Software Product Lines

maurizio.morisio@polito.it

Product lines

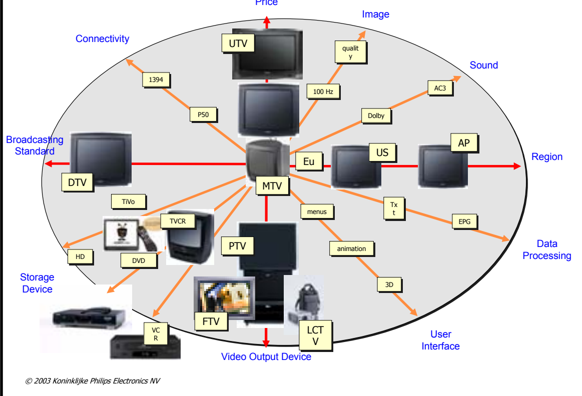
Product lines

Nokia

- 25-30 new products a year
- Variants/Features
  - Number of keys, input methods
  - Display size
  - Sets of features
  - Languages (58)
  - Protocols and APIs (CDMA, TDMA, AMPS, GSM, GPRS, ..)
- Constraints on variants
  - market segmented (low – high end)
  - Configurable (on - off)
  - Plug and playable
  - Changeable after product release

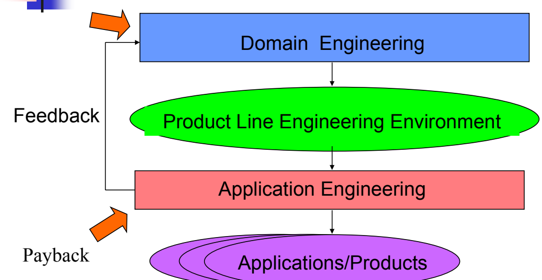
  - Source: Anders Heie, SPLC2 Keynote Presentation

Philips - TV Diversity

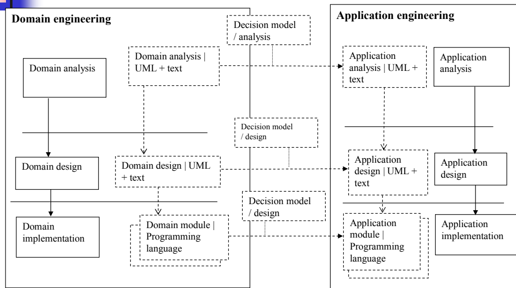


© 2003 Koninklijke Philips Electronics NV

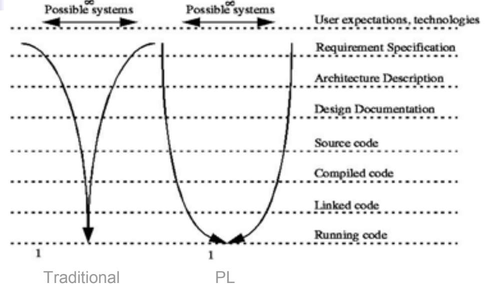
Process



Process



Variability



Binding variation points

When to commit a variation point to a particular variant?

Binding time:

- Product architecture derivation (configuration mgmt.)
- Detailed design (component specializations)
- Compilation (macros and aspects)
- Linking (linker directives)
- Runtime (user and environment customizations)

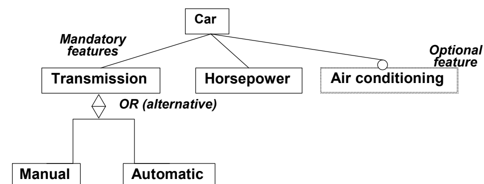
The key points

- Process and organizational change
- Techniques
  - Product line vs. products
  - Domain analysis
  - Scoping, commonality and variability analysis

Domain analysis and scoping

- Understand the domain
  - Terms, problems and solutions
  - Define the PL
    - What is in, what is out
    - Describe the PL: commonality and variability

Commonality Variability



- Part 1
  - Definition, motivation
- Part 2
  - Key practices
- Part 3
  - Practices in action, case studies

Part 1

Definition

- SPL is a set of software intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way
- Economy of scope

Definition

- Core assets
  - Architecture, components, domain models, requirements, ..
- Product line
  - Set of products (in same domain, from same core assets)
- Domain
  - Collection of related functionality
    - Ex telecommunication, that includes subdomains switching, protocols, telephony, switching
- Development
  - Build, buy or commission core assets and spl

What SPL are not

- (fortuitous) reuse of small pieces of code
  - SPL reuse of large modules, in planned way, modules designed for reuse in several systems
- Single system devel with reuse (start from system previously built) (end up with 2 different systems)
  - SPL, core assets are developed and evolved, core assets are the key IPR elements for company
  - SPL seen as single product, each system is a tailoring of the core assets

- CBD
  - SPL uses components but
    - All components are prescribed, way of customizing them is prescribed, overall archi is prescribed
- Reconfigurable architecture
  - Ex OO framework
  - SPL defines the overall architecture, but also more
- Releases and versions of a single product
  - SPL manages many products (each with its versions) in parallel (each is instance of core assets)
  - \Old\ instances are products as new ones (not obsolete)

- Set of technical standards
  - Ex which middleware, which web browser, which interfaces
  - SPL may use them, but defines much more

Benefits

- Productivity
- Time to market
- Maintain market presence
- Growth
- Quality
- Customer satisfaction
- Mass customization
- Inability to hire

Benefits from reuse of

- These deliverables, for each instance of family
  - Requirements
  - Architecture
  - Components
  - Performance models
  - Testing
  - Planning
  - Processes
  - people

Benefits for

- CEO
  - Productivity, time to market, market presence (capture niches), growth
- COO
  - Less people to do more, no need for hiring
- Tech manager
  - Predictability (same archi, same process, same plans)
- Developer
  - Less stress, less integration hassles, competence on SPL
- Architect, core asset developer
  - Design for reuse in multiple instances, challenge and skill
- Marketing
  - Better confidence on functions, performance, deliveries, customer satisfaction
- Customer, end user
  - Better product, more flexible

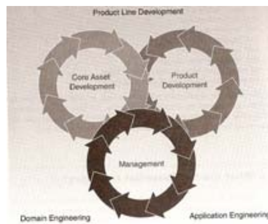
Additional Costs

- Requirements
  - Capture requirements for a family
- Architecture
  - Support variation
- Components
  - Support variation/extension with no performance loss
- Performance models
  - Constraints added (no move process to processor,..)
- Testing
  - Support variation
- Planning
  - Support variation
- Processes
  - Support unique product needs (variation)
- People
  - Additional skills and training

- No clear model investment- return

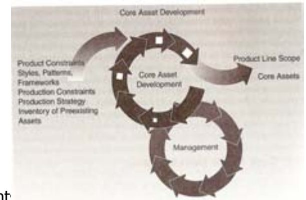
Activities

- Core asset development
  - Or domain engineering
- Product development
  - Or application engineering
- Management
  - All parallel, all iterative
    - Core assets are used to develop products and
    - Products feed back assets



Core asset development

- Activities/outputs
  - Scoping
  - Core assets
  - Production plan
- Inputs
  - Product constraints
  - Styles pattern frameworks
  - Production constraints
  - Production strategy
  - Inventory of preexisting assets



Core asset development

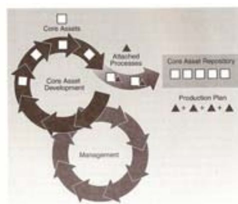
- Scoping
  - Decide which product are in the PL
    - Enumeration or
    - Description of common parts and variabilities
  - A key activity
    - Too large scope, too high variability, PL tends to single products one at a time
    - Too narrow scope, little leverage, PL stagnates

Core asset development

- Core assets
  - See list before
    - Architecture – and constraints, patterns
    - Components – and test cases, design doc
    - Requirements
  - Also process attached to each core asset
    - Detailing how core asset should be used to develop product
    - Ex, requirements. 1 use baseline requirements 2 specify variation for each variation point 3 add requirements if missing 4 validate that extensions / variations can be supported by archi
  - Also definition of how the asset base will evolve

Core asset development

- Production plan
  - Process to develop a product from the core assets
  - Basically made of process parts attached to each core asset
  - Includes handling of variation
    - Adding component from a selection
    - Changing or parameterizing a component (inheritance or else)
    - Generating
  - Use of tools
  - Must be targeted to the future users



Core asset development - inputs

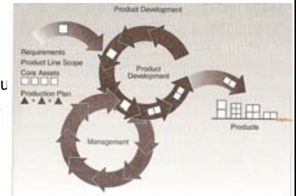
- Product constraints
  - ..ilities to satisfy: Security reliability etc
  - Performance
  - Market and technology outlook
  - Interfaces with external systems

Core asset development - inputs

- Architectural styles, patterns
  - Chosen as viable/proven solutions
- Production constraints
  - Standards, legacy systems components to be used, available/required COTS
  - Time to market and cost
- Production strategy
  - Core assets developed first, then products, or core assets extracted from products?
  - Cost center(s) covering the costs?
  - Products generated or assembled?
- Inventory of preexisting assets to be mined

Product development

- Output
  - Products
- Inputs
  - Requirements for produ
    - As delta or variations
  - Scope
    - Is product in scope?
  - Core assets
  - Production plan



Core assets vs. product development

- Mature PL organizations recognize that their key activity is core asset development, not product development
  - Concentrate on ONE not on MANY

Management

- Put all together
- Organizational management
  - Funding model, resources, motivation
  - Responsible for success
- Tech management
  - Oversees development
- PL manager: Leadership and not only control

Part 2

- PL Practice areas
  - Software engineering
  - Technical management
  - Organizational management

Practice areas

- Software engineering
  - Architecture definition
  - Architecture evaluation
  - Component development
  - COTS utilization
  - Mining existing assets
  - Requirements engineering
  - Software system integration
  - Testing
  - Understanding relevant domain

Practice areas

- Technical management
  - Configuration management
  - Data collection, metrics and tracking
  - Process definition
  - Scoping
  - Technical planning
  - Technical risk management
  - Tool support

Practice areas

- Organizational management
  - Building a business case
  - Customer interface management
  - Developing an acquisition strategy
  - Funding
  - Launching and institutionalizing
  - Market analysis
  - Operations
  - Organizational planning
  - Organizational risk management
  - Structuring the organization
  - Technology forecasting
  - Training

Practice areas

- Software engineering
  - Understanding relevant domain
  - Requirements engineering
  - Architecture definition
  - Architecture evaluation
  - Component development
  - COTS utilization
  - Mining existing assets
  - Software system integration
  - Testing

Understanding domain

- Domain knowledge: set of concepts and terminology understood by practitioners in an area of expertise
  - Ex. distributed banking applications require knowledge on banking practices, distributed systems, workflow management, DBs, networking, GUIs.
- DK is related to success in developing SPL
- How to achieve DK?

Understanding domain

1. Identify areas of expertise
2. Identify recurring problems and known solutions
3. Capture and represent this information so that it can be communicated to and reused by stakeholders

Usually obtained by

- previously built applications
- experts
- domain analysis

Domain analysis

- Elicit information from
  - Experts
  - Products
  - Books, papers, handbooks
- Represent in domain model
- Domain model. Documentation of DK. Essential to reduce dependency on experts, improve quality of information, and support communication.

Understanding domain

- Issues
  - Unexpressed glossary is most of the time source of misunderstandings
  - Involving the key experts
- PL issues
  - Domain analysis for PLs concentrates also on commonality and variability

Understanding domain

- Practices
  - OOA, UML
    - Same UML concepts, applied to domain modeling
  - FAST [Weiss and Lai] Family oriented Abstraction Specification and Translation
    - Emphasis on DSL and generation
  - FODA [Kang90] Feature Oriented Domain Analysis
    - Features as key abstraction
  - Synthesis, SPC

Understanding domain

- Risks
  - Analysis paralysis
  - Only 'mental models' (no shared documentation)
  - Lack of access to experts
  - Inadequate tool support
  - Design instead of analysis
  - Missing/fading management commitment

Requirements engineering

- General issues
  - Requirements, functional + non functional
  - Elicitation, analysis, specification, verification, management
  - Different stakeholders: end user, executive, developer, marketer; need to know requirements of all, and negotiate when conflicts
- PL specifics
  - Requirements for PL, requirements for product
    - Must be kept separate, but product requirements should be derived from and traceable to PL requirements
    - PL requirements contain common part + variation points
    - Product requirements fix each variation point
    - Product requirements should be easily generated from PL requirements

Requirements engineering

- PL specifics
  - R elicitation. Needs to anticipate all variations of products in the scope during PL expected lifespan
    - Use case variation points, domain analysis techniques
  - R analysis. Find commonalities and variabilities; what if economic analysis (what cost and ROI change if more commonality and less variability); defines precisely the scope
    - FODA (Feature Oriented Domain Analysis) technique
  - R verification. On PL requirements, on product requirements
  - R management. Change requests for requirements have to be assessed at PL level, and on impact on architecture + core assets

Requirements engineering

- PL specifics
  - Important to have early pass of PL requirements from key stakeholders, to initiate early design work, define initial architecture, check feasibility both on architectural and economic level (business case)

Requirements engineering

- Product specifics
  - Determine if required new product is in scope of PL
  - Guide all process (production testin deployment)
  - Steer evolution of PL

Requirements engineering

- Practices
  - Domain analysis
    - Usually OO analysis applied to a domain
    - Often includes feature analysis
  - Feature analysis
    - Feature = user visible characteristic/function
    - Tree of features, nodes can be AND / OR
      - FODA, FORM
  - Use cases
    - Enriched with variation points, in terms of include, extends, uses, parameterization
  - Change cases
    - Uses cases that represent potential future requirements
    - Are linked with (present) use cases that they impact
  - Stakeholder views
    - Set of requirements for a stakeholder. To identify and evaluate conflicts
  - R Traceability

Requirements engineering

- Risks
  - Failure to distinguish PL and P requirements
    - PL requirements are for core asset builders, P requirements are for P developers
  - Insufficient generality in PL R
    - Design does not support evolution over time
  - Excessive generality
    - Excessive effort to produce core assets and products
  - Wrong variation points
  - Not considering non functional R

Foda

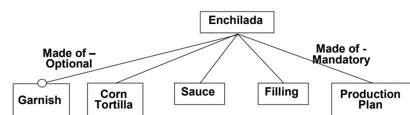
- Feature
  - Characteristic valid for PL
    - Functional, structural
    - Generic, concrete
- Can be mandatory, optional
- Can have value

The Enchilada PL

- Enchiladas verdes: Corn tortillas baked with a zesty filling, covered with a green tomatillo sauce. Your choice of chicken, beef, pork, and cheese.
- Enchiladas rojas: Corn tortillas baked with a zesty filling, covered with a red ancho chile sauce. Your choice of chicken, beef, or pork.
- 7 variations …

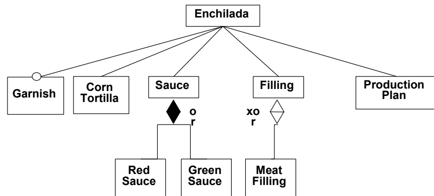
Foda – feature tree

- commonality

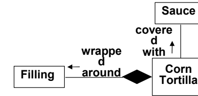


Foda – feature tree

- variability

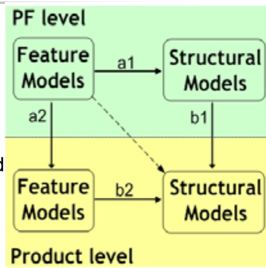


- Structure to be expressed otherwise, keeping links
  - ex UML



Documenting choices

Derivation is systematic
iff decisions are documented

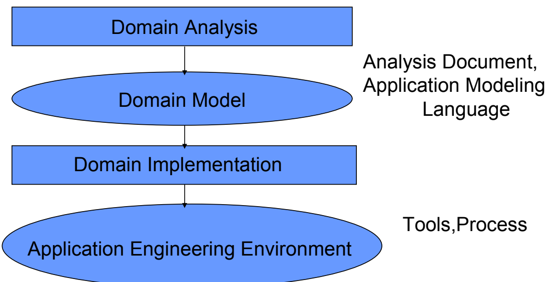


Other notations

FODA		FORM		GP		Featu-RSEB		J. Bosch		MEANING
	Manda-tory		Manda-tory f		Manda-tory		Compo-sed of		Compo-sition	**Mandatory** (if reachable, then f must be chosen)
	Optio-nal		Optio-nal f		Optio-nal		Optio-nal f		Optio-nal f	**Optional** (if reachable, may be chosen, or not)
	Alterna-tive		Altern-ative f		Alterna-tive		Vp-feature (XOR)		xor-specialisation	**One-of-many** choice from a group
-	-	-	-		Or-features		Vp-f use time bound (OR)		or-specialisation	***n*-of-many** choice from a group
-	-	-	-	-	-	-	-		External feature	**External** Feature
-	-	-	-		Open-feature	-	-	-	-	**Open** Feature
-	-	-	-		Prema-ture	-	-	-	-	**Premature** feature

FAST – Weiss and Lai

Domain Engineering



The Domain Model

- Conceptual Framework
  - Family Definition
    - Commonalities and Variabilities Among Family Members
    - Common Terminology for the Family
    - Abstractions for the Family
  - Economic Analysis
  - Application Modeling Language (AML): Language for stating requirements

The Domain Model (2)

- Mechanism for translating from AML to software
  - Alternative 1: Compiler
  - Alternative 2: Composer

Building The Conceptual Framework

- Qualify The Domain
  - Is it economically viable?
- Define The Decision Model
  - What decisions must be made to identify a family member?
- Define The Family
  - What do members of the family have in common and how do they vary?
- Design The Application Modeling Language
  - What is a good way to model a family member?
- Design The Application Engineering Environment
  - What are good mechanisms for using the decision model and the Application Modeling Language?

Defining The Family: The Commonality Analysis

- **Dictionary of terms**
  - Technical terms that define a vocabulary for the domain
    - Primary Condition: The availability of a unit: working: ready, unready, or unusable
- **Commonalities: Assumptions that hold for every member of the family**
  - Every unit must be in one of the four primary conditions.
- **Variabilities: Assumptions that define the range of variation for the family**
  - Some unit names have inhibit states.
- **Parameters of Variation: Quantification of the variabilities**
  - Whether or not a unit name can have an inhibit state: Boolean

Commonality Analysis of Configuration Control

- Approximately 20 staff-weeks
  - Spread over 6 months
  - 6-12 experts
- Produced a Commonality Analysis
  - Definitions
  - Commonalities
  - Variabilities
  - Parameters of Variation
- Reviewed by entire organization

Definitions

- Configuration: A set of units, the relationships between those units, and the status of the units.
- Primary condition: The availability of a unit: working, ready, unready, or unusable
- Realization: A sequence of steps from an initial configuration to a final configuration

Commonalities

C8. Every unit has a status.

C9. Every unit has a name and number, e.g., QLPS-0-1. Units with the same name provide the same functionality.

C10. All units of the same name have the same set of conditions.

Variabilities

V5. Some unit names have inhibit states.

V7. The units to which a unit is related vary from unit to unit.

V9. A child unit has one or more parent units. The parent units may be of different names.

V10. The number of units of each parent unit name that must be working or ready...

Parameters of Variation

P5: inhibit state -- whether a unit name can have an inhibit state -- Boolean

P6: restore parallelism -- whether units of this name may be restored in parallel

P22: parent name information -- by unit name

P25: parent units

Architecture definition

- Architecture
  - Components (in term of interfaces)
  - Connections
    - Implemented Corba, DCOM, Java RMI
  - Qualities
    - Performance, modifiability availability security, ..
  - Interaction with other systems
  - Style(s) chosen
    - P2P, layered, …

- Interface between two components
  - Set of assumptions that the programmers of the two components can safely make about the other component [Parnas 71]
    - Includes behavior, resources consumed, error handling
    - Much more than syntactic interface

Architecture definition

- Support for variation
  - Architecture for PL supports several products, in parallel, each product may be different for qualities, network, physical configuration, middleware, scale factors etc
- Variation mechanisms
  - Inheritance, delegation, build time parameters for components, pick component out of many, leave out a component, ..
  - Better if mechanism can be supported by tool, and language
    - Integration will be made several times, one per product

Architecture definition

- Documentation
  - Of architecture
    - Various UML extensions
    - Various views (structural, process, deployment
  - Of process to build/derive product

Architecture definition

- Variation mechanisms
  - Inheritance
  - Extension point
    - Part of a component can be augmented with functions
  - Parameterization
    - Placeholder for component, then defined at build time. Ex macro, templates
  - Configuration and module interconnection languages
    - Cfr make, ant
  - Generation
    - Specific high level language to parameterize and compose components
      - Cfr Eclipse EMF to define DSL
  - Compile time selections of different implementations (#ifdefs ..)

Architecture definition

- Risks
  - Components do not fit together
  - Products don't meet functional or quality goals
  - Products in theory in scope, but cant be produced
  - Tedious, ad hoc building process

Architecture definition

- Causes of Risks
  - Lack of a skilled architect
  - Inadequate information to produce architecture
  - Lack of supportive management
    - No culture of architecture based development
  - Poor tools
  - Poor timing
    - Architecture frozen too early or too late

Architecture definition

- Symptoms of flawed architecture
  - Too many parameterization points
  - Inadequate specifications
    - Only functional description of interfaces
    - No temporal constraints, ordering
  - Inadequate decomposition
    - Components do not support all functionality needed
  - Wrong level of abstraction for components
    - Too general, too specific
  - Excessive intercomponents dependencies

Architecture evaluation

- A kind of inspection to validate an architecture
- Techniques
  - ATAM, SAAM, SPE, ARID, ADR

Component development

- Starting from descriptions (interfaces + requested qualities) elaborated in architecture definition
- Can be developed from scratch, adapted from legacy, procured

Component development

- To develop a product a component can be
  - Used as is
  - Used after binding built in variabilities
  - Used after modifications
    - Adapter design pattern
    - Modify source – to be avoided (maintainance of original + clone)
  - Developed new
    - Standards and reviews to make it suitable to enter the asset base

Component development

- To develop a product a component can be
  - Used as is
  - Used after binding built in variabilities
  - Used after modifications
    - Adapter design pattern
    - Modify source – to be avoided (maintainance of original + clone)
  - Developed new
    - Standards and reviews to make it suitable to enter the asset base

Component development

- Variability mechanisms [Jacobson 97]

Mechanism	Time of specialization	Type of variability
Inheritance	Class definition	Add/modify attributes and methods
Extension	Requirements	Use case extension
Uses	Requirements	Use case include
Configuration	Before run time	Component specialized via parameter file (JavaBeans property file)
Parameters	Component implementation	Function/method paramaters
Template	Component implementation	Java 4 templates, C++ templates
Generation	Before or during runtime	Tool produces definitions from user input. Ex configuration wizard

Component development

- Variability mechanisms [Anastasopoulos 00]
  - Aggregation/delegation
    - Delegating object has a repertoire of candidates (objects + methods) that can perform specific function in different ways
  - Inheritance
  - Parameterization
  - Overloading
  - Attribute value pairs (Delphi)
    - Value of attribute is changed
  - Dynamic class loading (Java)
  - Static library
    - Library contains external functions, linked after compilation
  - Dynamic library
    - Linked at run time
  - Conditional compilation
    - Module contains several implementation, one is chosen at compile time
  - Frame technology
    - Hierarchy of frames, each containing options in terms of preprocessing rules. Top frame produces a module ready for compilation
  - Reflection
  - Aspect oriented programming

Component development

- Risks
  - Not enough variability
  - Too much variability
  - Choosing wrong variability mechanism
  - Poor quality of components

COTS utilization

- Some core components can be COTS. To find if/which
  - Analyze architecture in detail
    - Can be changed if using a COTS gives large payoff
  - Verify organizational constraints for COTS
    - Internal standards, norms, rules
  - Study marketplace
    - Possibly perform a continuous survey of marketplace
  - Be flexible on requirements for COTS
  - Develop an evaluation approach
    - Attributes to evaluate (evaluation criteria)
    - Notably support for variability
  - Select
  - Buy
  - Test (the single COTS), integrate, test
  - Maintain
    - Integrate new version of COTS, or competitor product

COTS utilization

- Practices
  - Several COTS evaluation and selection practices
    - SEI
    - Kontio
    - Morisio Tsoukias
    - ..

COTS utilization

- Risks
  - Unknown interactions
  - Poor COTS product quality
  - COTS does not satisfy non functional requirements
    - Security, performance
  - Lack of adaptability
  - Replacement for COTS components
  - Lack of vendor support
  - Diverging evolution of COTS

Mining existing assets

- Looking for already developed assets
    - especially high level: design, architectures, requirements, tests
  - Understand what is available
  - Understand time and cost for reengineering and use
    - Consider cost for whole PL lifetime
  - Decide

Mining existing assets

- Practices
  - Option analysis for reengineering
    - OAR, SEI
  - Clone detection in code
    - Ex CloneDR tool
    - (large) clones may point out commonalities
  - Architecture reconstruction tools
    - Rigi, Software Bookshelf, Discover, Code Base Management System, Dali
  - Component wrapping

Mining existing assets

- Risks
  - Unsuccessful search
  - Overly successful search
  - Fuzzy search criteria
  - Failure to search non software assets
  - Assets found turn out to be inappropriate
  - Wrong reengineering estimates

System integration

- Putting components together
  - Waterfall, integration at the end
  - Incremental, integration done as component is ready
  - Requires wider definition of interface
    - Not simple signature of function, see Parnas, problems of synchronization, resource consumption, performance, etc

System integration

- Integration and PL
  - Better to reuse also integration, integration cost is split among all members of PL
  - Ex. PL core assets
    - System function groups – around 30. Each composed of circa 20 system functions. Developed by large team
    - System function. Each composed of circa 20 Ada units
    - Ada unit – 1K Ada code
  - Product made integrating System function groups. Integration of system function group made once per the PL

System integration

- Integration and Products
  - Integration effort may range from
    - 0 – only setting parameters or running generator, and building
    - 100 – developing component(s), wrappers, integrating, testing

System integration

- Practices
  - IDD textual description of interfaces of components
  - IDL, syntactic description of signature of components
  - Continuous integration
  - Pre-integration
  - Wrapping
    - Transform interface of component
  - Generators
    - Ex sysgen to generate OS for specific platform. All variability is known in advance
    - Ex FAST (Lai Weiss)
  - Middleware

System integration

- Risks
  - Informal description of interfaces
  - Too small components, integration effort explodes

Testing

- In broad sense of V&V
- Issues
  - Split of integration/testing between PL and P process
    - Shift more testing on PL if higher safety domain
- Practices
  - Test assets organized around production assets
  - Self test, especially at PL level (core assets)
  - Regression testing, especially on points of variability
  - Develop testing environment for acceptance testing

Testing

- Reusable test assets
  - Around core assets, to ensure/facilitate same testing approach to all commonality parts
    - Coverage criteria, sampling strategies, test cases
    - Structured in same hierarchy as core assets

Testing

- Practices
  - Architecture evaluation, see before
- Risks
  - Inadequate unit testing
  - Inadequate specifications
  - Inadequate/missing tools

Practice areas

- Technical management
  - Configuration management
  - Data collection, metrics and tracking
  - Process definition
  - Scoping
  - Technical planning
  - Technical risk management
  - Tool support

Configuration management

- Terms: item, configuration
- Traditional development
  - Maintain a configuration for each product
  - Products are independent
- PL
  - Each core asset as a configuration
  - All products (instances of PL) managed together

CM - example

- E-commerce retail software. Three customers, selling wedding rings, bicycles, pianos; in US, EU, Canada. One with shopping cart, others without. One delivers to dealers, others do not.

CM

- Requirements variation
  - Items sold: bikes, rings, pianos
  - Location: US, EU, Canada
  - Shopping cart: Y, N
  - Delivery: dealer, factory
- Components, 20
  - 15 common
  - 5 with variants: Delivery, Main, Models, Selection, Title

CM - The ideal tool

- Define dependencies requirement variation – component version
- Select requirement variation
  - Ex: bike, US, shopping cart
- Obtain components (in right version) for build

Configuration management

- Risks
  - CM process and
  - CM tool
  - Not sophisticated/robust enough to support PL
    - CM tool usually targeted at managing one system over time

Process definition

- Key changes
  - Core asset development and instance development. Flows among them. roles
- Key decisions
  - Change control on core assets
    - instances cannot / can change core assets
    - Upgrade of instances when core assets change
  - Separate group to develop and maintain core assets

Scoping

- Decide what is 'in', what is 'out'
- Done (informally) in traditional development
  - Boundary, decided before RE
- Essential, and harder, in PL
  - Scope definition document, core asset of PL
  - Fuzzy at the beginning, crispier later
  - Must allow decision maker decide if an instance belongs or not to the PL
  - If scope too large, core assets cant be built
  - If scope too narrow, too little instances will be built, no economic advantage

Scope and requirements

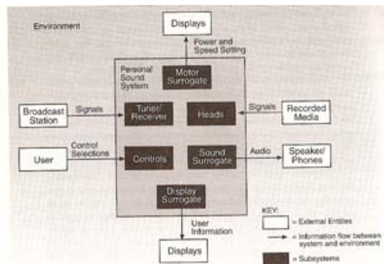
- Similar activities but
  - Scoping occurs earlier, to build a business case and decide if doing the PL
  - Scope document is more generic than req document, and used by managers, marketing people
  - Scoping also implies initial CV analysis

Scoping

- Practices
  - Examine existing products
    - Gather documentation, understand functions and characteristics
    - Decide if in or out
  - Run workshop
    - Business vision and goals
    - Exisiting products
    - Standards and technology

Scoping

- Techniques
  - Context diagramming



Scoping

- Techniques
  - Attribute product matrix
  - Variability is in attributes and attribute values
    - Some values may be common or out
    - Some attributes may be out
      - Rank attributes by priority and decide what are in

	Low cost model	Mid priced	High end
Radio tuner	analog	Digital presets	Digital presets
Displays	none	frequency	Frequency, graphical equalizer
Audio controls	volume	Volume, bass	Full spectrum equalizer

Scoping

- Techniques
  - PULse Eco
    - Attribute product matrix
    - Value function: defines value and cost associated to having a certain attribute or not
  - Product line scenarios
    - Usage scenarios, for common part to all products
    - For parts specific to some products

Practice areas

- Organizational management
  - Building a business case
  - Customer interface management
  - Developing an acquisition strategy
  - Funding
  - Launching and institutionalizing
  - Market analysis
  - Operations
  - Organizational planning
  - Organizational risk management
  - Structuring the organization
  - Technology forecasting
  - Training

Building a business case

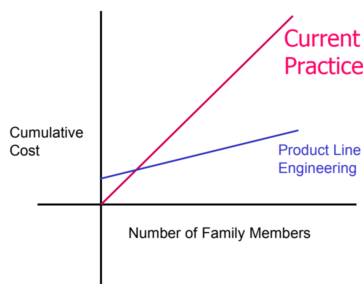
- Tool for business decision (one of the core assets)
  - Initially, building the SPL? yes – no
  - Then, tool to monitor and assess results
- What changes are needed?
- What are the benefits related?
- What are costs and risks?
- What is the measure of success?

- Changes
  - Organizational
  - Technical
  - Marketing and sales

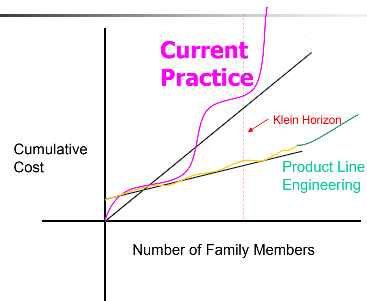
- Possible benefits
  - Reduced time to market
  - Reduced cost
  - Higher productivity
  - Improved quality
  - Increase customer base/market share
  - Ease of upgrades

- Measures
  - ROI, IRR
    - Usually very hard to apply, especially upfront
  - Cost versus number of instances build is usually easier to compute (a posteriori)
  - Try to estimate/collect
    - Initial cost PL (build core assets first time)
    - Incremental cost PL (maintain core assets)
    - Product development costs

Economics Of Families (Simplified)



Economics Of Families (More Realistic)



Break even

- Usually 2 or 3 family members built
  - AIM, Lucent.
  - Jacobson97, 1.5 to 3
  - Poulin IBM, 3
  - SPC, 1.67 to 4.86

- Measures
  - In reported studies cost was not the main driver
  - Time to market and the capability of handling more complex (variability) systems were the driver

Part 3

Case studies

- Cummins
- CCT
- MarketMaker

Cummins

Cummins

- Engines, diesel and natural gas
  - Engine control, hw + sw
    - Goals (contrasting):
      - high power, low emissions, low consumption
    - Constraints:
      - High reliability (mission critical)
    - Parameters (variations)
      - temperature, load, road incline
      - Power, 50 to 3500 HP
      - Cylinders, 4 to 18
      - Fuel systems, natural gas or diesel
      - Worldwide, different regulations and interfaces, 2 users and maintainers

Status

- 1994
  - Each control ~130Kloc C, assembler, microcode
  - Each control designed and implemented ad hoc
  - 6 projects underway
  - 12 projects planned
    - Need to hire 40 developers, not possible
  - Decision to switch to PL approach

Actions

- Stopped all projects
- Defined
  - Development environment group
    - Define process, tools, CM
  - Core group
    - Defines core assets
    - Starting from more advanced and stable of the 6 running projects
  - Project groups
    - Customizes into specific project

The business case

- Mainly, impossible to continue as is
- On the long term
  - More capability to deliver more functionality
  - With same staff (productivity gain)
  - With shorter lead time

Building core assets

- Make
  - No, for time pressure
- Buy
  - No, for company policy (no rely on external vendor for products/services that differentiate the company, represent competitive advantage)
- Commission
  - No, same as above
- Mine
  - The only left ..

Mining core assets

- Problem
  - the various controls were built with different architectures → impossible to mine components
- Approach
  - Start with one complete control system,
  - Allocate common features to core asset group, do domain analysis to define degrees of variation
    - Participate market + domain experts, product team, core team
  - Allocate specific features to product teams

Variability parameters

- Engine type (9 types)
- Cylinders (4 to 18)
- Displacement (3.9 to 164 liters)
- ECM on board (out of 12)
  - Electronic control modules
- Microprocessors (5 types)
- Fuel systems (10 types)
- Fuel (natural gas or diesel)

Variability mechanism

- #ifdef in sw components
  - 2600 ifdefs to select implementation versions
- Build time parameters
  - For control engineers, thousands
- User parameters
  - For the user, to be set after delivery, hundreds

- Domain analysis defined the parameters, and their bind time

Tools

- Build tools to handle
  - Declarations
  - Hardware configurations
  - Tool interfaces
  - File selection

Architecture

- The vision: two groups of layered components + functional level
  - Interface to hw, (to isolate from platforms)
  - Specific function
  - Group of functions (ex fuel system functions)

Architecture

- Weak point
  - Group function often did not match with lower levels
  - Product groups changed it
  - Architecture vision deteriorated

Launching the PL

- Select core group, plan work for them
- Select legacy sw as starting point for core assets
- Define architecture, modify core assets to comply with it
- Define project teams, define process (core assets → product), select tools
- Pilot first product, change and improve process
- Roll out other products, change and improve process

Company culture

- Collaboration oriented
  - 3 business units, none is independent, by choice
    - Habit to collaborate poses no problem in accepting core asset group idea

2 part organization

- Core asset group vs. product teams
- Core asset groups provides
  - Core assets
  - Knowledge on domain and assets
  - Feedback on developing a new product

2 parts - issues

  - Failed in many cases
- Staffing
  - Requires most talented resources
  - Projects not willing to let them go
  - People may feel to work far from key business area
- Morale
  - Product teams get rewards, core asset teams get blame
  - Core asset team perceived as dictators on evolution

Success factors

- For 2 part organization in Cummins
  - A charismatic PL champion
  - Company culture oriented towards collaboration
  - Funding model
  - Product based consensus on priorities for changes to core assets

Funding model

- Who pays for core asset group
  - Budget for core asset group established
  - Each BU pays in proportion to its sales
    - Not in function of use
    - Incentive to use core assets as much as possible

Core assets changes

- Weekly meetings between core asset group and all product groups
  - Product managers request changes, new features
    - Discussion / elicitation of commonalities
  - Shared prioritization of changes for core asset group
  - Advantages
    - Core asset group is released from deciding priorities (and appearing as 'dictator')
    - Core group must work as service to product groups
    - Product groups have to prioritize their requests
    - All product groups must define global priorities. Core assets perceived as common resource

Running the PL

- Weekly meetings to evolve core assets (PRs and CRs)
- All core assets under CM
- Defined workflow for PRs and CRs
  - All product teams access them
- Defined workflow for product creation
- A common toolset for
  - Requirements and traceability management
  - Architecture definition
  - Data dictionary
  - System build and configuration
- Effort and defects collected for core assets, effort for products

Running the PL

- Common tools for testing, automated unit and regression testing
  - Common for core assets and product teams
- Testing artifacts as first level core assets
- Core asset team performs generic testing
- Product team performs product testing
- Monthly review meetings
  - At architecture and algorithm level

Results (1994 to 2000)

- 20 products developed (20 software builds), 1000 separate engine applications
- 75% of product software comes from PL, on average
- Effort per product from 250 pm to 10
- 200 new features
- Common look and feel for products
- Developers are 'portable' through projects
- 100 sw staff, developing the same products would require 360 with previous approach

Lessons learnt

- Mine / reuse important assets
  - Architecture, components, requirements, tests
  - Small grain reuse easier, but with little pay off
- CM
  - As backbone of the PL, provides product builders with components, their changes, tehir history, and traces among them
- Product building tool
  - Paper guide written, but never used
- Reducing core assets releases
  - Each has to be maintained 'forever'
- Requirements and domain
  - Domain analysis to be performed by domain expert, and requirement analysis to be performed at company level to understand commonalities
- Architecture
  - Component based reuse. Components mined and reused. No architecture vision, components are connected at will by products. Sometimes core assets are changed by products.
  - Weak point, because of time pressure, no way to define and enforce common architecture

CCT

Ground CC

- Ground based spacecraft command and control system
  - Monitor spacecraft functions
  - Configure spacecraft service and payload
  - Manage spacecraft orbit and attitude
  - Mission planning
  - Control ground system equipment (antenna)
  - Format and retransmit payload commands
  - Log data and commands

Ground CC

- Usually one project/system per spacecraft
  - ~500KLoc
- Need to exploit commonality, to reduce cost
  - Different US agencies, NRO, Air Force
  - Different projects, DCCS, SSCS, C2
  - CCT started 1997 as a product line

Launching

- Developing a business case
- Developing acquisition strategy
- Funding
- Structuring the organization
- Technical planning
- Organizational planning
- Phases

Developing a business case

- Goals
  - Reduce life cycle cost
  - Reduce risks
  - Promote interoperability
- 49% to 89% commonality in requirements of DCCS, SSCS, C2
- Significant cost savings over 10 yrs period
- Similar development costs

Acquisition strategy, funding

- US Government (NRO, Usaf)
  - Funds and acquires (property of) PL
    - Also legacy and COTS
  - Subcontracts to Raytheon
    - Can also develop other systems using assets
  - Uses PL to build control systems

Organization

- NRO unit
  - CCT program office
- Raytheon units
  - Contractor program office (PM)
  - Program support (CM, QA)
  - Domain engineering and architecture
  - Component engineering
  - Application engineering
  - Test engineering
  - Training
  - Sustainment engineering (maintain PL)
- Working groups to distribute knowledge/decisions
  - Architecture, stakeholder WGs

Planning Documents

- CM plan
- QA plan
- PM plan
  - 6 increments, 2 years, 20 deliverables
- CCT sustainment plan (maintenance)
- CCT transition plan
  - Guide PL → project
- All on web site

Phases

- Asset development
  - Raytheon, 6 increments
- Asset sustainment and process refinement
  - Raytheon, at customer site, to adapt PL to project
- PL sustainment and improvement
  - NRO + Raytheon, evolution of PL

Guide to work with potential customers

- 1 Assessment
  - Candidate user fits into CCT vision?
  - Candidate user needs will alter scope of CCT domain?
  - CCT schedule will satisfy candidate user needs?
  - New candidate user will impact existing users?
  - Are there any contracting issues associated with supporting the candidate user?

- 2 Detailed evaluation
    - If previous assessment successful
  - Technical issues
    - % Candidate user requirements inside/outside PL
    - Benefits from extending PL
    - Compatibility of architecture
  - Schedule
    - Feasibility to meet candidate user needs
    - Impact on existing users schedules
  - Cost
    - Feasibility of calculating costs for potential user
    - Potential overall savings
  - Risk
    - For meeting commitments with candidate user
    - Increase of risk for existing users
  - Contract issues

Engineering activities

- Domain engineering and architecture
- Component engineering
- Application engineering
- Test engineering
- Sustainment engineering
- Training

Domain engineering

- Analyze and represent common requirements and variations across several systems
  - Analyzed DCCS, SSCS, C2
  - Defined scope of PL
    - Eg. Telemetry streams (up to 2) inside, persistence outside
  - Defined common requirements for PL, use case driven
    - Two categories, planning and execution, basis for logical view of architecture
    - Defines contractual basis
    - Common requirements described textually + use cases and sequence diagrams. Variation as variation points in use cases
  - Defined glossary

Architecture

- Architecture definition
  - UML, 4+1 view
  - Description of each component, variation points and variation mechanism
    - Components strictly coupled with architecture
    - Limitation on number of design elements and interaction mechanisms
  - CORBA as support with standard services
    - Naming, event notification, object persistence, callback
  - Additional services
    - Manipulation of time values, coordinates, printing, logging, data display
- Architecture evaluation
  - SAAM
  - Stakeholders meet and try scenarios of usage and modification of architecture

Architecture

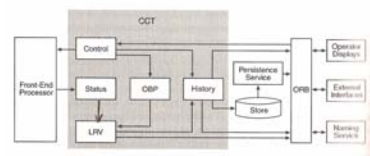
- Two subsystems
  - Real time (execution)
  - Non Real time (planning)

  - No direct interaction between the two susbsystems, except via R/W data on shared files
    - Planning subsystem writes command on file, execution reads file
- Components (grouped in categories) in each subsystem

Architecture

- Variations
  - Adding components outside PL
  - Variation points in components inside PL

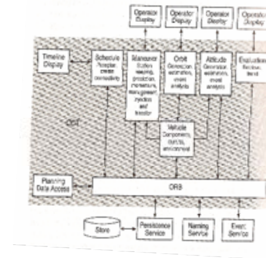
Execution



Execution – component categories

- Status. Receive telemetry data, perform integrity check, decode, define last recorded value (LRV, best estimate of raw value)
  - VP: data formats, data format error handling, validity checks
- LRV. Receive raw data from status, transform in engineering values, check limit, generate alarms
  - VP. Definition of new LRVs and integration with existing ones. Inheritance and parameterization
- Control. Encode client commands to spacecraft. Language to combine/automate commands.
  - VP. Formatting algorithms. Verification logic
- On board processing. Model computer memory status on spacecraft
  - VP. Extensions to memory model
- History. Log, retrieve data sent/received from spacecraft. Flat files.
  - Products responsible to provide other storage models (DB, ..)

Planning



Planning – component categories

- Orbit. Estimate orbit, propagate orbit.
  - VP. Use data from outside
- Attitude. Estimate and propagate attitude
  - VP. Use other algorithms or data
- Maneuver. Plan maneuver and generate command sequence (thruster firings). Spin stabilized and 3-axis stabilized algorithms
  - VP. Framework to provide complete component (high variation due to dependency on spacecraft)
- Vehicle. Provide model of spacecraft (unique features) for other components
  - VP. Same as above
- Schedule. Provide list of activities to be planned over time.
  - VP. Scheduling and conflict resolution algorithms
- Evaluation. Processing of telemetry and control data.
  - VP. Evaluation algorithms

Variation mechanisms

- Dynamic attributes
- Templates
- Inheritance
- Parameterization
- Function extension (callback)
- Scripting

Variation summary

Table 10.2 Component Variation across CCT

Domain	Component Category	Components	Variation
Execution	Status	4	1
	LRV	4	1
	Control	6	14
	On-board Processing	3	2
	History	5	7
	Other (playback)	1	1
Planning	Orbit	7	19
	Attitude	4	9
	Maneuver	5	12
	Vehicle	3	5
	Schedule	3	13
	Evaluation	4	3
Object services		10	13
Infrastructure		42	11
TOTALS		101	111

From architecture to product

- Select UI/GUI and DB products
- Select Corba vendor
- Address security needs
- Select / mine COTS components and legacy components. Determine implementation of CCT components.
- Package components as executable applications and allocate to nodes of network

Architecture evaluation

- What if Scenarios - planning
  - Inclusion of payload management in product
  - Replacement of DB with other, provided DB interface has to be changed
  - User provided orbit package to be used, with different inputs
  - More sophisticated scheduling, partly done onboard of s-craft requests/inputs from s-craft
- Execution
  - Inhibit commands
  - Change OS, change ORB
  - Dynamic reconfiguration of components by user
  - Prioritized (emergency) commands
  - Callback routine hangs up – recovery?

Component engineering

- Development of components
- Requirement: reuse DCCS components

Application engineering

- Development of C2 ground control as first product

Test engineering - CCT

- Unit testing of components
- Requirement verification of components
- Integration test
  - Exercising variation points
- System test – performance
  - Simulating actual user
- All steps repeated 6 times for 6 increments

Test engineering – C2

- C2 system used as first application
- System and performance test

Sustainment engineering

- Handles, at PL level
  - Problem reports
  - Change requests (new requirements)
  - Issues by potential users of CCT assets

Results

- product using CCT, estimate on 1997-2009 period
  - Development. 18% cost reduction
  - Sustainment. 27% cost reduction

Results

- C2 system
  - Problem reports, reduction 90%
  - Code developed, 75% reduction
  - Cost, 50% reduction
  - Schedule, 50% reduction

Lessons learnt

- Tool support inadequate
  - Models in different tools, hard to find, integrate and maintain, no traceability
    - UML, Rose
    - Requirements, Slate
- Domain analysis decisions not recorded
  - Why feature is missing or included
  - Would be useful for product users
- Architecture focus and documentation
  - Architecture focus not clear initially (toolkit development)
  - Architecture mostly in head of architects
  - Hard to understand for users

Lessons learnt

- Product builders guide inadequate
  - More emphasis on building, testing, documenting single components
  - Little support for integration/adaptation from user perspective
  - No documentation on overall performance in function of assets/variants selected
- Limited metrics
  - Reuse, defects
- Cross unit teams essential, and worked

References

- Clemens and Northrop, Software Product Lines
- Weiss and Lai, SPL Engineering
- Greenfield et al. Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools.