A Case Study of Software Reengineering

M. Sc. Seminar

Harry Hornreich

Advisors: Prof. Daniel Berry & Prof. Noah Prywes

Motivation

The problem of maintaining and enhancing existing legacy systems has been recognized as a major problem in software engineering.

Researchers have proposed solving this problem by organizational changes and methods for systematic software reuse.

However, the transition from large, complex legacy applications to a future based on software reuse and automatic program generation has proved very difficult.

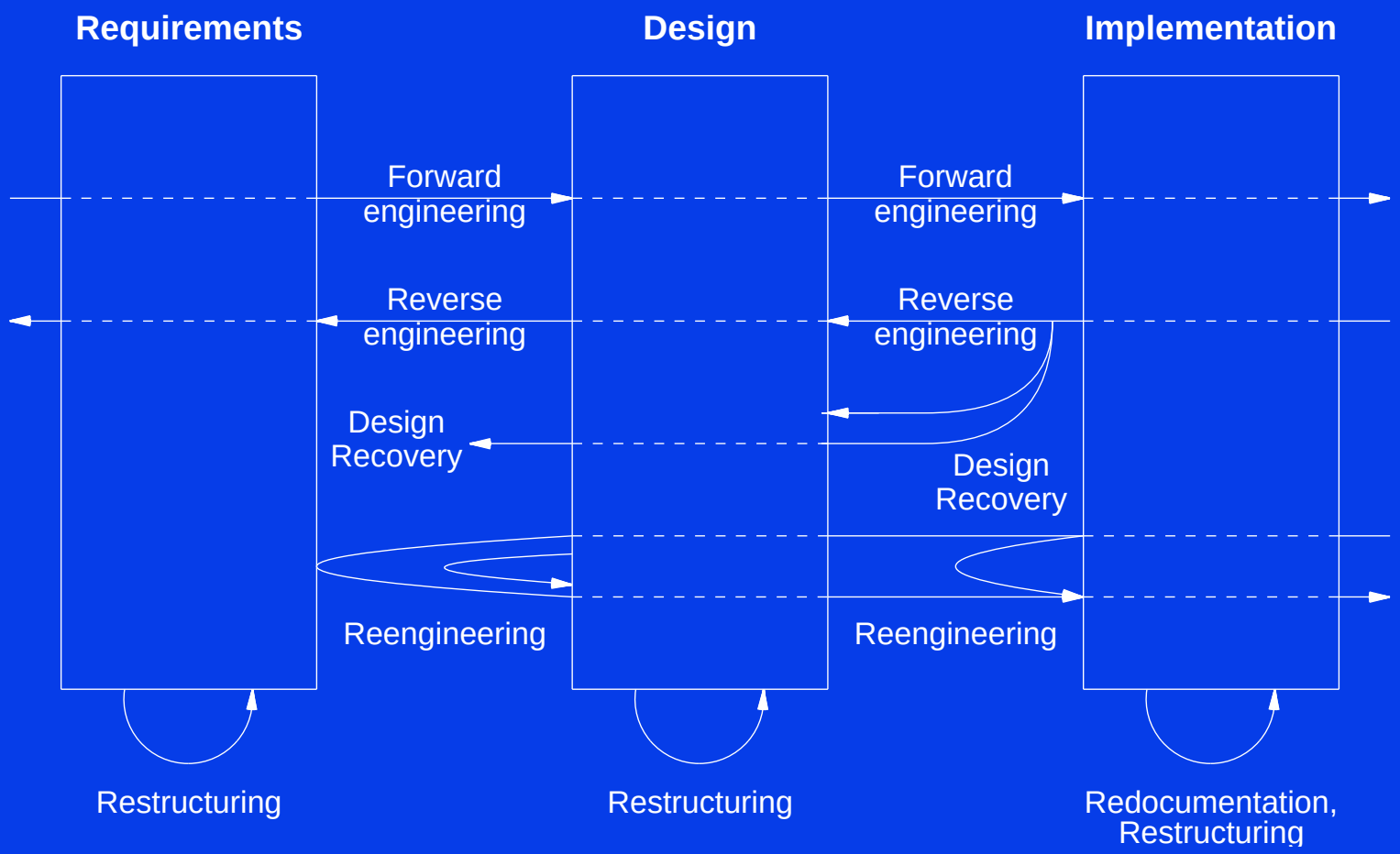
Software Maintenance

Software maintenance **is the "modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changing environment" (ANSI).**

Types of maintenance:

- **Corrective**
- **Adaptive**
- **Perfective**
- **Preventive**

Relationship Between Terms



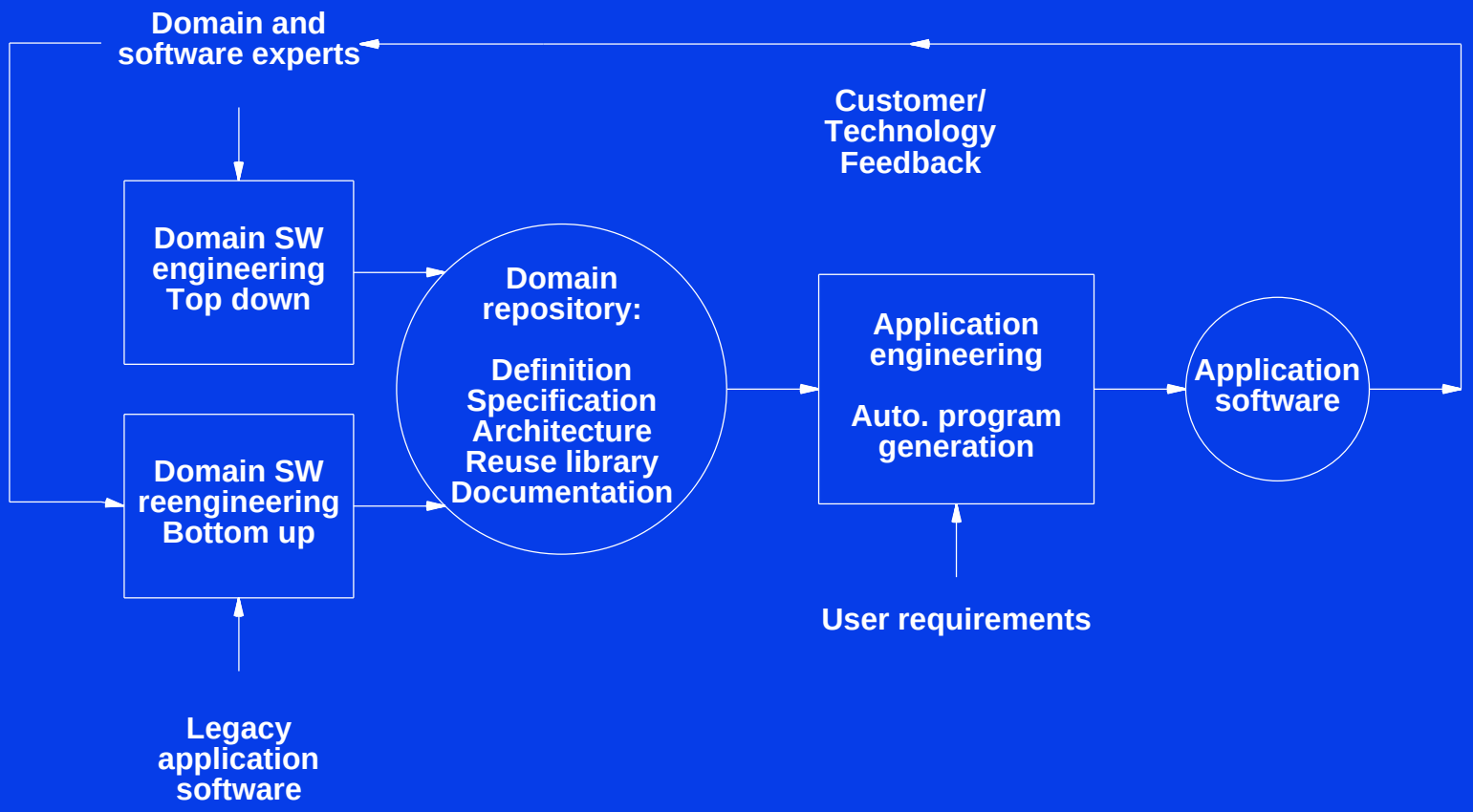
Synthesis

- **Prescribes an ordered sequence of steps for the management, analysis and specification of a *domain*, which contains the architecture of a product family of reusable software components and the decision rules needed for their selection.**
- **The top-down process of creating the domain is called *domain engineering.***
- **New applications are constructed by selecting components from the domain, as indicated by the decision rules, in a process called *application engineering.***

Domain Reengineering

- **Ahrens and Prywes (A & P) have augmented the top-down domain engineering process with a bottom-up *domain reengineering* process that extracts architecture, design, business rules, etc. from legacy software.**
- **New applications are still constructed using application engineering.**

Augmented Transition Method



Thesis Objectives

- To evaluate the proposed domain reengineering process.
- To reach conclusions on the required enabling technology for the reengineering process.
- To develop a method for the evolutionary development of a domain according to external requirements.
- To reach conclusions on the required enabling technology for the new method.
- To refine the theory of SWUs to support the above processes.

The Experiment

- A *case study* can show the effects of a technology or method in a typical situation, but cannot be generalized to every possible situation.
- Case studies are not as scientifically rigorous as *formal experiments* but they can provide us with sufficient information to judge if a method has any promise in it, and whether it is worth to proceed to controlled experimentation.
- A 1965 experiment to show that interactive programming is more effective than batch programming failed to produce significant results because the effect of the independent variable was drowned out by individual differences in programmers of equal experience. One was found to be 28 times more effective than another of equal experience.

The Experiment - Cont.

- Our case study is a “which is better” type of case study: A & P versus conventional maintenance.
- Our hypotheses are that the A & P method:
  - requires less time to produce an application
  - produces more reusable code
  - requires less code modification to produce an application

Experiment steps

1. Select legacy code program *P* as pilot
2. Domain reengineer *P* and create initial domain
3. Devise a set of requirements *R'* for *P'*
4. Experimenter and control create individual versions of *P'* and test them against the same set of tests
5. Devise another set of requirements *R''* for *P''*
6. Experimenter and control create individual versions of *P''* and test them against the same set of tests

The following measurements were collected:

- Implementation hours
- Number of added, deleted, and modified code lines

Experiment Validity

- We selected a typical legacy program as the pilot.
- The experimenter had no previous knowledge of the pilot program.
- The experimenter was not disclosed the requirements until he needed them.
- Only discussion of the requirements was allowed between the experimenter and control.
- Similar versions were compared against a common base-line before proceeding to the next step.
- The control had a clear advantage over the experimenter (years of programming, knowledge of the domain, knowledge of the pilot, previous knowledge of the requirements).

The ffortid Program

ffortid is a ditroff post-processor for handling Hebrew, Arabic, Persian, as well as other right-to-left languages.

ffortid rearranges each input line so:

- Text in right-to-left fonts is in its correct direction.
- Lines containing Arabic and Persian are left justified by stretching words and not by adding additional white space.

Why ffortid?

- ffortid is a typical legacy program (on a small scale):
  - Written over a long time span (1983-1991)
  - Written by several programmers (3)
  - Had several versions (1.0, 2.0 & 3.0)
  - In working condition and in use
  - No original design documents
- ffortid is a good candidate for experimentation:
  - Reasonably sized (2510 lines)
  - The experimenter had no previous knowledge of the program

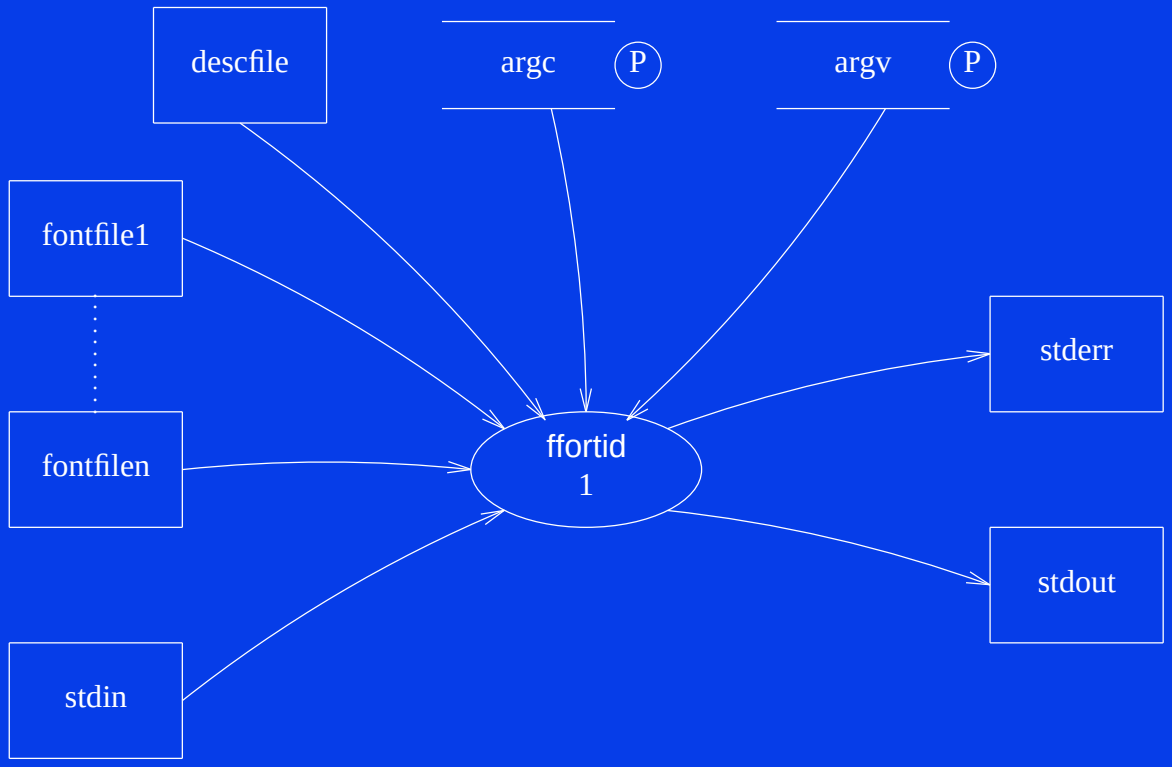
Software Unit Theory

A *Software Unit* (SWU) is a well-defined component of a software system, that provides one or more computational resources or services.

- The *scope of a SWU* is the body of code which it abstracts.
- The *capabilities of a SWU* are the services it can provide.
- The *interface of a SWU* is a description of how its services can be accessed by its clients and how these services affect other SWUs.
- The *requirements of a SWU* are the services it depends upon in order to provide its own services.

Service Flow Diagrams

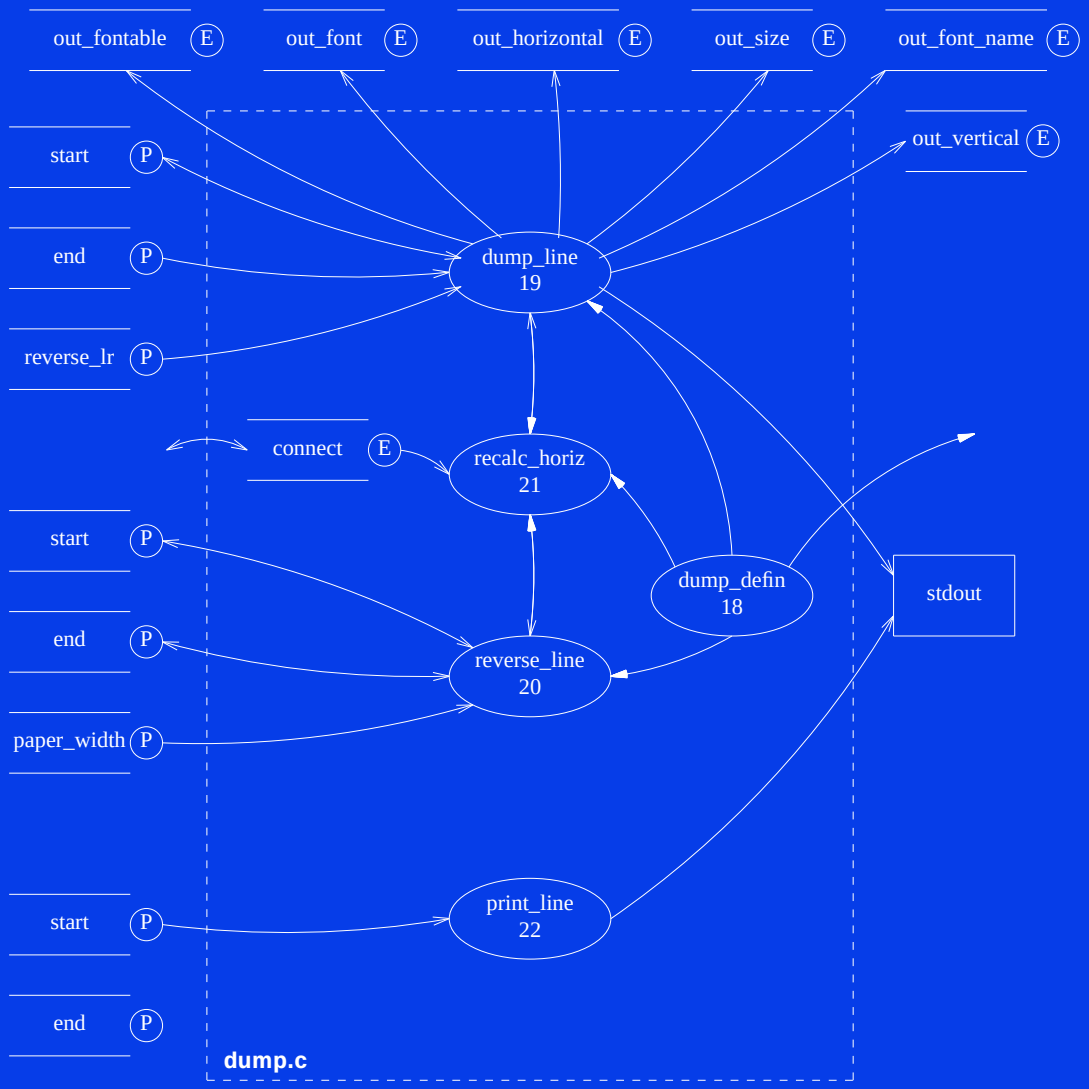
A Service Flow Diagram (SFD) is a graphical description of the service flow between one or more SWUs.



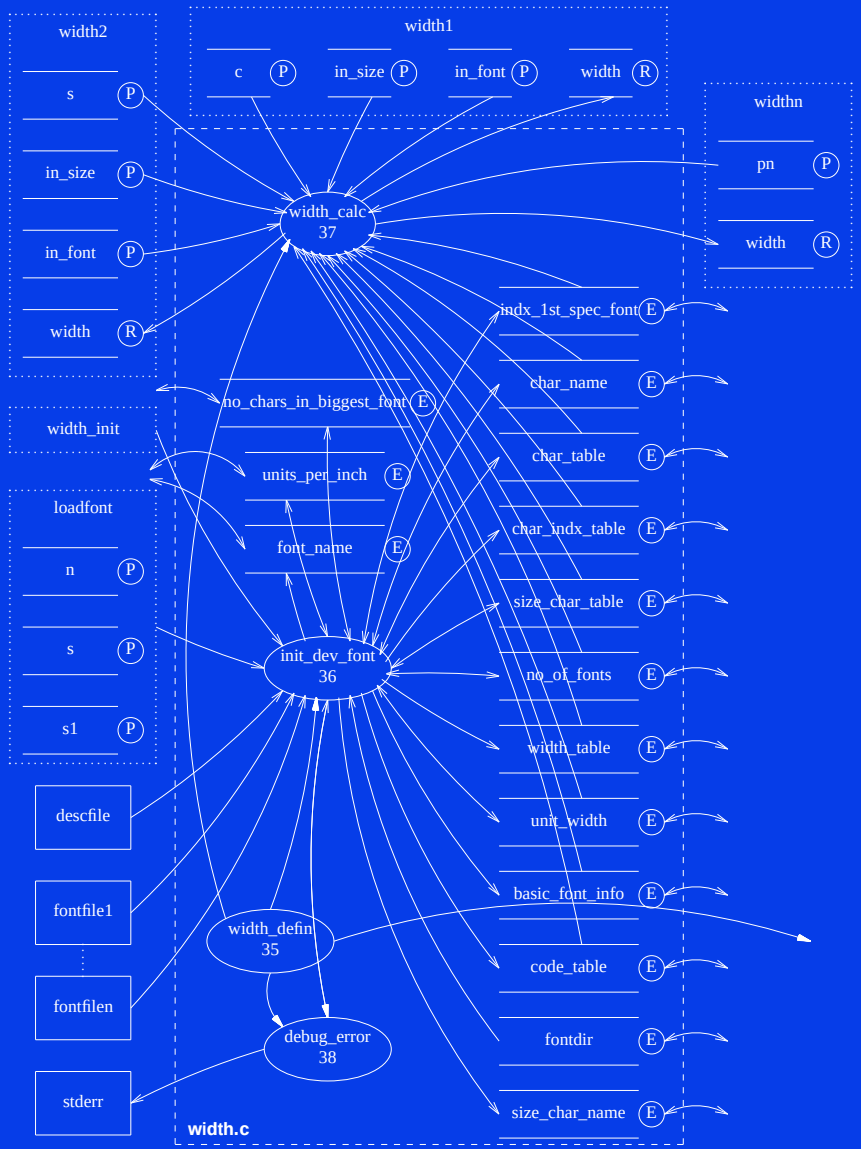
SFD Icons

Software Unit  XXX is the name of the SWU. *n* is its number (optional)	**IO File**  XXX is the name of the file	**Local Variable**  XXX is the name of the variable
Parameter Variable  XXX is the name of the variable	**Return Variable**  XXX is the name of the variable	**External Variable**  XXX is the name of the variable
SWU Borderline  XXX is the name of the SWU	**Parameters Group**  Groups parameters of *func* for SWU entry point	**Data Flow Relationship**  Data flows from SWU A to SWU B
Bi-Directional Data Flow Relationship  Data flows from SWU A to SWU B and vice-versa	**Call Relationship**  SWU A calls a function in SWU B	**Use relationship**  SWU B uses declarations or definitions in SWU A

SFD of SWU with sub-units



A complex SFD



Reverse Engineering a SWU

Reverse Engineering a SWU has the two major goals:

- Recreate the architecture and design of the SWU
- Assist understanding of the SWU

And several minor goals:

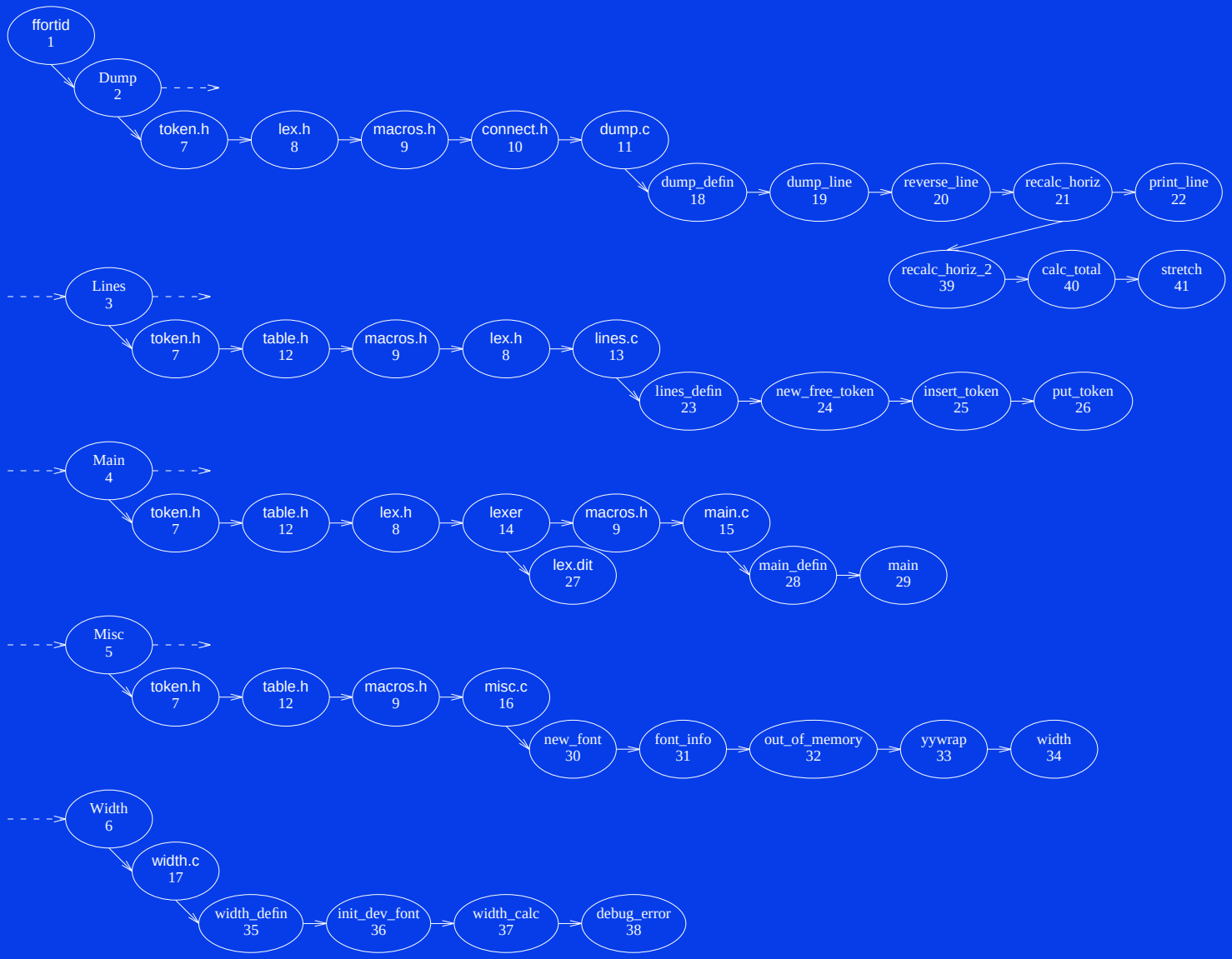
- Identify SWUs that are candidates for reuse
- Recover information not documented in the code
- Detect incorrect documentation
- Detect errors in code
- Detect side-effects in SWUs

Reverse Engineering a SWU - Cont.

Reverse engineering steps:

- **Partition the SWU into sub-units and continue recursively.**
- **Partition according to the syntactical structure of the SWU and the principles of coupling and cohesion.**
- **Partition down to the desired abstraction level of good reuse candidates.**
- **Determine the attributes of the sub-units in a bottom-up fashion.**

ffortid 3.0 Decomposition



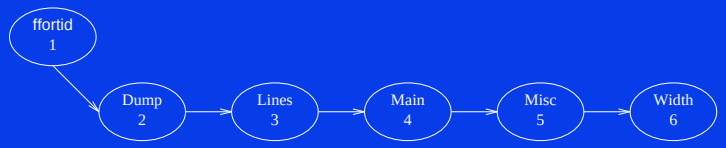
ffortid 3.0 SWU #1 Page

Software Unit #1 — ffortid

1.1 Software Unit Type

Program. (lex.h, lex.dit, token.h, macros.h, connect.h, table.h, dump.c, lines.c, main.c, misc.c, width.c)

1.2 Scope Diagram



1.3 Capabilities

ffortid takes from its standard input dtroff output, which is formatted strictly from left-to-right, finds occurrences of text in a right-to-left font and rearranges each line so that the text in each font is written in its proper direction. Additionaly, ffortid left and right justifys lines containing Arabic & Persian fonts by stretching connections in the words instead of inserting extra white space between the words in the lines.

1.4 Interface

command line options:

ffortid [–**r***font-position-list*] ... [–**w***paperwidth*] [–**a***font-position-list*] ...
[–**s**[**n**|**f**|**l**|**a**]] ...

The **-r***font-position-list* argument is used to specify which font positions are to be considered right-to-left. The **-w***paperwidth* argument is used to specify the width of the paper, in inches, on which the the document will be printed. The **-a***font-position-list* argument is used to indicate which font positions, generally a subset of those designated as right-to-left (but not necessarily), contain fonts for Arabic, Persian or related languages. The **-s** argument specifies the kind of stretching to be done for all fonts designated in the **-a***font-position-list*

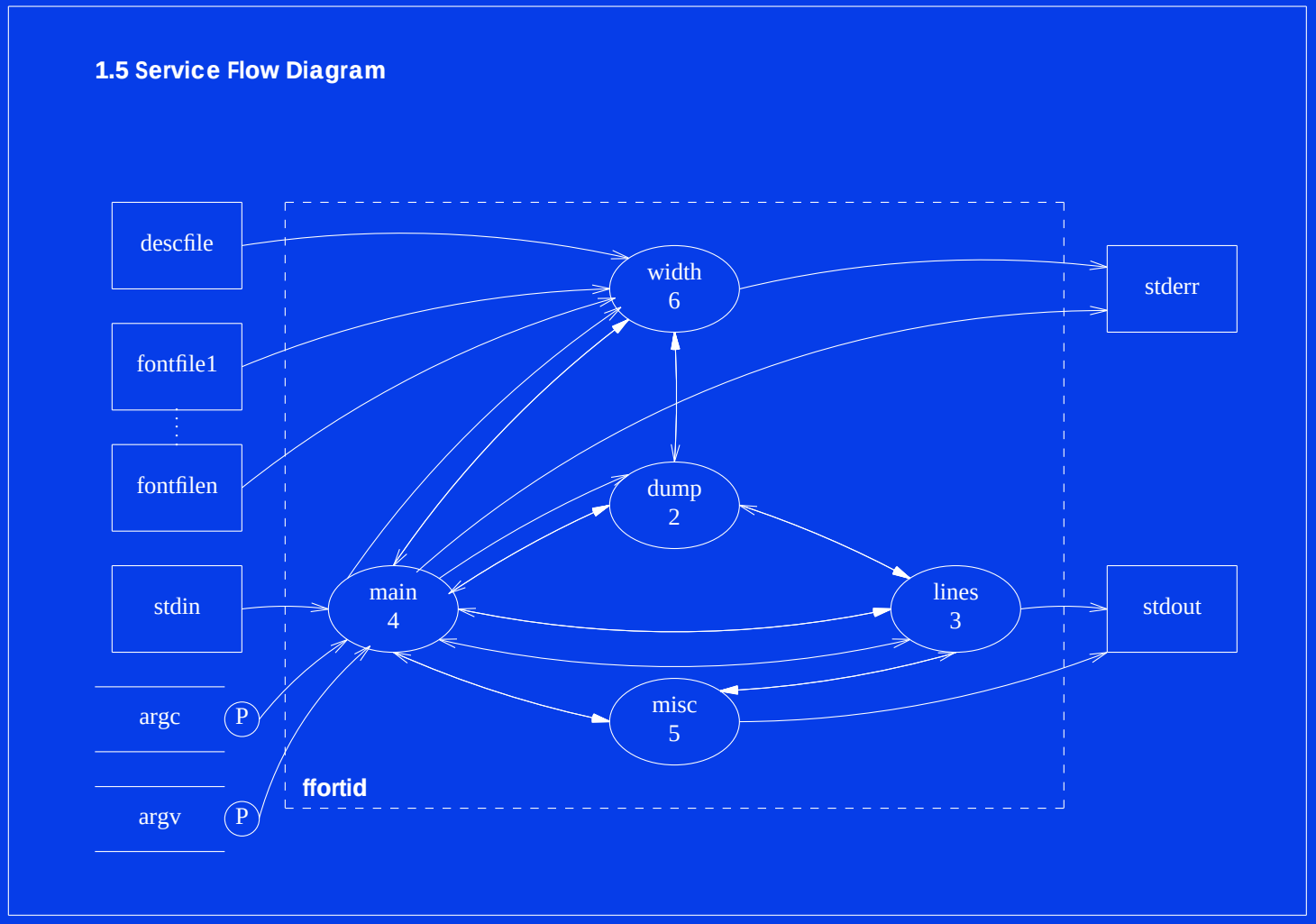
1. **-sn** — Do no stretching at all for all the fonts.
2. **-sf** — Stretch the last stretchable word on each line.
3. **-sl** — Stretch the last stretchable word on each line up to a maximum length.
4. **-sa** — Stretch all stretchable words on the line by the same amount.

The default is no stretching at all.

Manual connection stretching can be achieved by using explicitly the base-line filler character **\(hy** in the dtroff input. It can be repeated as many times as necessary to achieve the desired connection length.

Side effects:
1. ffortid reads dtroff output from stdin and prints dtroff output to stdout.
2. ffortid prints encountered errors to stderr and halts program.
3. ffortid allocates and frees memory from the heap. If out of heap memory ffortid prints a ``out of memory´´ message to stdout and halts program.

ffortid 3.0 SWU #1 SFD



Conclusions from Reverse Eng.

- Reverse engineering is like archeology
- We did not attempt to understand the details of the code only its structure
- We added our own comments
- SWU pages must be constructed bottom-up
- A query mechanism on code is necessary
- SFDs were not useful for very low level abstractions
- SFDs were done last but helped give global picture
- Reverse engineering large legacy applications must be performed using dedicated CASE tools

CASE tools for Reverse Eng.

- Can suggest alternatives and implications of a partition
- Generate automatically the interface and requirements
- Generate automatically most of the side-effects
- Generate automatically SFDs
- Extract useful comments from the code
- Handle most of the “paperwork’’ involved
- Build a database for queries and view generation
- Allow the addition of our own comments
- Allow easy correction of previous partitioning decisions

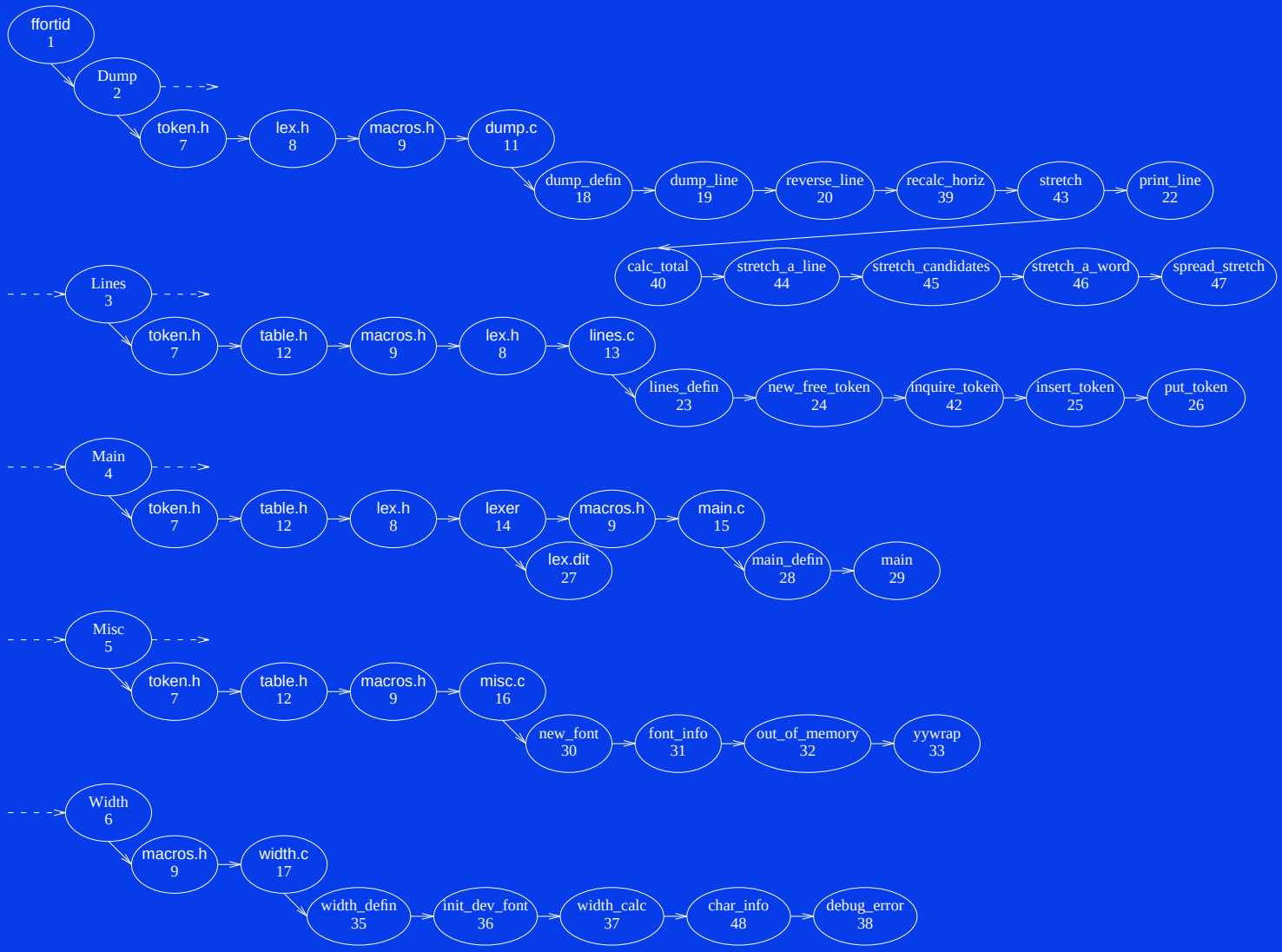
The Initial Domain

- Domain definition: “The family of ditroff post-processor applications capable of reversing text in right-to-left fonts and capable of left justifying lines by stretching Arabic text’’
- Each SWU in ffortid 3.0 decomposition becomes a reusable component in the domain
- Currently their is only one application SWU and one method of combining the components
- The domain’s reusable components are not always very adaptable or easily reusable
- We avoided performing preventive maintenance and instead wanted to observe the evolutionary development of the domain

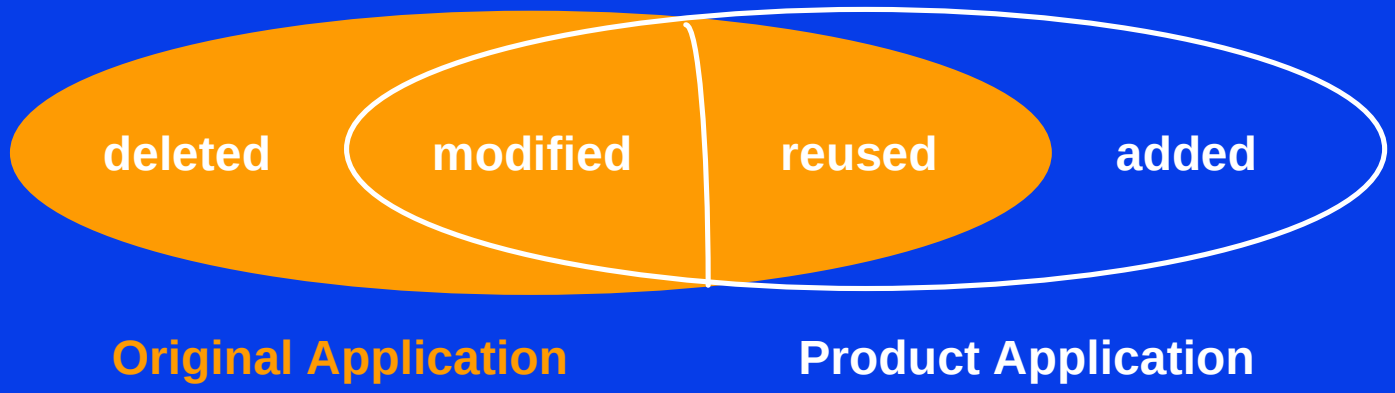
Requirements Implementation

- Major implementation steps:
  - List requirements at user level
  - Express the requirements in a more detailed fashion as capability, interface and requirement modifications of the application SWU
  - Implement the access interface modifications
  - Implement the capability and requirement modifications
  - Implement the result interface modifications
- Each single modification is implemented in the same fashion:
  - Perform a focused top-down search through the SWU hierarchy for the low-level SWUs possessing the attribute to be modified.
  - A single modification can result in modification side-effects

Updated Domain Architecture



Measuring Reuse



Reuse Ratio = reused lines / original lines

Modification Ratio = modified lines / original lines

Addition Ratio = added lines / product lines

Results

	Del. Lines	Mod. Lines	Added Lines	Final Lines	Implementation Time
Experiment 4.0	684	22	969	2795	—
Control 4.0	784	36	1997	3723	—
Experiment 5.0	126	23	947	3616	39
Control 5.0 †	44	82	789	4468	77 - 94.5

† All results regarding ffortid 5.0 are not final

Results Analysis

	Reuse Ratio	Modification Ratio	Addition Ratio
Experiment 4.0 Control 4.0	72% 67%	1% 1.5%	35% 54%
Experiment 5.0 Control 5.0 †	95% 97%	1% 2%	26% 18%

† Not final results

Conclusions

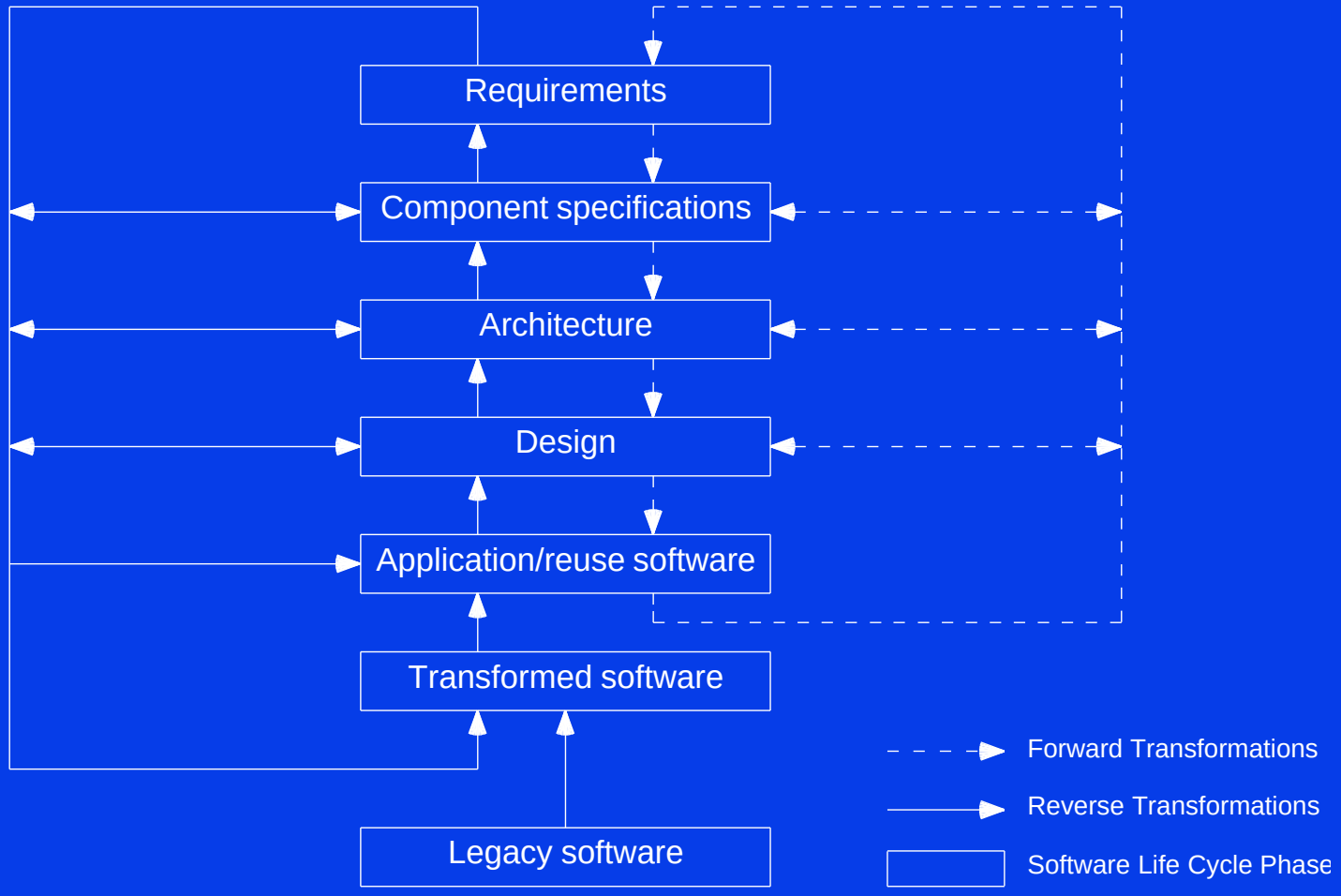
- There is no significant difference between the reuse and modification ratio of both methods.
- The product produced by the experimenter (ffortid 4.0) is more reusable than the control's:
  - smaller time to perform changes in ffortid 5.0
  - smaller product size in ffortid 5.0
  - system documented in ffortid 5.0
  - higher quality code in ffortid 5.0
- SWU theory is applicable to real software and can be used to document the architecture and design of existing systems and to help the modification of these systems.
- Domain reengineering with the use of dedicated CASE tools can improve the maintenance of of legacy systems.

Problems in Current Life Cycle Models

Assumptions in current life cycle models:

- **Maintenance is a separate life-cycle phase.**
- **New software applications require new software development.**
- **New applications based on reusable components can and should be developed only in a top-down manner.**
- **CASE technology for forward software development can perform almost all maintenance.**

The Legacy and Reuse Life Cycle Model



ffortid History

- ffortid 1.0 (1983-1984)
  - Author: Cary Buchman (UCLA)
  - Capabilities: Hebrew only
- ffortid 2.0 (1986)
  - Author: Mulli Bahr (HU)
  - Major Modification: Output Optimization
- ffortid 3.0 (1989-1991)
  - Author: Johny Srouji (Technion)
  - Major Modification: Arabic with connection filler stretching

All these people worked with Prof. Berry who was the customer and provided project continuity.

ffortid Source Files

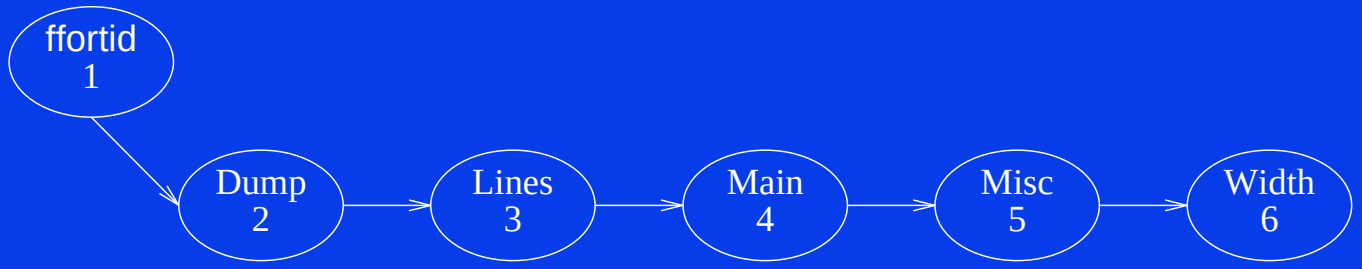
Num	File	Size (lines)	Functions
1	lex.h	30	-
2	lex.dit	37	-
3	token.h	34	-
4	macros.h	20	-
5	connect.h	256	-
6	table.h	18	-
7	dump.c	704	10
8	lines.c	296	6
9	main.c	506	1
10	misc.c	129	5
11	width.c	480	10
Total		2510	32

SWU Theory - Cont

- The *type of a SWU* categorizes it into one of several types of similar service providers
- The *environment of a SWU* is all the software in the context in which it is used which is not in its scope
- The *interface of each SWU* service can be divided into its access interface and its result interface
- A *resource* is a SWU service that does not have a result interface
- results not returned through the access interface are called *side-effects*

Software Sub-units

- Every non-trivial SWU can be decomposed into its software sub-units
- The scope of each of the sub-units must be mutually exclusive and the sum of the scopes must be equal to the scope of the parent SWU
- A SWU is decomposed according to some partitioning criteria
- The architecture of a SWU is a rooted tree of SWUs
- SWU decomposition is shown graphically using a scope diagram



Software Sub-units - Cont

Given a SWU S and its decomposition into sub-units $s_1 ... s_n$ the following lemmas hold:

$$(1) \quad \text{capabilities}(S) = \bigcup_{i=1}^{n} \text{capabilities}(s_i) \setminus \bigcup_{i=1}^{n} \text{hidden} - \text{capabilities}(s_i)$$

$$(2) \quad \text{interface}(S) = \bigcup_{i=1}^{n} \text{interface}(s_i) \setminus \bigcup_{i=1}^{n} \text{hidden} - \text{interface}(s_i)$$

$$(3) \quad \text{requirements}(S) = \bigcup_{i=1}^{n} \text{requirements}(s_i) \setminus \bigcup_{i=1}^{n} \text{capabilities}(s_i)$$

$$(4) \quad \text{side} - \text{effects}(S) = \left(\bigcup_{i=1}^{n} \text{side-effects}(s_i) \right) \cap \text{interface}(S)$$

ffortid 4.0 Requirements - Cont

From comparison of old and new manual pages we created a list of 3 required enhancements:

- Change the command-line options and add the capability to automatically stretch letters and/or connections according to theses options and the new information in the width tables.
- Add the capability to manually stretch letters.
- Add the capability to control automatic stretching of words with manually stretched letters and/or connections by two new command-line options.

SWU Modifications

- **A SWU modification can potentially affect its capabilities, interface or requirements.**
- **4 major types of modification:**

Type	Capabilities Modified	Interface Modified
I	no	no
II	no	yes
III	yes	no
IV	yes	yes

- **In type II* and IV* current interface is only added to and not modified.**
- **In type III* current capabilities are only added to and not modified.**