

**Multi-Dimensional Software Families:
Document Defined Partitions of a Set of Products**

David Lorge Parnas

Abstract

More than 30 years after it was proposed that a set of closely related programs would be easier to maintain than a set of independently developed programs, the software industry is still struggling with the complex problem of making updates to collections of “almost alike” products. This talk has four themes:

- A program-family might not be a software product line but there are advantages that accrue if a software product-line is a program family.
- Full exploitation of the program-family concept requires explicit design effort.
- Full exploitation of the program family concept requires recognition that a set of programs can and should be partitioned into families in many ways.
- Precise documents can be used to characterize families and subfamilies, and make it easier to maintain a well-designed set of program products.

1/1

1. "Program Family" vs "Software Product Line	1	28. Example: Program Function Document	28
2. Why are Program Families and Product Lines Confused?	2	29. Module Interface Document	29
3. A Personal (Brief) View of History	3	30. Example of TFM-Style Document	30
4. "Architecture of the IBM System/360" (1964)	4	31. Module Guide	31
5. "Design and Development of Program Families" (1976)	5	32. Module Internal Design Document	32
6. "Software Requirements for the A-7E Aircraft" (1977)	6	33. Document: Specification or Description	33
7. Procedure for Designing Abstract Interfaces (1981)	7	34. Parameter Evaluation Time	34
8. Software Product-Line Engineering (Weiss and Lai 1999)	8	35. Need for Precise, Complete and Consistent Documents	35
9. The Practical Motivation (review)	9	36. Need for Ease of Reference	36
10. Fighting Symptoms vs. Removing Causes	10	37. Eschew "Formal Methods"	37
11. Three Development Models for Program Families	11	38. Advice to industry - I	38
12. Two Level Tree (Domain/Product)	12	39. Advice to industry - II	39
13. Issues With the Two Level Tree Model	13	40. Advice to industry and Academia	40
14. Multi-Level Tree (1976)	14	41. Ideas for academics	41
15. Issues With the Multi-Level Tree Model (1976)	15	42. A Vision of Future Professional Software Development	42
16. Multiple Partitioning Approach - I	16	43. Keep Your Eye On the Goals	43
17. Multiple Partitioning Approach - II	17		
18. Multiple Partitioning Approach - III	18		
19. Documents as Predicates	19		
20. Document Defined Program Families	20		
21. Document Expressions	21		
22. Why Are These Ideas Useful?	22		
23. Abstract Commonalities vs. Common Artifacts	23		
24. Constraints vs. Enumeration of Possible Values	24		
25. Parameterized Documents: Commonalities, Variabilities	25		
26. Variabilities Can be Made Redundant	26		
27. Commonalities, Explicit Variabilities, Secrets	27		

1/1

"Program Family" vs "Software Product Line

When is a Set of Programs A Program-Family? (Parnas 1976)

- “When it is worthwhile to study programs from the set by first studying the common properties of the set and then determining the special properties of the individual family members”. (pragmatic definition)

What is a product line?

- “Set of products offered by producer to customers”

Product-Line is a commercial concept motivated by market analysis and usually determined by marketing people.

Program Family is an Engineering concept, determined by designer decisions.

- Several product lines could contain members of a family.
- Product lines could contain members of several families.

1/43

Why are Program Families and Product Lines Confused?

There are many benefits to having a product line be a program family.

- Reduced maintenance costs
- Faster, more predictable additions to product line
- More reuse
- Products can have company “image”.

The more these products have in common, the more advantages you have.

If you look at software products closely, you will find many **unnecessary** differences reducing benefits.

A critical research question is how **designers** can reduce or eliminate such differences and increase benefits.

2/43

A Personal (Brief) View of History

- Amdahl, G.M., G.A. Blaauw, F.P. Brooks, Jr. 1964: "Architecture of the IBM System/360," IBM Journal of Research and Development, 8: 87-101
- Parnas, D.L., "On the Design and Development of Program Families", *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 1, March 1976, pp. 1-9.
- Heninger, K., Kallander, J., Parnas, D.L., Shore, J., "Software Requirements for the A-7E Aircraft", NRL Report 3876, November 1978, 523 pgs.
- Britton, K.H., Parker, R.A., Parnas, D.L., "A Procedure for Designing Abstract Interfaces for Device Interface Modules", *Proceedings of the 5th International Conference on Software Engineering*, March 1981, pp. 195-204
- Weiss, D.M., Chi Tau Robert Lai, "Software Product-Line Engineering: A Family-Based Software Development Process", Addison Wesley 1999
-
- Many more books, papers, methods. Many authors unaware of these! Note a growing concern with artifacts, tools, rather than design principles.

3/43



"Architecture of the IBM System/360" (1964)

Double Revolution in Computer Hardware

•New instruction set and structure (not my taste)

(I hear the same ideas presented as new every few years.)

•One "architecture" for set of really different computers

Dispute: "Architecture" or "Facade"

Previously the implementation had driven the instruction set.

Family defined by a document: "Principles of Operation"

Read it before you pick your model from the product-line.

This was all that they had in common.

No "variabilities" discussed in this family-defining document.

4/43



"Design and Development of Program Families" (1976)

1969: Developers plan to develop three operating systems independently and meet to make them compatible later.

- Observation: This never works. The earliest design decisions in a project are usually the most difficult to revise.
- Common aspects should be decided first.
- Proposed a family tree in which only the leaf nodes were products. No product was descendant of another.
- Each level introduces sub-families.
- Documentation of shared decisions essential.
- Documents must be binding on all who follow.

1976 paper proposed that we make decisions in a different order (based on above) and document these decisions in the form of interface specifications.

5/43



"Software Requirements for the A-7E Aircraft" (1977)

Part of a "design for ease of change" effort.

Designers cannot make everything equally changeable.

Program designers can't know what will change.

Several versions in use at the same time.

As part of our requirements document we listed

- fundamental assumptions (things unlikely to change), which today would be called commonalities.
- expected changes, which today would be called variabilities.

These lists and the rest of the document defined a program family of which only a few members would actually be produced.

6/43



Procedure for Designing Abstract Interfaces (1981)

Another aspect of “design for change”.

Interfaces that won’t change for devices that will be replaced.

- Interface as a set of assumptions, can be reviewed by SMEs in that form and then translated into programmer’s API form.
- Based on documenting commonalities and translating variabilities that can not be hidden into parameters.
- Many advantages of (then) current practice (still widespread) based on either rewriting code or using a descriptive data structure.

Viewed as designing a family of programs.

Explicit up-front attention to time of change (rewrite, recompile, restart, pause, “hot-change”).

7/43



Software Product-Line Engineering (Weiss and Lai 1999)

A Family-Based Software Development Process

Detailed description of how to turn abstract design principles into a systematic industrial process.

Detailed examples based on industrial experience.

Recognizes the importance of documents; uses a combination of forms and informal documents.

Basically looks at a two-step model (domain, product) that limits the “tree” to a two level tree.

New idea of incorporating the commonalities in a program generation tool and language.

8/43



The Practical Motivation (review)

With such pressure to get the first product out (or the next product) why should we slow down now for future ones?

- Maintaining set of “almost alike” products is expensive and difficult.
- We want to avoid doing the same thing again and again.
- For users, unnecessary differences between products are annoying.
- Having a set of similar products to “move” through required changes slows a company down and increases its development costs.
- Documents that apply to all products in a product line, are not usually good for any one of them.

The documents you write must be really used and useful.

9/43



Fighting Symptoms vs. Removing Causes

The “Humpty Dumpty” syndrome:

- If software is not suitable for reuse, we build repositories and they remain empty or the contents are never used.
- If interfaces are complex and undocumented, we work on “teamwork”.
- If the requirements are not known, we call them “non-functional”.
- If we don’t document well, we work on information retrieval.
- If we don’t know what to say, we work on languages.
- If we don’t know which configurations work, we need variability management.

“Too little, too conventional, too late”, should be our motto.

We have to solve these problems when designing, not after they are apparent. You can’t put things back together then.

10/43



Three Development Models for Program Families

I classify approaches as follows

- Two Level Tree (Domain/Product)
- Multi-Level Tree (Family, Subfamily ... leaf node product)
- Multiple Partitioning Approach (newer)

Ordered from Least Flexible to Most Flexible.

11/43

Two Level Tree (Domain/Product)

Top Level (called Domain Engineering):

- Identify all things that will be in all members of family.
- These commonalities characterize the family.
- Build tools and components that are specialized to build family members.

Lower level (called Product Engineering):

- Use those tools to build products.

If you do a good job designing the “domain” the products will be easy to build and have a lot in common.

12/43

Issues With the Two Level Tree Model

Product Line in which any pair of members have a lot in common but there is nothing that they all have in common.

- Would still be useful to apply product line techniques.
- Is it a family? Not as defined in 1976 - but still useful to identify commonalities even if they aren't universal.
- What if some subset of products have more in common with each other than the whole set - why can't we have subfamilies?
- Must there be common artifacts? (/360)

13/43

Multi-Level Tree (1976)

Identify the commonalities for the whole family

Partition the family - identify additional commonalities

Repeat as far as useful

14/43

Issues With the Multi-Level Tree Model (1976)

Product Line in which any pair of members have a lot in common but there is nothing that they all have in common.

What if the order of certain decisions is irrelevant? How do you structure the tree? Is it a tree? (Dave Weiss).

What if two family members that are far apart in the tree still have some commonalities that are not root-node commonalities? What if there isn't a common ancestor with those commonalities?

Conclusion: 1976 tree approach is more useful than the special case (two level) but naive (too simple)

15/43



Multiple Partitioning Approach - I

Sets S_1, S_2, \dots, S_n partition set S if

- $S_1 \cup S_2 \cup \dots \cup S_n = S$
- $(S_i \cap S_j \neq \emptyset) \Rightarrow (i = j)$

Alternatively, using predicates

- Predicates P_1, P_2, \dots, P_n with domain that includes set S partition S if
- $P_1 \vee P_2 \vee \dots \vee P_n = \text{true}$
- $(P_i \wedge P_j \neq \emptyset) \Rightarrow (i = j)$

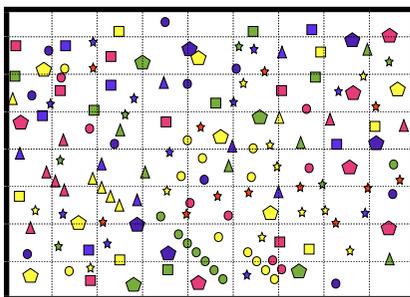
The subsets are partitions in the partitioning.

A set may have more than one partitioning.

16/43



Multiple Partitioning Approach - II



- We can partition by colour, shape, row of center, column of center, grid of center, and combinations. (e.g. green square)
- Every partition is a family.
- A product may belong to more than one family but only to exactly one family in each partitioning.

17/43



Multiple Partitioning Approach - III

A product line may be partitioned in many ways.

Each product belongs to many families. There will be one for each partitioning.

Examples:

- partition by platform (hardware)
- partition by software platform
- partition by set of displayable natural languages
- partition by input panel
- partition by display type
- partition by programming language used
- partition by interface type (e.g. protocol)
- partition by price considerations

18/43



Documents as Predicates

If you compare a description with the object that it purportedly describes, you can tell if the document is *true* or *false*.

If you compare a specification with an object that it purportedly describes, you can tell if it satisfies the specification or not.

In this way, a document is a predicate on a set of objects. It partitions that set of objects into (at least) two sets.

Parameterized documents partition more finely.

If the document contains a useful amount of information that you should read before studying an individual object, the document describes or specifies a family.

19/43

Document Defined Program Families

Precise documents partition sets of programs.

- Treat them as predicates

A parameterized document defines a set of documents, one for each possible set of values for the parameters.

- If there is a set of programs such that every program in the set is covered by exactly one of those documents, that set of programs is a family and the document defines and partitions that family.

Alternative documents:

- If we have a set of documents, D, and a set of programs P such that every program in P is covered by exactly one of those documents, P is a family and D defines and partitions that family.

What you should read about all members before studying the individual is in the defining document.

20/43

Document Expressions

Since we can treat documents as predicates, we can combine them using the operators that are used in predicate expressions (\wedge , \vee , \neg , \rightarrow) to get *document expressions*.

Using document expressions, we can identify a subset of programs. One application of this is for automated update systems (but it is also useful for manual maintenance).

21/43

Why Are These Ideas Useful?

These ideas allow you to design, develop family of programs with a lot in common without restrictions.

What they have in common is precisely documented.

When adding a new program to the family, the documents can be used to constrain the developers of that program.

The precisely documented commonalities and the precise documentation of their differences make this family easier to maintain and extend by adding new products.

If it isn't documented it is less likely to be a family in any useful sense because the true commonalities will be limited.

22/43

Abstract Commonalities vs. Common Artifacts

Current papers seem to assume that product lines must share code artifacts.

- The original IBM /360 family did not share physical components. It was defined by an abstraction. Same for later hardware product lines.
- My intel-based Mac and PowerPC Mac are part of a family.
- Sharing artifacts has obvious advantages, but sharing abstractions is possible when artifacts cannot be shared and has equally real advantages for users and developers.
- Sharing abstractions allows rapid progress in implementation.
- In many applications you get a shared “look-and feel” with very different components.
- Sharing abstractions only works if the abstraction is documented. Otherwise commonalities will be hard to discover and will evaporate.

23/43



Constraints vs. Enumeration of Possible Values

Once variabilities have been identified, it is usual to enumerate the possible values.

This is “backward looking” not “forward looking”.

“We cannot see into the future”.

However, we can write down constraints that must apply in the future. We must capture what we know and we do know a lot.

If we can find no reason that something is impossible, (i.e. if no constraints rule it out), we must prepare for it.

24/43



Parameterized Documents: Commonalities, Variabilities

The document(s) that define a family can contain parameters.

- The document captures the commonalities.
- The parameters are the explicit variabilities.

If the families contain alternatives that are too different, we introduce “choice” parameters in the document. The values of these parameters identify one of several alternative documents.

The choice parameters are also variabilities.

25/43



Variabilities Can be Made Redundant

It is common to talk about two lists, variabilities and commonalities.

Every fact about a family member is either a commonality or must be a specific set of values for the variabilities.

If you state all of the commonalities, you have an implicit definition of the variabilities.

If you have a parameterized document that states all of the commonalities, you have a way of classifying systems.

There are usually other, hidden, variabilities.

26/43



Commonalities, Explicit Variabilities, Secrets

If we have a document describing a set of programs, we can distinguish 3 kinds of information.

- **Commonalities:** the information given in the document.
- **Explicit variabilities** (parameters whose value can vary among program family members)
- **Secrets:** Information not given in the document. These are implicit variabilities.



Example: Program Function Document

Specification for a search program

$$(\exists i, B[i] = x) \mid (\forall i, ((1 \leq i \leq N) \Rightarrow B[i] \neq x))$$

j'	B[j'] = x	true	∧ NC(x, B)
present' =	true	false	

Commonalities:

- External Data Structure (int array B[1:N]); int j, x; bool present
- required effect on that data structure
- functional nature of effect indicates no need for internal memory

Parameters/explicit variabilities: integer N

Secret: Sort Algorithm, internal variables

This defines the family of programs.



Module Interface Document

Method: Possible values of each output given as function/ relation of history of input and output values.

No mention of state variables (actual or otherwise).

Commonalities:

- I/O variables as declared in document
- I/O behaviour (as above)
- Parameters appearing in interface document

Variabilities:

- Values of parameters

Secrets:

- Internal Data structure, abstraction function, program functions



Example of TFM-Style Document

Software Quality Research Laboratory - University of Limerick - Ireland

Keyboard Checker: Tabular Expression

N(T) =

		~(T = _) ∧			
		N(p(T))= 1	1 < N(p(T)) < L	N(p(T))= L	
keyOK	~keyesc ∧	2	N(p(T))+1	Pass	
			N(p(T)) - 1	N(p(T)) - 1	
			N(p(T))	N(p(T))	
	~keyOK ∧	1	1	N(p(T))	N(p(T))
			1	N(p(T))	N(p(T))
		Fail	Fail	Fail	
	1	N(p(T))	N(p(T))		



Module Guide

Hierarchical decomposition giving secret for each module.

Each level must partition the module above.

Parameters for (partial) inclusion of a component.

Alternate decompositions indicated by choice parameters.

Family characteristics:

- Possible structures indicated by choice and inclusion parameters.

Variabilities specified by parameter values.

Secrets (some to be told in other documents)

- interfaces, internal design of modules, algorithms.

Hierarchical tabular format for module guide proposed by Marius Dragomiriou is useful.

31/43



Module Internal Design Document

Further reviewable decisions before coding.

- Complete data structure
- Abstraction function
- Program function specifications for externally usable programs

Alternative designs, partial inclusion, and other parameters will capture the explicit variability.

Secrets (not specified) are the algorithms - but not their functions.

Data structure defined using programming language

Tabular notation useful for functions.

32/43



Document: Specification or Description

Description: Facts about the product.

Partial specification: Description in which the only facts reported are requirements.

(Full) Specification - Specification reporting all requirements for a product.

“Contract” not a good word, includes more than spec.

We can describe an existing program family or specify a future one with the same documents. The purpose of each document must be stated.

The phrase “specification language” makes no sense to me with these definitions of “specification”.

33/43



Parameter Evaluation Time

We must specify when a parameter may change(1980).

- Redesign time
- Code generation time.
- During System restart
- Brief pause in service
- While offering uninterrupted service (old idea new word)

Each time requires different implementation techniques.

- There are no restrictions.
- Code must contain the parameters explicitly.
- Data structure must be dynamic.
- Program to update data structure must be present.
- Parameters dealt with as program variables.

There is nothing new here, a “dynamic product line” was offered by Wang Co. in its word processors.

34/43



Need for Precise, Complete and Consistent Documents

Vague documents allow people to work at cross-purposes.

Incomplete documents allow people to guess or base their work on false information.

Inconsistent documents allow people to confidently work using incompatible assumptions.

Natural Language (or Esperanto) won't work.

Mathematics is the key to getting the necessary properties.

It is not the axiomatic or equational mathematics that you see from CS theoreticians. It is the "evaluation" based mathematics that Engineers almost always use. (no search, just evaluation)

35/43



Need for Ease of Reference

Implementers do not want to search for information.

A strict set of organizational principles is needed.

- A place for everything
- Everything in its place
- Missing information must be obvious
- Inconsistent information must be detectable
- Correctness can only be checked by review and testing.

Example:

- With the 1977 A-7 document, I could answer questions looking at no more than 7 pages (paper document, no search program).
- With tabular mathematical expressions, questions are usually answered by using 2 headers to find one cell.

36/43



Eschew "Formal Methods"

The name was a fatal mistake:

- It took people away from classic applied mathematics towards "theory".
- They should have learned how to use existing conventional mathematics
- Instead, field invents "unripe" mathematics

Present oversimplified models, not accurate documents.

Not designed for readability or ease-of-reference.

No agreed semantics, continual modification.

Based on philosophy, logic, not Engineering mathematics.

Based on deduction (search) not expression evaluation.

The only "new" problems are piece-wise continuous functions and expressions using partial functions.

Technology transfer effort was a harbinger of failure.

37/43



Advice to industry - I

The product-line is a marketing decision.

Making your product-line a program family is Engineering.

Anyone can have a product line.

- It will only be a family if you use precise documents to define it.
- Without binding documentation there will be unnecessary and undocumented deviations, and too few common properties to be useful.
- If you want it to be "worthwhile to study programs from the set by first studying the common properties of the set and then determining the special properties of the individual family members," you should document those properties. (Marius Dragomiriou).
- You won't get the full benefit of these concepts without documenting.

38/43



Advice to industry - II

Learn to produce professional documents, not stories

- You won't be the first.
- It has gotten easier through research.

Learn to use professional documentation for reviews, testing, inspection, etc. .

Define your family by a document expression.

Pick your new member by specifying parameter values.

Use evaluation to obtain product-specific documents.

Test the product against the documents.

If you need to extend the family do it by revising the documents that define that family.

39/43

Advice to industry and Academia

Teach How To Design.

Good Up-Front Design yields Simpler Documents and Simpler Document Structure.

I am repeatedly amazed at the number of developers that I meet who do not understand well-known design principles.

Patterns are not a substitute for principles - often repeat mistakes.

Design is learned by doing, not by listening.

If your instructors give long “introductory” lectures, they are doing the wrong thing and may be the wrong people.

Instruction must be “skills-based”, i.e. teach doing.

40/43

Ideas for academics

Don't play the “numbers game” (CACM, Nov. 2007, page 19)

- Try it successfully before you sell it.
- One good paper trumps 20 forgotten ones.

It's not the language; its the design!

Documentation is the ideal application area for many CS technologies:

- theorem provers
- computerized algebra systems
- language processors

Eschew Undefined Modeling Languages (UMLs).

Go back to basics; remember the goals.

41/43

A Vision of Future Professional Software Development

Software Developers Designing through documentation

- Making and recording design decisions in precise complete documents
- Reviewing those documents against precise standards
- Implementing using the documents
- Testing and inspecting against those documents
- Keeping documents up to date
- Finding reliable information quickly in those documents.

Designing Product Lines as Program Families

Defining Program Families by Document Expressions

Defining individual products using Document Expressions.

Every Product has an individualized set of documents.

42/43

Keep Your Eye On the Goals

- Maximizing commonality (including abstractions)
- Exploiting that commonality
- Communicating that commonality (including abstractions)
- Maintaining that commonality while extending the family

P.S. *This talk is part of a large family of talks. The commonalities are diffuse, abstract, and undocumented. I am sorry!*