Issues in Architecture Evolution

Issues In Architecture Evolution

Dewayne E Perry
Empirical Software Engineering Laboratory
Electrical and Computer Engineering
The University of Texas at Austin
perry @ ece.utexas.edu
General Outline

- **Design Intent in Evolution (Maintenance Design)**
  - Work with my student Paul Grisham
  - Supported by NSF CISE SRS Grant CCF-0820251
  - ICCBSS 06 Keynote
  - SharkADI 07 Workshop: Architecture & Design Intent

- **Controlling Dynamic Evolution (Architecture Configurations)**
  - Builds on Dynamic Reconfiguration work from 1996
  - Work on the Abstract Architectural Model with
    - Sutirtha Battacharya
    - Supported by NSF CISE SOD Grant IIS-0438967
  - Work on the Architectural Configurations
    - Robert Watson
    - Enos Jones
    - Hyrum Wright
    - Supported by NASA Grant NNX08AC48G
“The issue is not documentation, the issue is understanding.”

- Jim Highsmith
Software Architecture

✿ Architecture: {Elements, Form, Rationale} (Perry/Wolf, 1992)

✿ Definition:
An initial design artifact representing a set of principle design decisions about a system and its structure

✿ Descriptive: A partitioning of the requirements of a system
- The current, implemented architecture for a system

✿ Prescriptive: Guidance for implementation of the system
- The designed or intended architecture for a system
- Implicit and Explicit design qualities

✿ Modern Software Design

Javascript yields cognitive benefits
- Large-scale design becomes possible
- Complex systems can be generally understood quickly
Problem Structuring

Well-Structured Problems (WSP):
- Relationship between problem, solution methods, and criteria
  - Eg, coding a well-defined algorithm

Ill-Structured Problems (ISP):
- Not well-structured (i.e., no domain guidance on solution methods or evaluation)
  - Eg, deciding what to build (requirements selection)

Problem Structuring:
- The act of turning ISPs into WSPs
- Software Analysis and Design:
  - Select requirements to implement
  - Given a requirement, decompose into a set of goals
  - Transform goal into a detailed design
  - Treat design as a WSP, and abstract its complexity, and use to solve another goal
Rational Decision Making

- A decision is made based on criteria and rationale
- Consequential choice of an alternative
  - Possible actions and outcomes
  - Utility function assigns value to options
  - Probabilities of outcomes
- Assumptions behind Rational Decision Making
  - Set of possible options are known
  - Probabilities of outcomes are known
  - Optimality is desirable
  - Cost of decision process is not a concern or is less than the cost of a sub-optimal decision
- Useful for WSPs

Naturalistic Decision Making

✦ Situational decisions
   ✦ Made on partial knowledge + personal expertise
   ✦ Preserved until they are invalidated

✦ Characteristics of Naturalistic Decision Making
   ✦ Dynamic or volatile situations
   ✦ Incomplete knowledge and ill-defined tasks and goals
   ✦ Knowledgeable and experienced decision makers
   ✦ Situational assessment over consequential choice
   ✦ Alternatives not considered until rejection
   ✦ Satisficing solutions

✦ Useful for ISPs
Problem Structuring and Decision Making

Software design is a combination of:
- Well-structured and Ill-structured problems
- Opportunistic and Prescriptive structuring methods
- Rational and Naturalistic decision making

Structuring Methods:
- Personal Experience
- Opinion, Ideas
- Domain Knowledge
- Group Interactions
- External Influences
- Existing Models of the Problem
- Existing Processes
- Preferred Evaluation Criteria
Early Work on Software Design Rationale

General Design Rationale Theory (1950s - 1990s)
- Goal: Produce better initial designs through explicit arbitration of known design alternatives
- Comes out of philosophical and legal theory domains

Potts and Bruns
- Structured design and analysis method on simple problem
- Separate process documentation from design results
- Reconstruct rationale from natural language descriptions

Parnas and Clements
- Software design not rational process
- Attempt to rationalize process for resulting design documentation
- Goal: Design process improvement

Early work on tools emphasized:
- Hypertextual navigation
- Visualization

Results of early studies
- Rationale modeling tended to impede creative process (Shum, 1994)
- Management and economic factors prevent adoption (Gruden, 1995)
A Survey on the Use of DR

Participatory Survey (Tang, 2006)
- 81 respondents with a minimum of 3 years experience
- Recorded respondents opinions about the use of DR

Results:
- Subjects responded favorably about the need for DR
- Less optimistic about adoption due to:
  - Poor tooling
  - Lack of management support
  - Positive justifications over negative justifications

Shortcomings:
- Survey cannot differentiate between DR and other forms of design knowledge documentation:
  - e.g., 80% of respondents said DR necessary to understand design
  - High favorability may be due to need for ANY form of docs
- Survey respondents tend to respond with positive affect in the abstract; considerably less when actual work is involved
Non-Software Study on Use of DR

Protocol Analysis study (Karsenty, 1996)
- Engineers (3) and technicians (3) in aerospace
- Given blueprints (design results) and DR (issue-based)
- Observations of 6 design evaluation meetings

Research Questions:
1. Do designers need to know design rationale to understand?
2. Do designers use design rationale documentation?
3. Does DR documentation capture relevant knowledge?

Results:
1. Of 138 design queries, 56% were DR questions
   - Also {Blueprint, Structure, Behavior, History, Documentation}
2. Opportunistic vs. Extensive use of DR
   - Drifting between blueprints and DR
3. Less than 50% of questions answered by DR documentation
   - Conclusion: DR useful, but not sufficient
Impact of Change with DR

**Controlled Study** (Bratthall, 2000)

- **2 Groups**
  - 7 senior industrial designers
  - 10 graduate students and software engineering faculty
- **Assigned 4 changes to 3 unfamiliar systems**
  - Access to DR information randomized
  - Change proposals compared to “key” solution from expert
- **Post-study interviews conducted for qualitative issues**

**Hypotheses:**
1. Changes take less time with DR
2. More “correct” changes with DR
3. Fewer “incorrect” or “superfluous” changes with DR

**Results:**
- Changes to simpler system benefited from DR, inconclusive on more complex systems
- Subjects responded favorably to available DR (affect)
Prototype Design Intent Study  (Grisham 2007)

- Graduate students asked to perform analysis of mature, complex open-source system for maintenance
  - Working in teams, identify sources of design intent from unstructured documentation,
  - Perform risk analysis on requirements changes, and
  - Identify affected architectural components based on requirements structuring

- Results:
  - Problem domain suitably complex for study
  - Difficult to locate information and terminate searches

- Shortcomings in study:
  - Students demonstrated naïve exploration methods
  - Self-reported results
  - No evaluation method for comparing results

- Improvements:
  - Controlled observation and protocol analysis methods
  - Simplify design project from existing domain to complete in < 4 hours
  - Isolate certain types of knowledge for comparison
Another Architecture Maintenance Study

“Add new functions to an existing product
  ➤ CLC-4-TTS Suite (web page accessibility)
  ➤ Evaluate decision modeling and decision support
    ➢ CBSP (design decision support)
    ➢ Archium (integrate decision modeling to arch.)

➢ Similar solutions generated from both techniques
  ➤ Learning effect?

➢ Approaches are complementary rather than competing
  ➤ Structured process of generating architecture from requirements with CBSP
  ➤ Capture of alternatives and rationale with Archium

➢ Can this study be replicated?
  ➤ Original developer participated in maintenance study
  ➤ Problem domain highly specialized

Current Design Knowledge Systems Research

- **PAKME (M. Ali Babar)**
  - Web-based architecture knowledge management tool
  - Design options represented as contextualized cases
  - Patterns, best-practices treated as knowledge

- **ADDSS (R. Capilla)**
  - 4+1+1 view of architecture
    - {logical, physical, process, development, use-case}
    - Decision view
  - Supports traceability

- **Archium (A. Janson)**
  - Component Language integrated with Java
  - Design decisions as first-class entities
  - Compiler and run-time support

- **Other**
  - Decision & traceability modeling for Enterprise Architect (A. Tang)
  - Other tool integration: e.g., Word+Visio, etc.
PAKME Design Knowledge Model

Partial Data Model

ADDSS Design Knowledge Model

Problems with Current DI Approaches

- Most tools in prototype or proof-of-concept phase
- More focus on data capture than data application
  - “What can we model”
  - Not, “What do we wish we had modeled?”
  - Data relevance not evaluated
    - What data is important to maintainers?
    - Strategies for selecting satisficing and underconstrained models
    - Significance of mitigating actions* and negative scenarios?

- Not enough emphasis on evolutionary (maintenance) design
  - Focus on initial design support
  - How to evolve DI models along with design?
  - Locating relevant data still an open issue
    - Free-text searches
    - No heuristics for model validation

How Do Software Designers Think?

- **Protocol analysis study** (Guindon, 1987, 1990)
  - Study of 3 “experienced” designers on small-scale problem

- **Opportunistic Decision Making**
  - Decisions made with partial knowledge influence later decisions as fact
  - Emergent knowledge and partial solutions
    - Discovery of partial WSPs from domain knowledge
  - Emergent requirements need attention
    - Immediate Structuring ISP into WSP
  - Drifting
    - Explore dependencies and assumptions
  - Scenario exploration
    - Make ill-structured requirements concrete
    - Verify partial solutions
    - Confirm inferred requirements

- Early design activities are opportunistic, rather than prescriptive
  - Emergent Hypothesis:
    - Rational decision modeling is probably counter-productive
How Do Software Architects Think

- Interviews and Ethnography (Zannier, 2007)
  - 25 interviews with software designers
  - Perspective modeling:
    - Developer: championed a design change as a critical incident
    - Mentor: championed designed change as abstract concept
  - Content analysis of transcripts to identify coded themes

- Largely support Guindon’s finding
  - Opportunistic decision-making, Satisficing solutions, Singular evaluation, and Drifting

- Modern software design is structuring
  - Ill-structured problems decomposed into smaller well-structured problems that can constrain the final solution

- Designers did not use knowledge from books and journals
  - Where do designers gain their knowledge as novices? (Can DR be used as a mentoring tool?)
How Do Software Maintainers Think

Protocol Analysis study (Siletto, 2006)
- Two studies on program understanding:
  - 9 students working in pairs on unfamiliar code
  - 16 experienced programmers working on familiar code
- What does a programmer need to know about a code base when performing a change task to a software system?
- How does a programmer go about finding that information?

Produced 44 distinct queries in 4 groups
1. Finding initial focus points
2. Building on those points
3. Understanding a subgraph
4. Questions over those subgraphs
   - Seems to include “rationale” and “intent” questions (“Why”)

No analysis of:
- Relation between question types and experience
- Effectiveness of strategies
- How an eventual design would emerge from these queries
What is Maintenance Design?

- Good design methods only delay erosion (Bosch 2002)
  - "Architecturally implied rules are not clear to the software engineers who work with it."

- Maintenance as much as 80-90% of total cost
  - Process cost → Claim: caused by early requirements errors
  - Cost is actually in evolution of the system for improvements and new features and uses (in addition to faults)

- 80% of time spent in design re-discovery (Davison 2002)
  - Program understanding cost

- Effect on system evolution of erosion
  - Difficult to understand system → impact of change analysis
  - Violated constraints → defects
  - Structural breakdown → small changes, multiple locations

- Goal: Perform architecture transformations, while preserving implicit design rules
Design is for Humans

CLAIM: There are some technical benefits of certain design strategies, but comprehensibility is the primary objective of modern design and analysis.

- Code elements are given “intentional” names
- Organization makes “clear” the intent of a set of instructions
- Modularity (coupling, cohesion) ➔ abstract complexity within an interface

CLAIM: Flexibility, elegance, testability, adaptability, etc. are all aspects of comprehensibility

CLAIM: We have spent considerably less time studying how people use designs than on producing the designs themselves
The Use of Mitigating Design Knowledge

- Self-reported case study (Wu, 2006)
  - Safety-critical industrial control system
  - Use a framework of mitigating actions to document design
  - Subjects in study were able to model requirements

- Safety requirements are often negatively normative
  - e.g., Must NOT

- Mitigating actions:
  - Define undesirable or dangerous outcome explicitly
  - Define decision in terms of avoiding that outcome

- Open research question:
  - Given that designers are opportunistic...
  - ...and design is largely a problem structuring activity...
  - ...are warning signs to eliminate solutions more useful to maintainers?
Design Artifacts

- A system is the runtime behavior of:
  - An execution environment (computer system), and
  - A program (compiled, linked, and loaded code)

- Many different levels of abstraction and viewpoints:
  - Machine code
  - Source code
  - Object model
  - Software architecture
  - Project plan and budget
  - System architecture
  - User manual
  - Requirements

- Source code must be kept up-to-date
  - Architecture and requirements may drift, e.g.
Design Artifacts Use Many Forms...

```java
public HttpServer (int port, int poolSize) throws IOException {
    _serverSocket = new ServerSocket (port);
    _httpResponse = "HTTP/1.0 200 MyServer 
Cache-Control: 
no-cache
Pragma: no-cache 
";
    try {
        _pool = new Pool (poolSize, HttpServerWorker.class);
    } catch (Exception e) {
        e.printStackTrace();
        throw new InternalError (e.getMessage());
    }
    _clients = new Vector();
    _acceptTID = new Thread(this);
    _acceptTID.start();
}
```
...But May Lead to Conflicting Views

As Marketing requested it
As Sales ordered it
As Engineering designed it

As Data Processing programmed it
As Services installed it
What the customer ordered
Design Artifacts and Process

- How are designs used?
- Who uses them?
- When are they useful?

- **Upstream**: Early in the design or development process
- **Downstream**: Late in the process
  - Use may be “accidental”

- **Upstream** cost-to-benefit ratio important
- Correctness and structure important for downstream use

No single artifact will meet every need

<table>
<thead>
<tr>
<th>Upstream</th>
<th>Downstream</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements</td>
<td>Corrective Maintenance</td>
</tr>
<tr>
<td>Design</td>
<td>Design Refactoring</td>
</tr>
<tr>
<td>Coding</td>
<td>Evolutionary Maintenance</td>
</tr>
<tr>
<td>Test</td>
<td>Installation</td>
</tr>
<tr>
<td>Integration</td>
<td>Test</td>
</tr>
<tr>
<td>Installation</td>
<td>Integration</td>
</tr>
<tr>
<td>Corrective Maintenance</td>
<td>Design</td>
</tr>
<tr>
<td>Design Refactoring</td>
<td>Evolutionary Maintenance</td>
</tr>
<tr>
<td>Evolutionary Maintenance</td>
<td>Corrective Maintenance</td>
</tr>
</tbody>
</table>
Program Understanding Problem

A person who “understands” can explain a program’s:

- Structure (design)
- Behavior (runtime)
- Effects on its operational context (the external world)
- Relationship to domain (appropriateness)

without using the language of the program itself.

For maintenance design understanding, we focus on:

- Identifying implicit structural design rules
- Understanding impact of requirements change on design
- Identifying design transformations that minimize:
  - Cost of source code changes
  - Impact of drift and erosion

What techniques do you use to understand a program?
Design Knowledge

✦ Add Knowledge to descriptive designs:
  ➡ Can provide traceability between requirements and design

✦ Examples:
  ➡ Design Rationale:
    ➢ Explicitly identify issues, alternatives, and decision process
  ➡ Design Histories:
    ➢ Cross-referenced with deliberation transcripts
  ➡ Design Patterns:
    ➢ Reuse successful design strategies
  ➡ Design (Architecture) Styles:
    ➢ Explicit codification of important design aspects (design rules?)

✦ Stated Benefits of Design Knowledge:
  1. Upstream: Enhance quality of initial design by encouraging optimal selection of design alternatives
  2. Downstream: Enhance quality of maintenance design by making implicit design rules explicit
Types of Design Knowledge

Dimensions of Decisions

- Levels of Knowledge
  - {Tacit $\rightarrow$ Documented $\rightarrow$ Formalized}

- Maturity of Decision
  - {Idea $\rightarrow$ Tentative $\rightarrow$ Decided $\rightarrow$ Approved $\leftrightarrow$ Challenged $\rightarrow$ Rejected}

- Validity, Relevance over time
  - Decisions may be superseded or rendered invalid by changes
  - Traceability?

Types of Documented Decisions

- Implicit and Undocumented
- Explicit but Undocumented
- Explicit and Explicitly Undocumented
- Explicit and Documented

Ontology of Design Knowledge

- **Categories of Decisions**
  - Existence Decisions ("Ontocrises")
  - Non-Existence Decisions, or Bans ("Anticrises")
  - Property Decisions ("Diacrises")
  - Executive Decisions ("Pericrises")

- **Relationships between Decisions**
  - Constrains
  - Excludes (Forbids)
  - Enables
  - Subsumes
  - Conflicts with
  - Overrides (Replaces)
  - Comprises (Is Made of, Decomposes into)
  - Is an Alternative to
  - Is Bound to (Strong)
  - Is Related to (Weak)
  - Relationship Closures
  - Etc
Design Knowledge Research Focus

Key Theory of Design Knowledge:
- Knowledge about design:
  - Supports the maintenance design process
  - Prevents design erosion
  - Leads to cheaper maintenance cost

Key Technical Issues:
- Integrate design knowledge into prescriptive architectures
  - Notations, Tools
- Validate descriptive archs. against prescriptive archs.
- Evolve prescriptive architectures
  - Processes, Structures, Tools

Key Research Issues:
- How to empirically measure the cost of design knowledge?
- How to empirically validate benefits of design knowledge?

Scope: Limit research focus to downstream usage
Evaluating Design Knowledge Systems

 cioè Software Engineering → Technology + Management

 cioè Experimentation provides statistical constraints

« Correlating factors are artificial
« Human variance influences outcome significantly

 cioè Adds techniques based on behavioral sciences

« Cognitive and Behavioral Psychology
« Sociology
« Management Science
« Design Science
« HCI and Usability

 CLAIM:
 We have spent more time studying designs and design artifacts than on how people use those designs
Sciences of the Artificial

- Relationship between world and artifact
  - Needs in the world stimulate innovation
  - Innovation changes the world and its needs
  - Expressible theory tends to be normative, not descriptive
Empirical Design

Artificial Sciences: inadequate experimental basis

- Theories have to be testable
- Testing is done in a combination of intellectual, behavioral, technological and physical worlds
  - Physical world provides hard constraints on theories
  - Technological world provides selectable constraints
  - Behavioral world provides probabilistic constraints
  - Intellectual context provides malleable constraints

C – Context (world)
I – Instrument
T – Theory
H – Hypothesis
R – Regimen (treatment)
Empirical Approaches

1. **Descriptive**
   - **Goal:** Build initial theories and models
   - Necessary first step – Many strategies for theory-building
   - Observe current practice and build models

2. **Relational**
   - **Goal:** Test theories and models
   - Requires at least 2 observations for comparison
   - Leads to quantitative models of correlation between variables
   - Not sufficient to prove causality
   - Compare use of design knowledge system against control
   - Compare use of different design knowledge systems

3. **Experimental**
   - **Goal:** Identify causality between independent and dependent
   - Use true experiments to isolate confounding variables
   - Difficult to create true experiments. More on that later!
Quasi-Experimental Methods

**Initial Studies**

*General Anecdotal*
- Typically self-reported
- Post-priori analysis of observable phenomenon
- Inference of treatment and independent variables
- Cheapest, most common, and least convincing

*Exploratory Study*
- Similar to Grounded Theory for discovery of emergent theory
- Typically used to prototype large-scale industrial studies
- Allows validation of experimental design

**Non-Observational Studies**

*Artifact Analysis*
- Inspect system and process artifacts for measurable phenomenon: source code, design records, etc.
- Multiple studies may be executed on single data set
- Emergent theory discovery through mining SW repositories
- Non-invasive data capture
Quasi-Experimental Methods

Observational Studies

- Ethnography
  - Integration of observer in industrial environment
  - Possibly participatory
- Self-Reported Case Study
  - Most common case study form in software engineering
  - Subjects are technique experts (typically inventors)
  - Subject affect is high
- Protocol Analysis
  - A controlled study where subjects solve exemplary problems
  - Subject behaviors are recorded and analyzed to infer cognition
  - Subject may be asked to verbalize thought process

Interviews, etc.

- Structured survey
- Unstructured Interview
  - Useful for emergent theories
- (Semi-) Structured Interview
  - Facilitates comparative analysis with multiple subjects
Simplified Model of Artifact Use

- **Cognition**: How you think
- **Behavior**: What you do
- **Affect**: How you feel
What Can We Observe?

- Only external Behaviors and the Artifact itself can be observed
- Affect and Cognitive phenomena must be inferred

Artifact

Cognition

Behavior

Affect

Requires honest participation

Cannot observe directly

© 2005-present, Dewayne E Perry
Analyzing Artifacts

- Inspect process artifacts for measurable phenomenon
  - Code, bug reports, estimates, requirements, notes, e-mail
  - Multiple studies may be executed on single data set
  - Emergent theory discovery through analysis
    - e.g. data-mining, mining software repositories
  - Non-invasive data capture

- Perhaps most common form of Empirical SE research

- Artifacts obey “technical” law, not “natural” law
  - Technical evaluation (compilation, model-checking, etc.)
  - Quantitative measurements possible
  - But only qualitative comparisons between different forms

- An artifact may be evaluated for purpose
  - An artifact must meet some need in its context
  - As the artifact changes the world →
    - No longer needed
    - Needs change
    - New needs arise
Examples of Cognitive Dimensions

Goal: Rate a system against several “dimensions”
- Record interactions with system while performing tasks
- Subject reporting on experience with system

Dimensions:
- Hidden / Explicit Dependencies
  - Are there important details hidden from view?
- Viscosity / Fluidity
  - How difficult is it to make changes?
  - e.g., Programmer’s Apprentice, Gandalf
- Premature Commitment
  - Do you have to make a decision before you’re ready?
- Consistency; low-level errors; etc.
  - Can the notation be checked for internal correctness?
  - Are common errors avoidable in the notation? (e.g. off by 1 error)

Qualitatively rate notation or compare notations
- Some quantitative measures can be made
- “Dimensions” not independent
Observing and Measuring Behavior

- Observe → Record → Analyze → Predict → Observe

- Types of studies:
  - Direct / Indirect Observation
  - Structured / Unstructured Observation
  - Participant Observation & Ethnography

- Grounded Theory:
  - Systematic research methodology (several approaches)
  - Qualitative data → Theory
  - Record everything, determine relevance to theory later

- Confounding factors:
  - Interaction effects (What if observer is “different”?)
  - Observer effect (People behave differently when watched)
  - Experimenter bias (I see what I want to see)
  - Secondary variance (Things that are out of my control)
  - Interpreter effects (Events don't fit model)
Measuring Affect

- Measuring people’s real feelings is difficult
  - Qualitative only
  - Scale varies from subject to subject
- Not clear how affect changes behavior

Methods of observation
- Survey (Likert Scale: Rate from 1 to 5)
- Structured / unstructured interview
- Focus group

Confounding factors:
- Observer effects (subject answers to please observer)
- Interpreter effects (observer hears what he wants to hear)
- Peer pressure (subject wants to “fit in”, “be good”)

What is relevance of affect to design comprehension?
- Maybe user chooses to use one artifact type over another, more useful type to answer question
Describing Cognition

- What is cognition?
  - Nobody knows

- People think, then act
  - Even though some people don’t seem to think at all!

- We infer “thought process”
  - Observe external actions
  - Apply abductive reasoning

- Often used to describe “behavioral” workflow for intellectual tasks
  - e.g., User Interface design

- Difficult to falsify hypotheses about cognition
  - Often there are two ways to get to the same outcome
Methods of Cognitive Study

● Interviews
  - Structured or Unstructured
  - Usually conducted after task complete
  - May describe idealized cognitive process
  - Terminology differences → difficult to merge results

● Talk-Aloud Protocol Analysis
  - Subject performs a task and describes thought process
  - Contrived situation to control confounding variables
  - Act of talking may change task flow

● Ethnography and Participant Observation
  - By participating in the task, the observer learns how to think like the subject

● Structured Quizzes
  - Ask comprehension and analysis questions
  - Log user interaction (queries) with documentation
  - Use “expert” to evaluate correctness of answers
Designing a Maintenance Design Study

Base Hypothesis:

Design Knowledge $\rightarrow$ less Design Erosion

Specialization of theory: artifact meets its designed purpose

Must measure comparative improvement in design erosion

Artifact $\leftrightarrow$ Cognition relationship

Design Strategy:

Design knowledge should answer questions asked during task

What are those questions?

Create model of maintenance task

Identify difficulties in task model

Structure artifacts to meet specific difficulties

Control:

Must measure phenomena in untreated cases

But we don’t know what this means in all cases
Effectiveness of Design Knowledge

Things to measure:

- **Errors**
  - Correctness determined by an “expert” or oracle
- **Cognitive breakdowns**
  - Task progress momentarily stopped
  - “Where do I go from here?”
- **Time to complete**
  - Also measure subtasks and drifting if appropriate
- **Confidence**
  - Affect; qualitative
- **Task repetition**
  - Are subjects spending time going back for information they’ve already requested
- **Task deviation**
  - Do subjects discover unexpected ways to complete task
    - World → Artifact → World
Confounding Factors: Learning Effect

**Internal Validity:**
- Subject cannot solve same problem twice
  - Subject becomes familiar with problem domain
  - Must change details of problem in meaningful way
- Subject becomes familiar with design knowledge structures
  - Efficiency and confidence will increase with use
  - Design task process may change after exposure to problem

**Design Challenges:**
- Isolate errors from DK notation inexperience from “low-level error” type errors
- Incorporate training into study overhead
Confounding Factors: Complexity

External Validity:
- Architectures evaluated by qualitative means
- Contrived examples may be too simplistic
  - Results do not generalize, or
  - Participant determines desired outcome
- Requires problem with multiple valid solutions
  - Mundane realism

Design Challenges:
- Use real-world problem design
- Must synthesize “realistic” design knowledge if not available
- Must create “control” data set by removing design knowledge
Goals of this Research

- **GOAL:**
  Develop repeatable empirical studies to study how Design Intent modeling is used in software maintenance

- **GOAL:**
  Understand how maintainers use design knowledge to evolve systems while preserving properties

- **IDEA:**
  Integrate design history and Design Intent modeling with EPM
  - Capture artifacts in project monitor
  - Associate with process communication (email, etc.)
  - Build a framework for studying reuse of design knowledge
  - Create a repository of test data for subsequent testing
    - Project Replayer?
    - Controlled, repeatable experimentation
  - Integrate multinational research
    - US, Japanese, European, Australian

- **ISSUE:**
  What design modeling tools do ECSA participants already use?
Open Research Questions

- How do software designers (aka Software Architects) work?
- Is there a difference between how initial architects and maintenance architects design?
  - More importantly, is there a difference between how initial system designers and maintainers use documentation?
- Is there a difference between software architects and other (perhaps “radical“?) designers?
  - Can we use design theory from other domains to build hypotheses?
  - Software engineering lacks predictive design models found in other complex systems engineering domains
  - Software design evaluation relies on personal judgment
- Who is doing systems maintenance?
  - Claim: Software architects are forward engineers (Avgeriou 2007)
  - Experienced architects are recruited off the project after design is complete
  - Maintainers:
    - May not have participated in the original design
    - May be novice designers
Research Roadmap

✦ Who is performing software maintenance
  ✦ Industry survey to measure:
    ➢ Experience
    ➢ Continuity of Design Team
    ➢ Project Size and Complexity
    ➢ Organizational Outsourcing

✦ How do maintenance designers work?
  ✦ Refinement of Prototype Study into short version
    ➢ Experienced vs. Novice designers
    ➢ Individual vs. Team Work
    ➢ Content Analysis to develop model

✦ Effectiveness of different types of Design Intent
  ✦ Initial ethnography to understand documentation in context
  ✦ Enhancement of prototype study
  ✦ Evaluation of proposed designs against quantitative criteria
Software Architecture

Architecture: {Elements, Form, Rationale} (Perry/Wolf, 1992)

Elements:
- Data, processing, and connecting elements
- Now conflated into components and connectors

Form
- Placement, properties, relations
- Constraints on placement, properties, relations
- Weighting to indicate relative importance

Where our Maintenance Design research focused on Rationale/Intent, here we focus on elements and form.

Context: complex simulations of manned exploration vehicles
Goal: an architectural approach to drive the design and development of those complex simulations
Data becomes much more prominent - conflation not helpful
Current State in Simulations

Simulation language and System

- Home-grown, but sufficient to task
- Governed by a very flat architecture/design description
  - For components
  - For interactions
  - For topology
  - For scheduling
- Automatically generates simulation system and schedules
- Provides execution and visualization environment

Goals of research

- Reverse engineer existing simulations to create architecture models
  - SDP - an analysis tool for reverse-engineering existing flat simulation descriptions to provide
    - Relationship descriptions
    - Visualizations of the concrete architecture of the simulation
- Create architecture model and support to create simulations via architecture descriptions
  - Archpad - a graphical architecture modeling system
    - Tailor to creating simulation architecture models
    - Basis for generating simulations
Abstract Architecture Model

- Based on Perry/Wolf model, originally created for the NSF SOD grant, Constraint-based Architecture Evaluation (started 2005)
- Developed further with Battacharya (2006) as part of his PhD thesis work
- Develop it further here in the simulation context and add specializations needed for this research
- Focus is on the critical abstractions rather than on creating a new language per se
Abstract Architecture Model

- **Model consists of three abstract constructs**
  - **Arch-element**
    - A component (data or processing) or a connector
  - **Arch-composition**
    - Represents the substructure of an arch-element
  - **Arch-region**
    - An arbitrary collection of arch elements
    - Arch-regions may overlap, contain or be contained in other arch-regions

- **Arch-element is the basic architecture component**
  - **Arch-element** =
    ```
    ( name,
    {service-specifications},
    {general-constraints},
    {dependency-specifications}
    )
    ```

- **General constraints apply to the arch-element as a whole**
Abstract Architecture Model

*Service specifications represent the basic provisions by the arch-element to the architecture*

\[
\text{Service-specification} = \left( \text{name}, \{\text{input-constraints}\}, \{\text{output-constraints}\}, \{\text{general-constraints}\}, \right)
\]

*Dependency specifications indicate what is required of other arch-element in support of this dependency*

\[
\text{Dependency-specification} = \left( \{\text{input-constraints}\}, \{\text{output-constraints}\}, \{\text{general-constraints}\}, \right)
\]
Abstract Architecture Model

An arch-composition is a set of elements together with mappings as to how they relate to each other.

Arch-composition

( name, 
  {arch-elements},
  {mappings}
)

Mappings accomplish several things:

- Map an sub-arch-element service-specification to the arch-element service specification
  - i.e., indicate which service specifications are used to satisfy the arch-elements interface
- Map internal satisfaction of dependency-specifications to their associated service-specifications
- Map unsatisfied service-specifications to the arch-element interface specification
- Map general-constraint satisfaction
- Map unsatisfied general-constraints to the arch-element interface
Abstract Architecture Model

General constraints
- Functional constraints
- Non-functional constraints
  - Performance
  - Fault-tolerance
  - Scheduling
  - Etc
- Architecture styles

Architecture regions share the same constraints
- Arch-region =
  (  
    {arch-elements},
    {general-constraints}
  )
AAM Issues

- Arch-elements conflate data, process and connecting elements
  - Data is critically important in the simulations
    - Describe the environment for physical objects
    - Describe physical the structure of physical objects
    - Structurally distinct from processing elements and connectors
  - Connectors need to be developed further
    - Typically thought of as point to point communication mechanisms
    - Probably will need multi-connection connectors
      - Many to one - e.g., multiple clients, one server
      - One to many - e.g., broadcasts
      - Many-to-many - e.g., cooperating components negotiating or reaching consensus
  - Beyond communication: coordination, mediation, arbitration
  - Will develop as needed

- Will need sequence/tree/graph of architecture configurations (motivation below)
Elements in the Simulation World

- Basic and composite elements
  - May be both depending on use in a particular architecture configuration
    - In one simulations may be treated as a basic component (e.g., the CEV)
    - In another it may be that we need to consider its constituent component (CEV as stage 1 rockets and astronaut capsule, e.g.)

- Basic architecture elements are physical objects
  - Such as the CEV or earth

- Basic elements are active or passive
  - E.g., the CEV is active, earth passive
  - Passive elements often contexts for active elements
  - Passive elements often sources of constraining influences on active objects (as the earth is on the CEV, e.g.)
Elements in the Simulation World

- **A product line style-like organization**
  - **Commonality**: architecture components used in a variety of simulations
  - **Variability**: architecture components specifically for certain simulations

- **Schedules are critical for simulations**
  - Envisioned as a general-constraint
  - Also for real-time systems
  - Schedules for various levels of the simulations
    - Micro-level schedules for individual components
    - Macro-level schedules coordinating multiple components
    - Over all schedules governing sequencing phases of a simulation

- **Motivation for the notions of configurations**
  - Specific physical events when the “world” changes
    - Eg, failures, transitioning from earth to space
    - Eg, de-coupling the rocket stage from the capsule
    - Eg, docking at the space station
  - May need to represent sequences, trees or graphs of events
Extensions to AAM

An architecture configuration (AC) is a special form of an architecture element that has an architecture composition associated (data bindings are explicitly part of the composition) and loci of control specifications (thread bindings)

\[
\text{Arch-configuration} = \\
( \text{name,} \\
\text{arch-element:arch-composition,} \\
\{\text{thread-bindings}\} \\
\{\text{schedules}\} \\
)
\]

Thread bindings indicate several things

- Loci of control
- Range of those control loci are the transitive closure of the dependency specifications
- Schedules are associated with those threads
Extensions to AAM

Graphs of architecture configurations represent a (projected) history of the simulation

- **Sequences**
  - represent sequences of events
- **Trees**
  - Represent sequences of events that include choices
- **Graphs**
  - Represent sequences of events with choices and merges

An architectural configuration graph (ACG) represents a complete simulation

```plaintext
Arch-configuration-graph =
  ( name,
    {arch-configurations}
    {thread-bindings}
    {schedules}
  )
```
Extensions to AAM

® The thread bindings of an ACG
   ◆ Tie individual AC threads together across the architectural configuration graph
      ➢ Some threads stop executing
      ➢ Some threads continue
      ➢ Some threads start up
   ◆ Defines the actual execution of threads where the AC threads bindings merely define the potential threads in a configuration

® Schedules in an ACG
   ◆ Define when the ACs begin and end

® Research problems
   ◆ The transitions in data bindings from one AC to the next
      ➢ Integrates dynamic reconfiguration work from 1996 (ARES)
   ◆ The transitions in execution from one AC to the next in the context of a changed architecture
Example ACGs

 Scenario: blasting off from earth and experiencing a fuel leak in the rocket boosters
   - AC1 - standard blast-off AC
   - AC2 - simulates the fuel leak,
     - data model changed to represent this problem
   ...  

 Scenario: blasting off and either docking with the space station or landing on the moon
   - AC1 - standard blast-off AC
   - AC2 - booster rocket detach
   - AC3 - leaving earth's atmosphere and entering deep space
   - AC4a - docking with space station
   - AC4b1 - entering the moons atmosphere
   - AC4b2 - landing on the moon
Summary

 цель Began with our abstract architecture model

- Usefulness for modeling architecture elements in simulations
- Schedules for individual architecture elements describable as constraints on the elements

 цель Initial extensions to model needed

- Differentiation of data, processing and connecting elements
- Further development of connecting elements beyond typical use

 цель To model complex simulations where physical changes take place, propose the ideas of architecture configurations and configuration graphs, integrating earlier work on dynamic reconfiguration

- Notions of locus of control, threads
- Higher levels of scheduling
- Binding and rebinding of data
- Binding of threads to actual execution threads