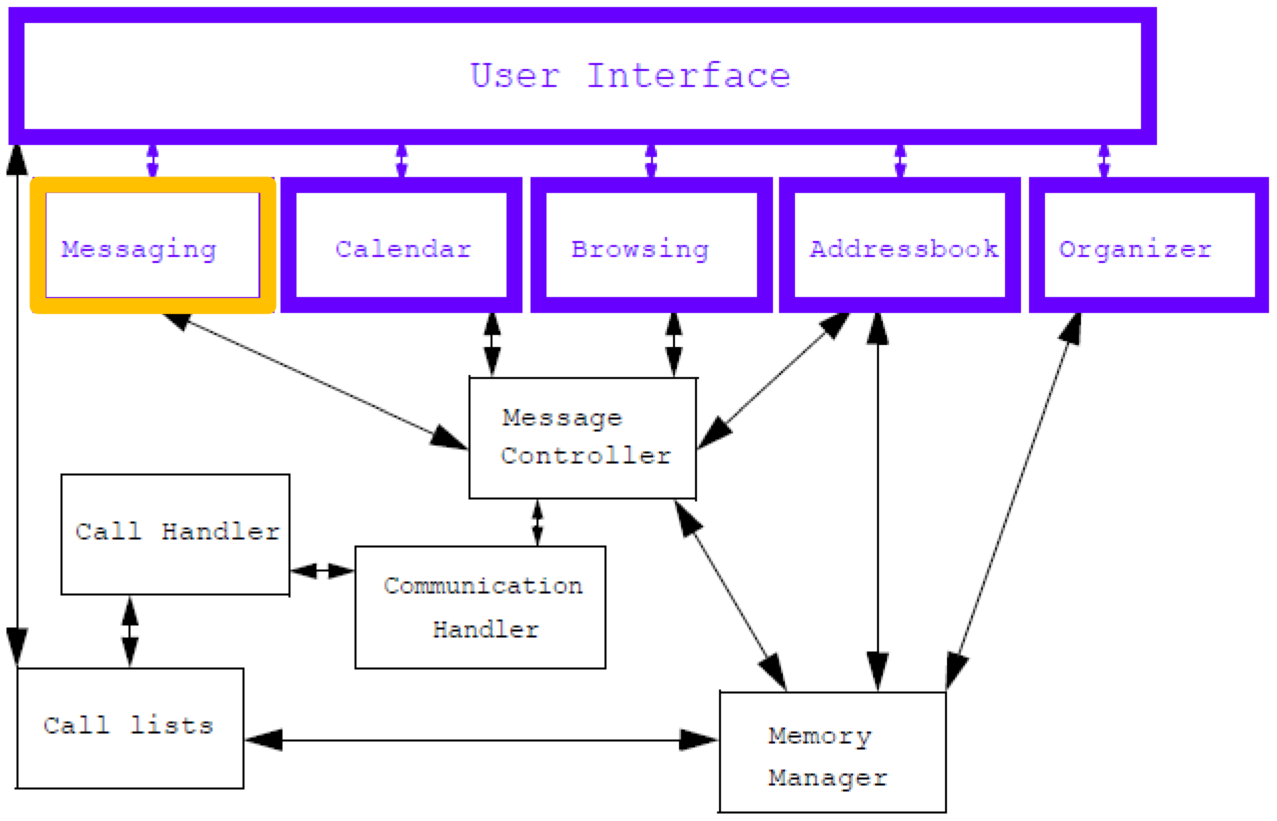
Example: *Messaging* domain

- Primary function
  - Handling messages (sms, email, …)
- Boundary rules
  - In: display text parts of messages
  - Out: display media parts of messages (pictures, files, …)
- Higher-level domains
  - User interface
- Lower-level domains
  - Message controller
- …

Case study

Scoping V

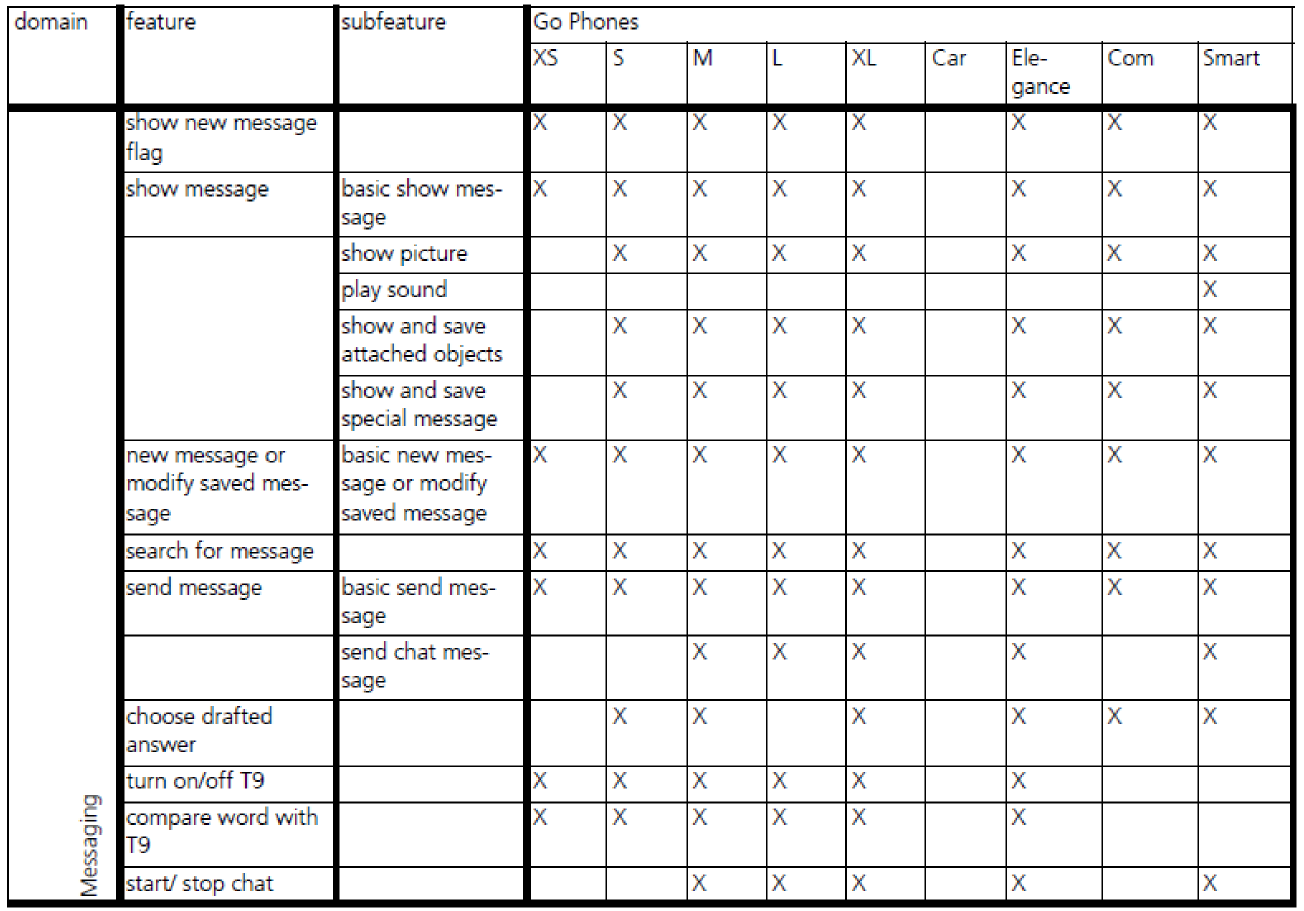
Based on identified domains: *Domain structure diagram*



Case study

Scoping VI

Based on domain analysis: *Initial product map*



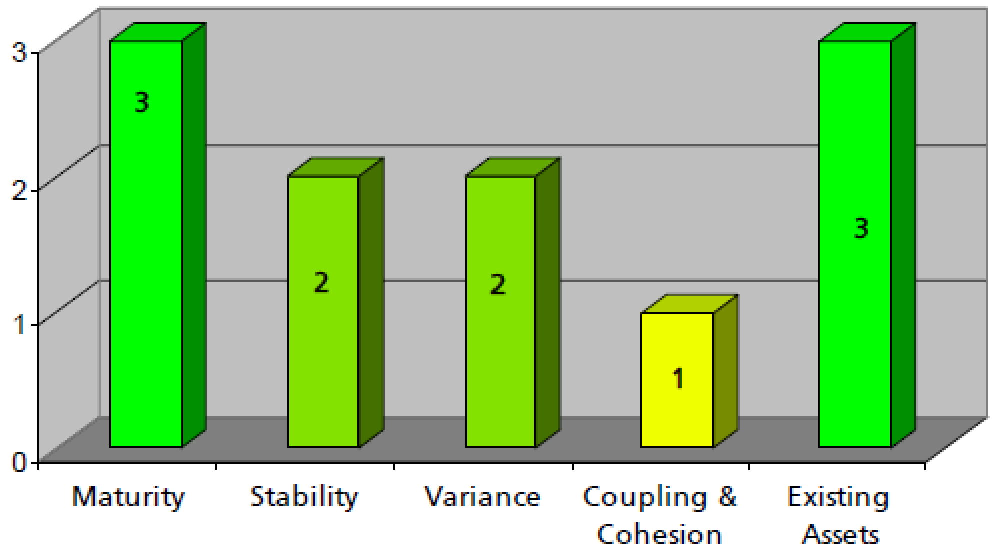
domain	feature	subfeature	Go Phones								
			XS	S	M	L	XL	Car	Ele-gance	Com	Smart
Messaging	show new message flag		X	X	X	X	X		X	X	X
	show message	basic show mes-sage	X	X	X	X	X		X	X	X
		show picture		X	X	X	X		X	X	X
		play sound									X
		show and save attached objects		X	X	X	X		X	X	X
		show and save special message		X	X	X	X		X	X	X
	new message or modify saved mes-sage	basic new mes-sage or modify saved message	X	X	X	X	X		X	X	X
	search for message		X	X	X	X	X		X	X	X
	send message	basic send mes-sage	X	X	X	X	X		X	X	X
		send chat mes-sage			X	X	X		X		X
	choose drafted answer			X	X		X		X	X	X
	turn on/off T9		X	X	X	X	X		X		
	compare word with T9		X	X	X	X	X		X		
	start/ stop chat				X	X	X		X		X

Case study

Scoping VII

Analyze benefits and risks of identified domains

- Assess reuse potential and cost-benefit ratio
- Parameters: maturity, number of relevant systems, …
- Example: *Messaging* domain

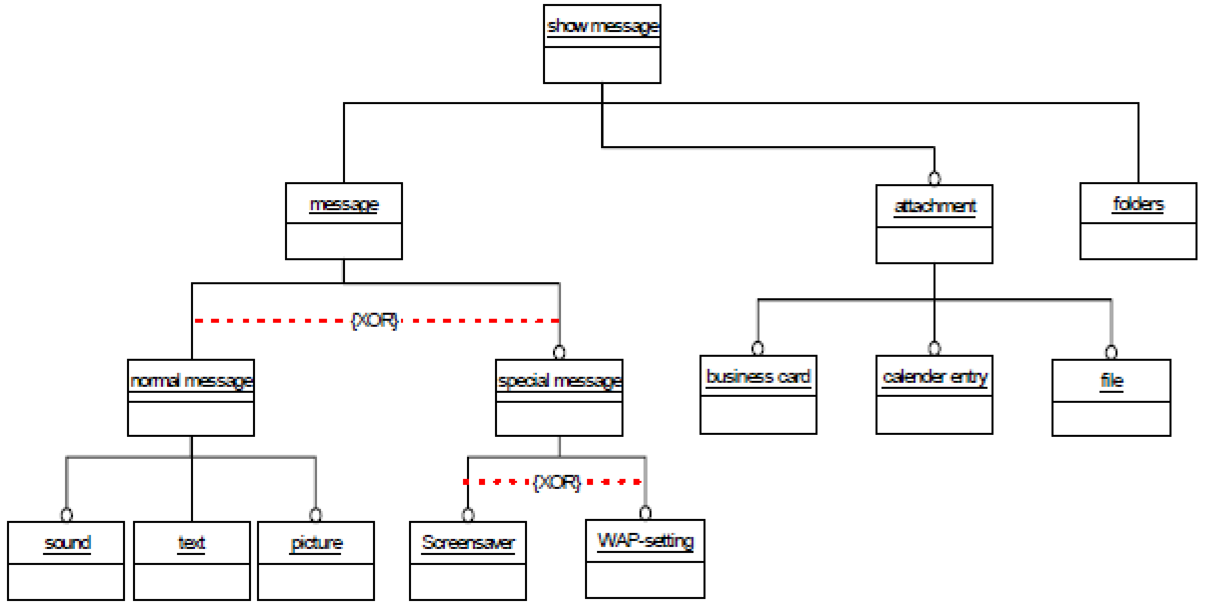


Case study

Scoping VIII

Define *variability model* for different features

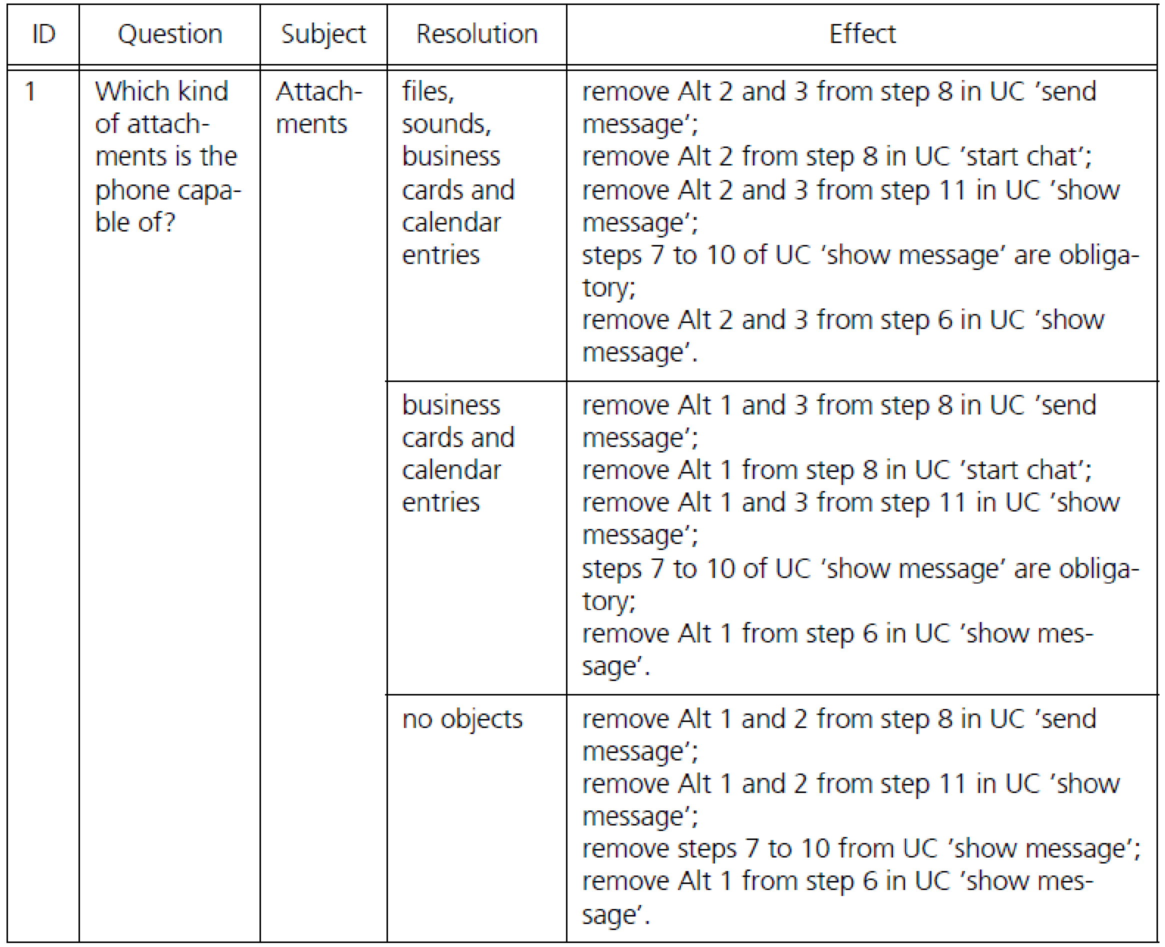
- Based on use cases from requirements phase (not discussed here)
- Example: *show message* feature



Case study

Scoping IX

Come up with (generic) *decision model* for instantiations



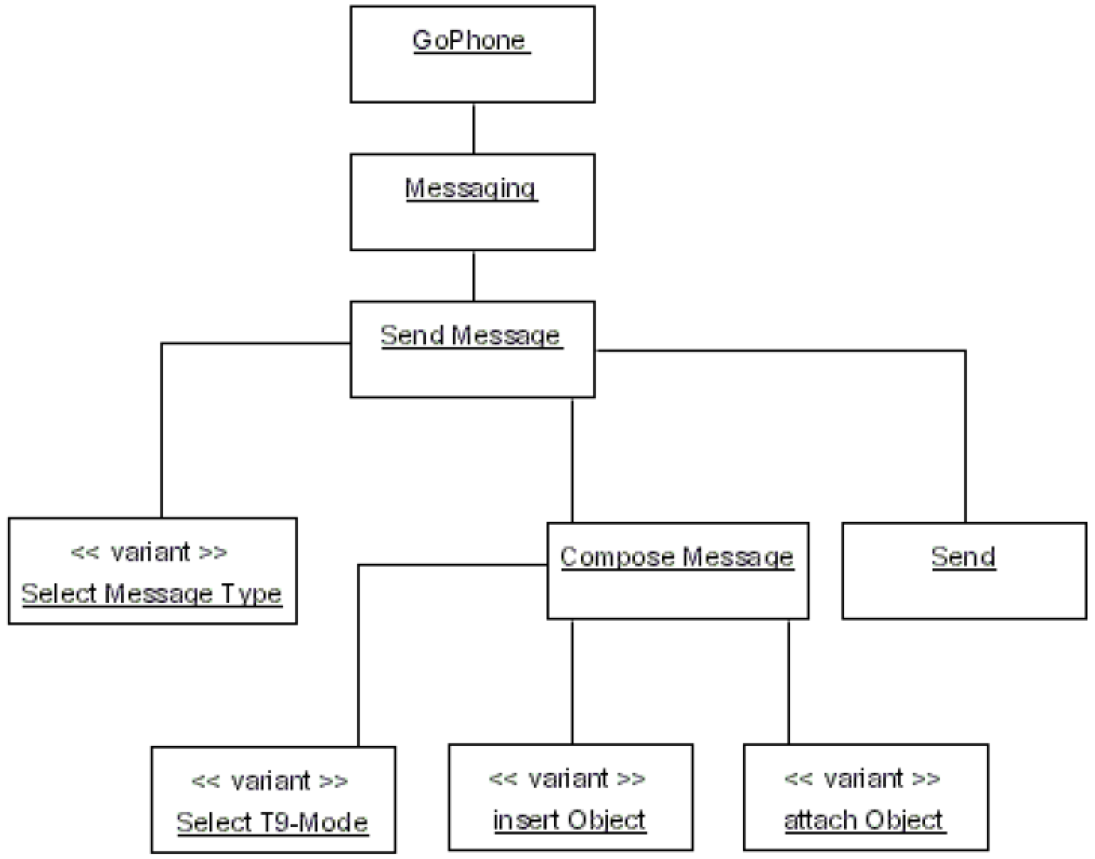
ID	Question	Subject	Resolution	Effect
1	Which kind of attach-ments is the phone capa-ble of?	Attach-ments	files, sounds, business cards and calendar entries	remove Alt 2 and 3 from step 8 in UC 'send message'; remove Alt 2 from step 8 in UC 'start chat'; remove Alt 2 and 3 from step 11 in UC 'show message'; steps 7 to 10 of UC 'show message' are obliga-tory; remove Alt 2 and 3 from step 6 in UC 'show message'.
			business cards and calendar entries	remove Alt 1 and 3 from step 8 in UC 'send message'; remove Alt 1 from step 8 in UC 'start chat'; remove Alt 1 and 3 from step 11 in UC 'show message'; steps 7 to 10 of UC 'show message' are obliga-tory; remove Alt 1 from step 6 in UC 'show mes-sage'.
			no objects	remove Alt 1 and 2 from step 8 in UC 'send message'; remove Alt 1 and 2 from step 11 in UC 'show message'; remove steps 7 to 10 from UC 'show message'; remove Alt 1 from step 6 in UC 'show mes-sage'.

Case study

Product line architecture I

Family engineering: map scoping results to generic PLA

- Especially emphasize variation points and variants

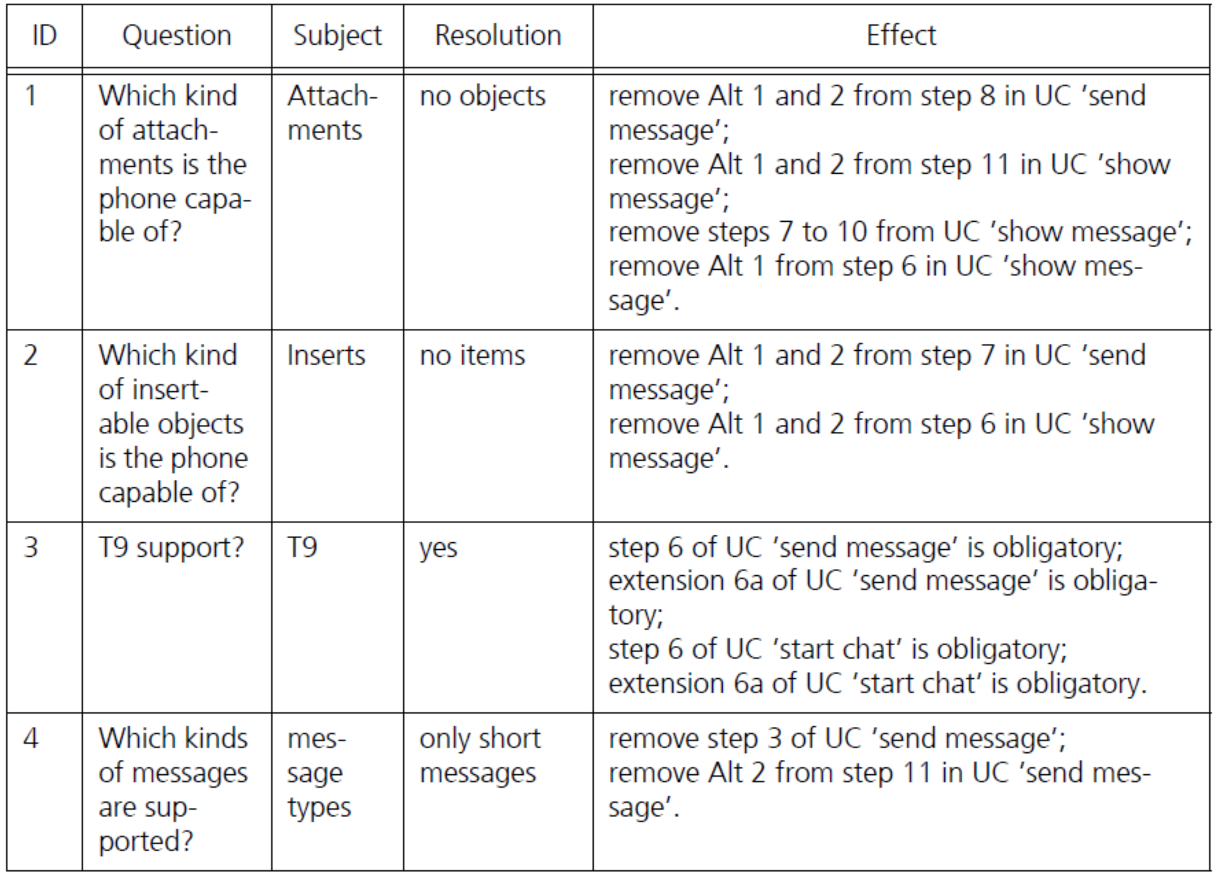


Case study

Product line architecture II

Family engineering: come up with a *resolution model*

- Instantiation of a generic *decision model* for one particular product



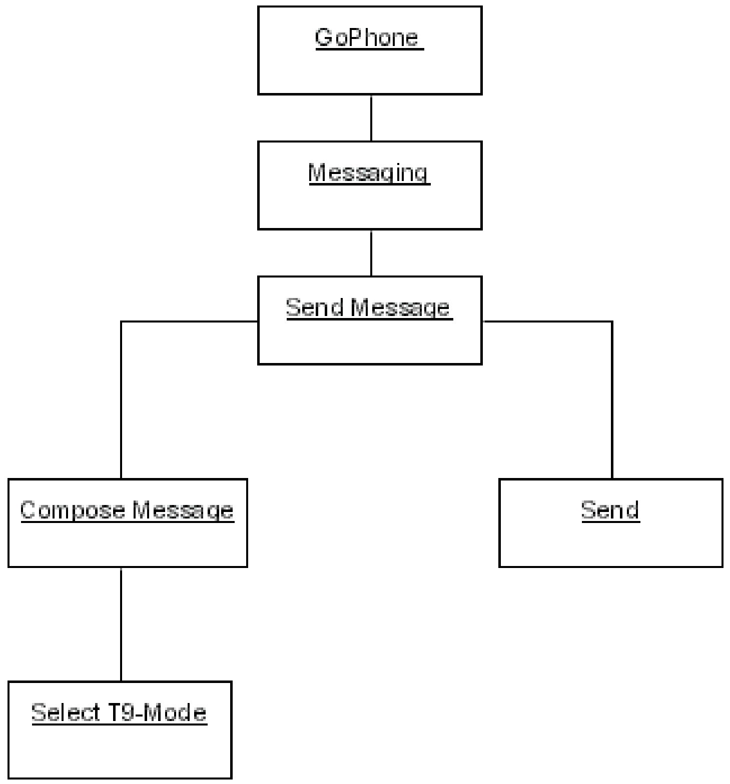
ID	Question	Subject	Resolution	Effect
1	Which kind of attach-ments is the phone capa-ble of?	Attach-ments	no objects	remove Alt 1 and 2 from step 8 in UC 'send message'; remove Alt 1 and 2 from step 11 in UC 'show message'; remove steps 7 to 10 from UC 'show message'; remove Alt 1 from step 6 in UC 'show mes-sage'.
2	Which kind of insert-able objects is the phone capable of?	Inserts	no items	remove Alt 1 and 2 from step 7 in UC 'send message'; remove Alt 1 and 2 from step 6 in UC 'show message'.
3	T9 support?	T9	yes	step 6 of UC 'send message' is obligatory; extension 6a of UC 'send message' is obliga-tory; step 6 of UC 'start chat' is obligatory; extension 6a of UC 'start chat' is obligatory.
4	Which kinds of messages are sup-ported?	mes-sage types	only short messages	remove step 3 of UC 'send message'; remove Alt 2 from step 11 in UC 'send mes-sage'.

Case study

Product line architecture III

Application engineering: instantiate generic PLA

- Resolve variation points according to the *resolution model*
- Can directly be used for code generation if detailed enough



Conclusion (1)

Key differences of software product line engineering in comparison with single software-system development:

- The need for **two distinct development processes**: domain engineering and application engineering
- The need to **explicitly define and manage variability**:
  - During domain engineering, variability is introduced in all domain engineering artifacts
  - It is exploited during application engineering to derive applications tailored to specific customer needs
- PLE typically exhibits **model-driven development** techniques to semi-automatically generate products from product line models

["Software Product Line Engineering." K. Pohl et al. (2005)]

Conclusion (2)

Pros:

- Well-done PLE significantly reduces development costs, increases quality, and decreases time-to-market.
- Well-done PLE leads to better understanding of domain and enables an economic way of structured reuse.

Cons:

- Necessary high up-front investments may be hard to justify towards internal and external stakeholders.
- Strong domain knowledge necessary for scoping
  → Incorporation of domain experts crucial
- Badly scoped domain may lead to assets that are either not demanded by any stakeholder or not anticipated
  → Risk of wasting time and money
- So far, limited success in practice

From requirements to system design

Software frameworks

"A framework is a set of classes that embodies an abstract design for solutions to a **family of related problems** and supports **reuse** at a larger granularity than classes."

["Designing Reuseable Classes." R.E. Johnson and B. Foote (1988)]

"A framework is a **generic structure** that is extended to create a more specific subsystem or application."

["Software Engineering." Ian Somerville. (2010)]

"… conventions, principles and practices for the description of architectures established within a **specific domain of application** and/or community of **stakeholders**"

[ISO 42010]

- There are classes of problems, which are similar in structure and differentiate from each other at certain points.
- Goal: take advantage of the commonalities and address the differences.
- In order to solve a concrete problem using a framework, it has to be adapted to the problem.
- A well-designed framework
  - enables reuse of large parts and
  - can be adapted flexibly to different problems.

Software frameworks

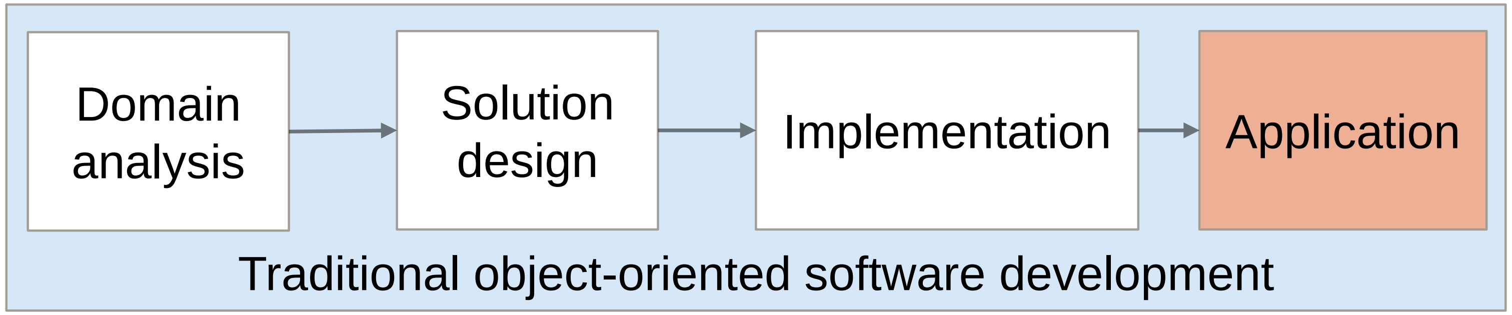
- Frameworks provide support for generic features that are likely to be used in all applications of a similar type.
  - For example, a UI framework will provide support for interface event handling and will include a set of widgets that can be used to construct displays. It is then left to the developer to specialize these by adding specific functionality for a particular application. For example, in a UI framework, the developer defines display layouts that are appropriate to the application being implemented.

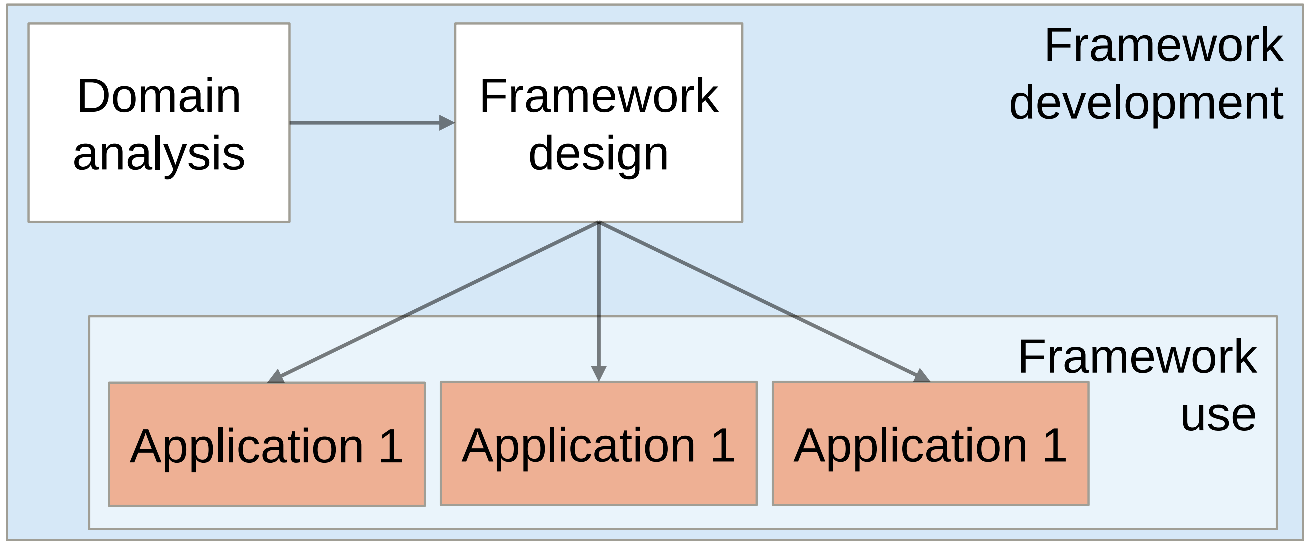
- Frameworks support design reuse in that they provide a skeleton architecture for the application as well as the reuse of specific classes in the system.

- Frameworks are language-specific.

["Software Engineering." Ian Sommerville. (2010)]

Development process of frameworks





["Guidelines for framework development process." S. Vojislav et al. (2011)]

Extending frameworks

- Frameworks are generic and are extended to create a more specific application or sub-system.

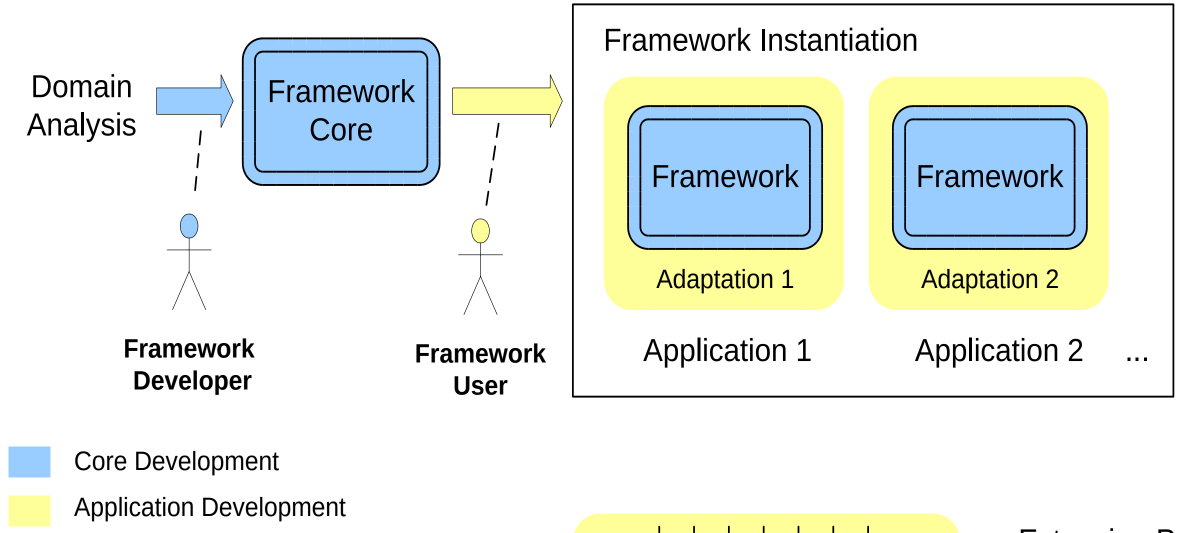
- Extending the framework involves
  - adding concrete classes that **inherit operations** from abstract classes in the framework and
  - adding **methods that are called in response to events** that are recognised by the framework.

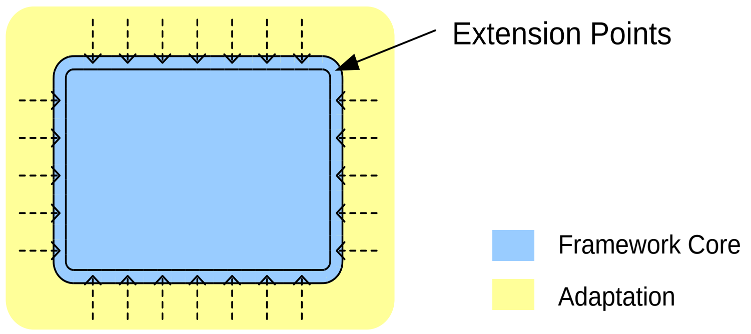
- A major problem with frameworks is their complexity and the time it takes to use them effectively.

- Combining (composing) different frameworks is often a challenge:
  - Concrete classes can only inherit from one abstract class in many languages.
  - Conflicts might arise in the temporal order of the handling of events from multiple frameworks.

Extension points

The places, where a framework can be adapted are called *extension points*, or *hot-spots*.

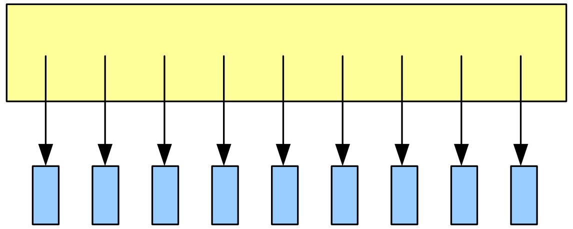




Software libraries and frameworks

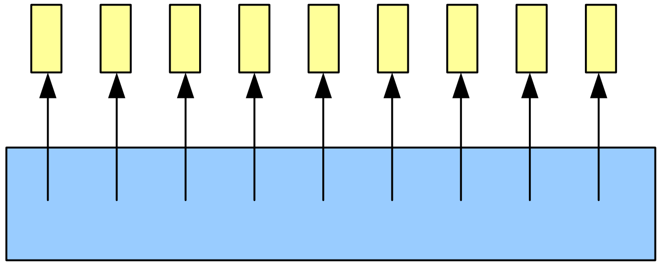
Class library

- Self-contained; pluggable ADTs; problem-specific functionality
- Call-down principle
  - The control flow is under the control of the application; the class library does not take over control



🟨 Code developed by the application developer

🟦 Reused code



Framework

- Reusable, semi-complete application
- Domain-specific functionality
- Call-back- or Hollywood-principle (don’t call us, we’ll call you)
  - Inversion of control: invert the control flow of the application

["Object-oriented application frameworks." M. Fayad and D. C. Schmidt. (1997)]

Examples of application frameworks

GUI application frameworks

- An *application* has *menus* (exit, cut, copy, paste), multiple *windows*, a shared *clipboard* with standardized data formats, a printer menu, a print preview (it can be started, shut down, suspended, …)
- MacApp by Apple for Macintosh Applications (Pascal, ObjectiveC, since 1985)
- Microsoft Foundation Classes (C++, since 1992)
- Eclipse Rich Client Platform (Java, since ~2004)

Web application frameworks

- A web application has sessions, templates, actions, forms (with fields and validators)
- Struts (Java, since ~2000)
- Rails (Ruby, since ~2004)
- Angular, React, etc. based on MVC pattern

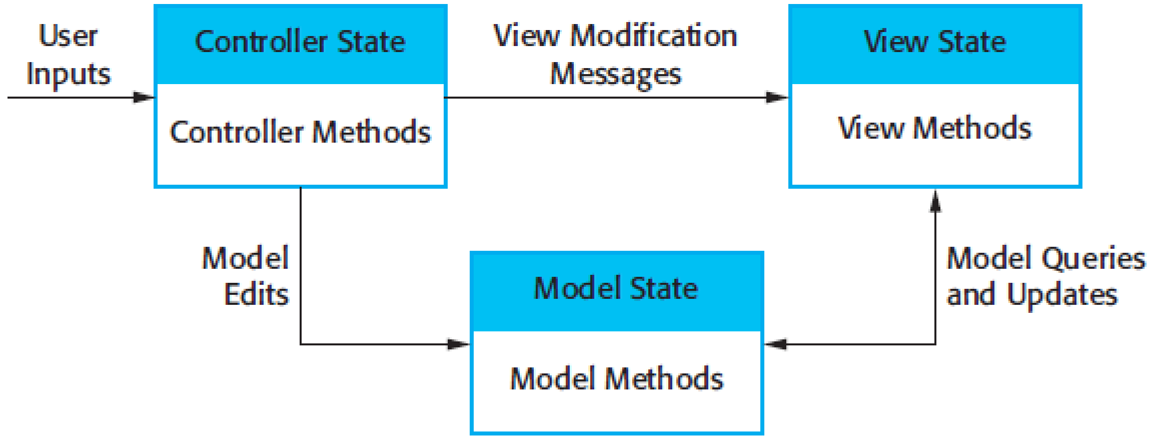
Test frameworks

- A test suite consists of tests, tests produce metrics, metrics are displayed in reports
- Jmeter (for Web Applications, Adaptations in Java)

Model-view controller (MVC) pattern

- MVC pattern is a composite pattern
  - It includes the **observer**, **strategy**, and **composite** patterns

You should already know these patterns!



["Software Engineering." Ian Sommerville. (2010)]

["Design Patterns." E. Gamma et al. (1995)]

Reference architectures

A *Reference architecture* is an abstract software architecture for a specific application area. It defines structures and types of software elements, and their interactions and responsibilities. The defined structures are applicable for all systems of a domain.

- Blueprints of architectures for software systems.
- A system conforms to a reference architecture,
  if the structures of the reference architecture can be found in it.
- A system may conform to more than one reference architecture, since it may exhibit different structures.

Types of reference architectures

- **Functional reference architectures** structure the required functionality through functional areas.
  - Functional decomposition according to technical aspects.
  - Functional areas later can be used as a basis for components.

- **Logical reference architectures** define the layers and components to be implemented.
  - Binding design rules simplify the design of the components.

- **Technical reference architectures** additionally define the programming language and the infrastructure to be used.
  - They are the basis for a detailed architecture, the implementation and the deployment.
  - They provide constraints and recommendations on deployment of components or complete subsystems.

Also called functional, logical, and technical blueprint.

Types of reference architectures

	Functional	Logical	Technical
Phase of the development process	Requirements analysis	Conceptual design	Detailed design, Implementation
Provides a basis for	Functional specification, planning of subsystems	Logical architecture, planning of the implementation	Detailed architecture, implementation, deployment
Elements	Functional areas as units of functionality	Components as units of design and implementation	Components as units of implementation and deployment
Stakeholder	User, manager, project leaders	Project leaders, architects, developers	Architects, developer, maintenance staff

Elements of a reference architecture description

	Functional	Logical	Technical
Architectural overview	Functional areas, data flow	Components, layers to be implemented	Technical components, layers which can be deployed
Textures (structures, principles and design concepts which occur often)	-	Design rule, may be expressed as a design pattern	Design rule, design pattern, code template
Reference interfaces	Named interfaces, if any	Named interfaces; defined in an implementation neutral definition language (e.g., IDL), if necessary	Defined in the programming language used

Examples of reference architectures

- Database-centric architectures
  - Mainframes
  - Data warehouse
  - Big-data architectures
- Message-oriented architectures
  - Message-oriented middleware
    - Message brokers
    - Java remote message invocation (RMI)
- Object-oriented architectures
  - Common object request broker architecture (CORBA)
- Component-based architectures
  - Java enterprise edition (JEE)
  - Microsoft .NET
  - Android
  - AUTOSAR
- Service-oriented architectures
  - Web services
    - Java API for XML Web services (JAX-WS)
  - REST architecture
  - Open services gateway initiative (OSGi)
  - Microservice architecture

From requirements to system design