Problem: How to extend and change software after its initial design?
Obstacles:
- Excessive information distribution
- Chain of data transforming components
- Integration of many functions in one
- Dependency loop (nothing works until everything works)
Answer: Design for change (extensibility and removability)

Main Paper (cont’d)
- How to design for change?
  - Requirement identification (minimal subset of family members)
  - Information hiding (modules and their secrets)
  - Virtual machine design
  - Uses relation hierarchy
- Example:
  - An address processing system (A set of programs to read in, store, and write out lists of addresses)
Main Paper (cont’d)

- What if the format of the input address is changed?
  1. Design another program from scratch
  2. Design for change (A hierarchy of virtual instructions)

```
int NAD(int frmt) { read in an address in a format specified by a parameter
int NADSEL(int frmt) { read in an address using one of a set of formats
int NADFO(int frmt, dev strg) { read in an address and store it
The address is assumed to be in a specified format
```

First Related Paper: Commonality Analysis

- Problem: How to identify family members?
- Answer: A systematic approach in the analysis phase
- Defining family members:
  - Sources of abstraction: Terminology, Commonalities
  - Variations of a family member
  - Variability: The scope of the variations of the entire family
- Commonality Analysis (CA) Elements:
  - Terminology, Commonalities, and variability
- Output: Documented family requirement information
- User: Designers

[David M. Weiss, 1997]

First Related Paper (cont’d)

- CA Process: Based on Regular Meetings
- Steps (sequential)
  - Preparing: the required sources for initial sessions
  - Planning: the goals and the scope of the analysis
  - Analyzing: characterization of family members
  - Quantification: defining the parameters of the variations
  - Reviewing: by an external reviewer
- CA is a part of FAST at Lucent Technology (since 1992)
- Motivated by Parnas’ idea of design-for-change

[David M. Weiss, 1997]
Second Related Paper:

A Model for Designing Adaptable Software Components

- Problem: How to design adaptable components?
  - Design for change (Parnas idea)
  - Efficiency (component’s behavior)
  - Application builders
  - Language independence (implementation)
  - Legacy codes
- Major challenges:
  - Providing environmental information for the components
  - Proposed approach: Using Arbitrator
  - Designing interfaces
  - Proposed approach: Active Interface

[George T. Heineman, 1997]

Second Related Paper (cont’d)

- Implementation Strategy
  - Framework: ADAPT specification language
- Example:
  - Adding new functions to a Spreadsheet
- Advantages of language-base approach:
  - From component designer’s view
  - From applications builder’s view
  - Efficiency of components
  - Re-engineering of legacy codes

[George T. Heineman, 1997]

Third Related Paper:

Software Component Technologies and Space Applications

- Problem: Traditional view of one-of-kind products
- Answer: Component technologies (domain-specific design-for-change)
- Challenges:
  - Encapsulation
  - Composition of components
  - Paradigm
  - Scability
  - Verification
- Key issue: Paradigm shift

[Don Batory, 1995]
Third Related Paper (cont’d)

- Implementation proposal:
  - Design maintenance
    - Domain modeling
  - Modifications required in other parts
  - Reflective computations
- Extending Parnas’ idea:
  - Generating efficient code automatically
- Motivation in space domain: productivity

[Don Batory, 1995]

Indirectly Related Paper 1: Refinement and Separation of Concerns

- Problem: Design-for-change
- Basic idea: Any changes should be reflected in all design aspects
- Solution: Change the definition of module
- Module:
  - Basic unit of abstraction
  - Encapsulates source code, documentation, formal properties, and performance model

[Don Batory, 2000]

Indirectly Related Paper 1: (cont’d)

- Refinement: Abstraction of a common feature
- Family of similar applications based on the refinement
- Design methodologies for architecturally changeable and extensible systems (e.g., GenVoca)
- Analyzing the effect of a change on all design aspects

[Don Batory, 2000]
Indirectly Related Paper 2:
Issues in the Design of an Extensible Operating System

- Problem: Overcoming obstacles in designing (dynamically) extensible operating systems
- Main observation: Requires extensible software infrastructure
  - Design-for-change or Parnas principle
- Domain-specific goals:
  - Incrementality
  - Correctness and integrity
  - Efficiency

[S. Savage and B. Bershad, 1994]

Indirectly Related Paper 2 (cont’d)

- Dynamic extensions and extensibility namespaces
  - Parnas’ VM
  - Parnas’ Uses structure
- OS domain-specific major concern: Correctness
  - Conflicting resources
  - Verification
  - Protection (by design)
  - Automating protection

[S. Savage and B. Bershad, 1994]

Conclusion

- The outcome of the main paper:
  - Design-for-change principle
  - Methodology
- The impact of the main paper
  - Analysis, Design, and Maintenance phases
  - Design-for-change affects reusability and adaptability