The Evolution from
Software Components to Domain Analysis

James M. Neighbors
System Analysis, Design and Assessment
Costa Mesa, California
USA
Goals

- Produce large, quality software systems.
- Build systems from reusable software components.
- Create systems that may be extended and maintained over a lifetime of many years.

Software Components

- McIlroy at NATO68
- craftsman vs. mass-production
- system size forces reusable components
Areas of Investigation and Experience

- software components
- program transformations
- system architecture
- large systems
- automatic programming and program generation
Software Part Libraries

- reuse without modification ("what" information)
  - classification problem
  - search problem

- reuse with modification ("how" information)
  - structural specification problem
  - flexibility problem

- overall library problem
  - many small parts for flexibility increases search
  - few large parts decreases search and decreases flexibility
Lessons from Software Component Libraries

- libraries are an immediate success
- libraries have been a success for years
- simple flat libraries do not scale up (sorts, lists, etc. are not the problem)
- domain-specific parts from domain analysts are powerful
Program Transformations

- motivation: store fewer source programs and specialize
- example transformation

  LHS: $X^*(\text{IF } P \text{ THEN } A \text{ ELSE } B)$ $\iff$
  RHS: $(\text{IF } P \text{ THEN } X^*A \text{ ELSE } X^*B)$
  EC: $X$ and $P$ are execution order independent

- matrix multiply example in paper
- refinement example

- formal algebra theory
Lessons from Program Transformations

- few equivalence preserving transformations
- optimization at appropriate level of abstraction
- idea of a domain to encapsulate level of abstraction
  - semantics independent of implementations
  - optimizations independent of implementations
System Architecture

- motivation: how to encapsulate system and domain information
- architecture is distinct from function
- stepwise refinement vs. levels of abstraction

- vertical partitioning vs. horizontal partitioning
- dynamic creates cells of encapsulation
Stepwise Refinement (top down development)

Levels of Abstraction (bottom up development)

real systems use both
Lessons from System Architecture

- exists and is separate from function
- big impact on performance and maintenance
- result from encapsulation mechanisms
- methods that assemble systems from components must also create architectures
Large Systems

- motivation: how do they get them to work?
- scale, nature and location
- research method
- interconnection results
- identification of subsystems using coupling and cohesion

-establishing control
Calls To Other Routines

75% of routines call 7 or less routines

50% of routines call 3 or less routines

% Routines (log scale)

Calls To Other Routines
Calls From Other Routines

- 60% of routines called by 1 or less routines
- 90% of routines called by 6 or less routines
to enclosing utilities

Subsystem
global
data

interface
routines
functional
decomposition
functional
agents
utility
routines

□ = routine, 20–150 in a subsystem
average about 35
ENTER ROUTINE (IN SINGLE QUOTES), %DOWN, %UP
• establishing control of a large system
  – staff level: 20K lines source per programmer
  – identify tightly coupled modules
  – form tightly coupled modules into subsystems
  – assign 10K-30K source lines in subsystems to programmer

• module interconnection languages (MILs)
  – PS programming language
  – PS resource description language
  – PL resource flow language
Lessons from Large Systems

- system architecture important for extension and maintenance
- issues change from small systems
- MILs are required and usually custom made
- subsystem concept must be used in assembling systems from components
Automatic Programming and Program Generation

- motivation: techniques and effects of very high levels of abstraction
- automatic programming: strong mechanism, general knowledge
- program generation: weak mechanism, problem domain specific knowledge
Lessons from Automatic Programming and Program Generation

- domain-specific languages an aid, notations not a problem
- domain-specific knowledge with weak mechanisms powerful but inflexible
- general knowledge with strong mechanisms weak but flexible
- power of assembly mechanism must be balanced against the ability to plan using the mechanism
Methodology Requirements

1. problem domain specific objects and operations
2. hierarchy of domains (modeling domains)
3. optimization in domain independent of refining implementations
4. burden of search for implementing components removed from user
5. simple optimization and refinement mechanisms
6. refinement mechanism must also provide good system architectures
7. refinement mechanism must cope with pre-refined large subsystems
Organizational Flow using the Methodology

- Literature
  - software technology developments
- Modeling Domain Analyst
  - modeling domains
- Domain Designer
  - interconnected domain structure
- Application Domain Analyst
  - application domains
- Users of Similar Systems
  - executable system
  - available domains
- User with a Specific System Need
  - needs for a specific system
- Systems Analyst
  - functional requirements
- Systems Designer
  - domain language system description and refinement info

Machine

Draco
How Draco Works
Parts of a Domain Description

1. parser and schema
2. printer
3. optimizations
4. components
   - one for each object and operation
   - multiple refinements (implementations) for each
5. generators
6. analyzers
q931 { InitialState = U00_Null;  [ Q.931 User Side FSM ]

U00_Null ::

recv(Resume,net) -> CallRefSelection,
    send(Resume,net), StartTimer(T318) >> U17_ResumeReq;
recv(SetUp,net)  -> CheckSetUpMsg {
    SetUpOk    -> send(SetUpInd,net) >> U06_CallPresent;
    SetUpManElementMissing -> send(ReleaseComp(96),net) >> =;
    SetUpManElementError  -> send(ReleaseComp(100),net) >> =;
};
recv(SetUp,user) -> CheckSetUpMsg {
    SetUpOk    -> CallRefSelection, send(SetUp,net),
        StartTimer(T303) >> U01_CallInitiated;
    SetUpManElementMissing -> send(ReleaseComp(96),net) >> =;
    SetUpManElementError  -> send(ReleaseComp(100),net) >> =;
};
recv(Status,net) -> CheckStatusCaField {
    CsZero   -> nullaction >> =;
    CsNotZero -> RelOption {
        RelOpt  -> send(Release(101),net), StartTimer(T308)
            >> U19_ReleaseReq;
        RelCompOpt -> send(ReleaseComp(101),net) >> =
    };
}
recv(Release,net)  -> send(ReleaseComp(0),net), RelCallRef >> =;
recv(ReleaseComp,net)  -> nullaction >> =;
timeout (default) |
recv(default,net) |
recv(UnrecognizedMsg,net) ->
    RelOption {
        RelOpt  -> send(Release(81),net),
            StartTimer(T308) >> U19_ReleaseReq;
        RelCompOpt -> send(ReleaseComp(81),net) >> =
    };
recv(StatusEnquiry,net)  -> send(Status(0),net) >> =;
recv(RestartReq, user)  -> restartUser: StopAllTimers,
    send(RestartInd, user), RelCallRef,
    send(RestartConf, user) >> U00_Null;
recv(DL_Rel_Ind,net)   -> nullaction >> =;
recv(DL_Est_Config,net) -> goto DL_EstConf_label;
timeout (T309)  -> t309tout: send(DataLinkFailureInd, user),
        RelCallRef >> U00_Null
Experience with Methodology

- works but has problems
- produces efficient programs making small (20K line) systems
- pre-refined major subsystems are important
- ability to refine major subsystems is important
- academic generalists
- industry specialists
- future work
Conclusions

- Domain Analysis a big success
  - process of defining problem domain
  - education of new people on problem domain
  - checking template for new systems
- lack of modeling domains a big problem
- academic projects must deal with modeling domains issue or face complexity failure
- industry projects must use modeling domains or risk becoming program generators
- joint academic and industry work a necessity
  - academics know modeling domains
  - industry knows problem domains
- problem of constructing software from reusable components has become the problem of constructing modeling domain hierarchy