Architecture Intent and Evolution

Michael Gilfix & Colin Petersen

April 4th, 2006

Outline

- Introduction: design intent and evolution
- An analysis of design erosion
- Study of the impact of design rationale in predicting change impact
- A dependency model methodology for managing change
- Conclusion

Evolution and Complexity

- Complexity of software systems increase over the lifetime of the system as a result of:
  - Structural constraints imposed by implementation
  - Cascading (rippling) effects of changes
  - Changing requirements, both functional and non-functional
- Knowledge management becomes an issue for evolution:
  - Knowledge of technology by developers
  - Fluctuating knowledge of the system due to staff turn-over
  - Partial knowledge at design time and forgotten knowledge
  - Human limitations –i.e., the system becomes just too big
- Other factors:
  - Ambiguous descriptions of specification and design
  - Cost and time pressures

Design Rationale and Evolution

- Design Rationale captures the architect's original intent – i.e., *why the system is built the way it is*
- Hypothesis: design rationale can speed and improve the quality of the software evolution process
  - Many design decision are made everyday at all levels. Are they compliant with the common goal?
  - Independence of decision making
  - Shared wisdom – can prevent mistakes that may not be obvious to the developer who does not have direct knowledge of customer or domain requirements
  - Prevents the circular discussion: decisions made in the heat of an in-depth design discussion may lose sight of the big picture
  - More information is a good thing *(or is it?)*

Different Levels of Architecture

- Discussions of architecture seem to implicitly talk about different granularities of architecture.
- As system size increases (and thus complexity and cost of evolution), granularity becomes important

For our discussion:

- ***Macro-level architecture:***
  - Interactions between components
  - Typically follows a loosely coupled or highly modularized model
  - Abstractions or contracts become paramount
- ***Micro-level architecture:***
  - Tend to be goal or function driven
  - Inherit constraints of larger system (i.e., threading models, availability of information)
- ***Architectural aspects or views:***
  - Deployment architecture
  - Management/security architecture (pervasive throughout the system)

Different Kinds of Evolution

- Software engineering process must reconcile different kinds of evolution:
  - Changing requirements
  - Translating design to implementation
  - Changes resulting from test cycles and performance tuning
  - Staff/Organizational changes
  - Environmental limitations (frameworks, platforms, deployment environments, etc.)
- These can each affect the architecture at different levels
- Studies that we've seen tend to focus on changing requirements and code impact

Putting Design Intent and Evolution in Context

What We've Seen So Far:

- Tools for describing architecture and capturing design rationale
- How design rationale can supplement traditional design specifications to improve the software development process
- How requirements are translated into designs and the tools (styles) for describing designs

The next step:

- How the evolution of software systems can be managed using intent and rationale-based methods

On Empirical Studies of Designers…

"The three chief virtues of a programmer are: Laziness, Impatience and Hubris."

- Larry Wall

Design Erosion

Design erosion: problems and causes

Jilles van Gurp and Jan Bosch. "Design Erosion: Problems & Causes". Journal of Systems and Software, Vol. 61, No. 2, 2002, pp. 105-119.

What is Design Erosion?

- **Perry** and Wolf (1992)
  - ***Architectural erosion:*** "the result of 'violations of the architecture'"
  - ***Architectural drift:*** "the result of 'insensitivity to the architecture' (the architecturally implied rules are not clear to the software engineers who work with it)."

Design Erosion Causes?

1) ?
2) ?
3) ?
4) ?

Design Erosion Causes?

1) Traceability of design decisions
2) Increasing maintenance cost
3) Accumulation of design decisions
4) Iterative methods

Design Erosion Causes (1)

- **Traceability of design decisions**
  - Design decisions are difficult to track and reconstruct

Design Erosion Causes (2)

- **Increasing maintenance cost**
  - As the complexity of the system grows, the tasks become increasingly effort consuming
  - This complexity leads to sub-optimal design decisions

Design Erosion Causes (3)

- **Accumulation of design decisions**
  - Design decision dependencies
  - When circumstances change, the system may no longer be ‘optimal’

Design Erosion Causes (4)

- ***Iterative Methods***
  - Extreme programming, rapid prototyping, etc. incorporate new requirements in the development phase
  - A proper design has knowledge of the requirements in advance

Design Strategies

- ***Minimal effort***
- ***Optimal design***

What do these strategies mean in terms of cost ?

Are these strategies feasible?

Design Strategies

In general…

- ***Minimal effort***
  - Low cost upfront
  - High cost in the future
    - Design decisions limit future iterations
- ***Optimal design***
  - High cost upfront
  - Low cost in the future
    - Any conflicts with decisions in the previous version are resolved

The Meat of the Paper

ATM Simulator Example

Goal: Show that Design Erosion is inevitable because of the way software is developed*

Good methods only contribute by delaying the moment that a system needs to be retired

* Is this a result of processes or forever partial knowledge of the full set of requirements?

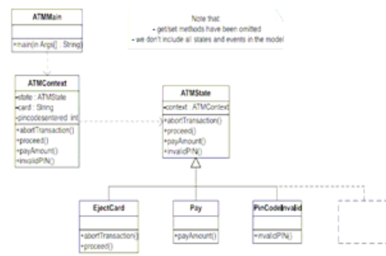
ATM Simulator Example *Version 1*

Requirements:

Core functionality. Provide a simple implementation of the ATM Simulator, based on the specification in the FSM.

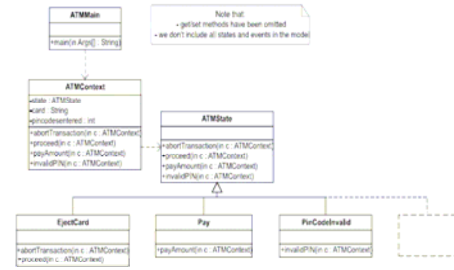
User Interface. Provide a primitive user interface to allow users to interact with the simulator.

ATM Simulator Example *Version 1*



Model the ATM with an FSM

ATM Simulator Example *Version 2*

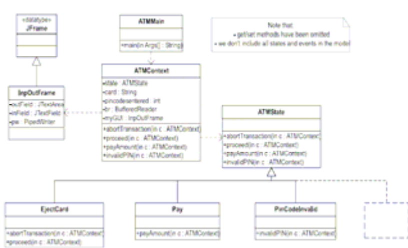


Memory Usage. The aim of the changes is to instantiate the state classes only once.

Performance. By reusing the state class instances, initialization time of the simulator is reduced for subsequent uses after the first initialization.

Flyweight Pattern

ATM Simulator Example *Version 3*

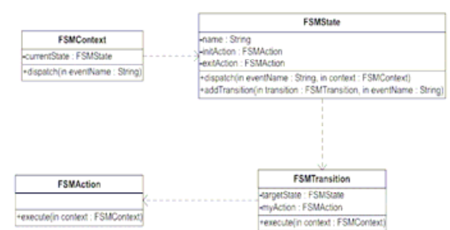


User interface. The user interface in the first two versions uses the command line for user input. However, when more than one simulator is used, a command line interface is no longer sufficient.

Parallelism. By making each simulator a thread, it is possible to run them in parallel.

Multi-user Version

ATM Simulator Example *Version 4*

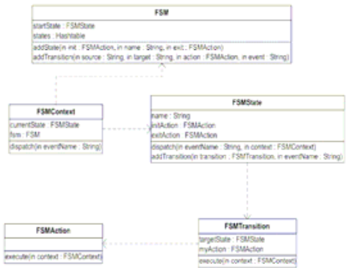


Configurability. Allow for run-time configuration, we want to be able to add new states and transitions at run-time.

Separation of concern. In the previous versions, we noticed that the details of the ATMSimulator get mixed with the typical behavior of finite state machines. Somehow, it should be possible to keep the two separated.

A Delegation-based Approach

ATM Simulator Example *Version 5*



Flexibility. The solution in version 4 puts the entire ATMFSM in a single class. A lot of this code is made static, which means that it cannot be changed at runtime and is difficult to maintain. In this version, we increase the flexibility by addressing this issue.

New FSM class included

Analysis

- ***Architectural Drift***
  - Initial version compact. Design, maintainability and flexibility less than ideal. End version has added structure and is more flexible but harder to understand.
- ***Vaporized design decisions***
  - Many early decisions are not in v5. Will be even more vaporized decisions for a larger system.
- ***Design erosion***
  - Optimal strategy still leads to design erosion.
- ***Accumulated design decisions***
  - Major restructuring of code.
- **Limitations of the OO paradigm**
  - Many of the solutions presented are found to be workarounds for the OO paradigm.
- ***Optimal vs. Minimal Strategy:***
  - Even the optimal strategy does not lead to an optimal design.

Some Questions.. ?

- What does erosion really mean in this context?
- Why is it bad (seems to have negative connotation)?
- How does erosion relate to iterative development/discovery in this context?
  - What if the original intent may have been incorrect?

Impact of Design Rationale

- ***Overall Hypothesis:*** Evolutionary changes will be faster and more correct if design rationale is available during change impact analysis
- *Change impact analysis* refers to modifying an existing system to meet new or modified requirements
- Authors verify hypothesis through a controlled experiment:
  - Subjects are asked to make changes to three systems (one was discarded in the course of the study)
  - Changes made by subjects are compared to a recognized expert, tabulated, and analyzed using statistical methods
  - *General comment:* Studies like this seem very hard to construct correctly!

Lars Bratthall, Enrico Johansson, and Björn Regnell. "Is a Design Rationale Vital when Predicting Change Impact? – A Controlled Experiment on Software Architecture Evolution" In *Proceedings of the International Conference on Product Focused Software Process Improvement*, 2000, pp. 126-139.

Organization of Experiment

- Experiments evaluate three hypotheses:
  - There is a difference in the length of time it takes to complete change tasks when design rationale is available ***(faster)***
  - There is a difference in how large a percentage of the required changes are correctly suggested when a design rationale is available ***(better)***
  - There is a difference in how many superfluous, incorrect, or unsuitable changes are suggested when design rationale is available ***(stronger)***
- Structure of the experiment:
  - A textual description of a system and its design rationale is given to the participants
  - Participants are introduced to a new system without prior knowledge of the system and asked to make changes
  - 17 participants are from two groups: senior industrial designers (7) and a group of Ph.D students and faculty members (10) in software engineering

Experimental Design

1. Before experiment, all participants are asked to document their knowledge and experience. Information used to randomize access to design rational (DR)
2. Provided overview of modeling language for the system (used graphical SDL language for finite state machines)
3. Participants were assigned four change requests in random order to three systems. author of DR descriptions had no knowledge of change tasks.
4. Time limits were provided for each participant
5. Participants were asked to fill out a form such as:

Indicate where you believe you will have to make changes for the current change task here.

What (Components in software)	An X indicates that you believe you would like to make a *change*	An X indicates that you would like to *add* a SDL-process or an SDL-block in this block
Cruise_Requirements Analysis		
Cruise_Requirements Analysis_Domain		

Case Study Analysis

System A – PBX Control System

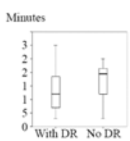


Fig. 6. System B, time used for change tasks

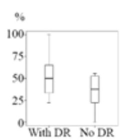


Fig. 7. System B, percentage of required change points

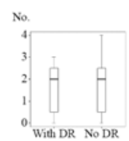


Fig. 8. System B, number of superfluous change points

System B – Car Cruise Control System

Fig. 3. System A, time used for change tasks

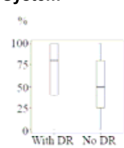


Fig. 4. System A, percentage of required change points

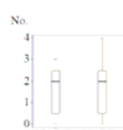


Fig. 5. System A, number of superfluous change points

Note: System A is considered significantly simpler than system B

Statistical Analysis

- *Use the Mann-Whitney non-parametric statistical significance test, with p <= 0.1*
- Some terms:
  - *Non-parametric:* the number and structure of the parameters to the experimental model are not known in advance and are determined from the data
  - *Statistical significance:* Unlikely to have occurred by chance. P-value indicates the maximum probability of rejecting a true null hypothesis (i.e., hypothesis has no effect). Lower values are more significant, but may introduce false negative (not detecting result when present)
- *To note:* Authors describe system A as being of considerably less complexity than system B.

	Sys = System A		Sys = System B	
	Mann-Whitney	Illustration	Mann-Whitney	Illustration
$H_{t,\,Sys,\,0}$	Reject	Figure 3	No reject	Figure 6
$H_{PercOK,\,Sys,\,0}$	Reject	Figure 4	No reject	Figure 7
$H_{NoExtra,\,Sys,\,0}$	No reject	Figure 5	No reject	Figure 8

Analysis of Interview Data

- "How much faster can you solve the change task (comfortably well) with access to DR?"

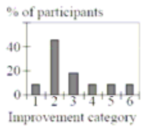


Fig. 9. System A, change in lead-time with access to DR

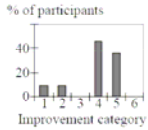


Fig. 10. System B, change in lead-time with access to DR

	Improvement categories
1	No opinion
2	0-19% faster with DR
3	20-39% faster with DR
4	40-59% faster with DR
5	60-79% faster with DR
6	80+ % faster with DR

- "To what degree do you think that a DR increases your correctness in change predictions?"

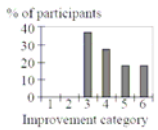


Fig. 11. System A, change in impact prediction correctness with access to DR

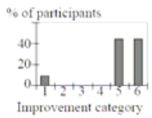


Fig. 12. System B, change in impact prediction correctness with access to DR

	Improvement categories
1	No opinion
2	It makes predictions worse
3	It does not affect correctness at all
4	I become marginally more correct
5	I become more correct
6	I become a lot more correct

Critique

- Although the authors claim success, the study seems inconclusive at close scrutiny:
  - System A was considered significantly simpler and was the only system where DR proved statistically significant, but as complexity increased, the benefit seemed to disappear. System C in study was discarded because it was too complex
  - The interview strongly indicated that people felt more comfortable with DR available
    - But, is this just a question of feeling comfortable with more information available?
    - Did it lure people into a false sense of security (incorrect decisions were still rejected)
  - All models were based on systems that could be described using finite state machines. This tends to reflect a certain class of system
  - Population was just too small
  - The randomization of access to DR is worrisome

Dependency Models

- Authors propose a modified Dependency Structure Matrix (DSM) approach for specifying software architectures
- Core ideas behind DSM:
  - Divide and conquer approach to describing software
  - Hierarchical decomposition of systems (large systems decomposed into sub-systems)
  - Decomposition should be done such that tightly coupled sub-systems are closer to each other, while loosely coupled sub-systems are kept further apart
- Addresses the problem of developers to creating undesirable couplings over time without an understanding of the larger system

Neeraj Sangal and Frank Waldman. "Dependency Models to Manage Software Architecture". CrossTalk - The Journal of Defense Software Engineering, November 2005.

Advantages of the DSM Approach

- Two key elements:
  - Precise hierarchical decomposition
  - Explicit control over allowed/disallowed dependencies between sub-systems
- Weakness of current box-and-arrow approach:
  - Diagrams become unwieldy as number of classes/entities increase
  - Separated from code –additional round-trip effort required to keep code and models synchronized
- Must be able to scale to allow visualization of thousands of modules

Basic Idea

- Matrix representation of modules and count of dependency linkage between modules
- A lower triangular matrix (no dependencies above the diagonal) is a structure without circular dependencies

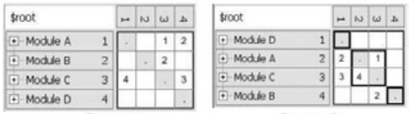
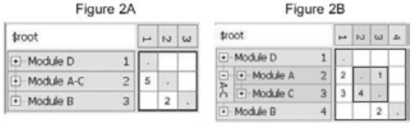


Figure 1A Figure 1B

- Can group together related modules to compress matrix view

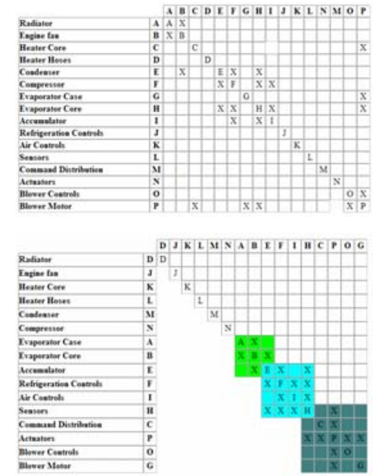
Figure 2A Figure 2B



- Can enforce architectural rules by disallowing relationships in matrix squares

Extracting Structure From Dependencies

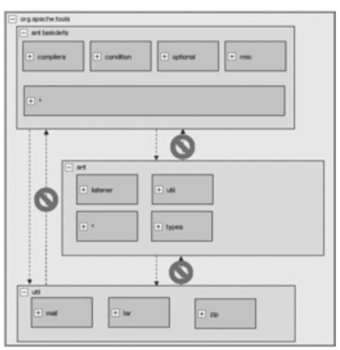
- Climate Control Example
- Suggested the following high level modular structure:
  - Front end air chunk (green)
  - Refrigerant chunk (blue)
  - Interior air chunk (gray)



Taken from:
http://www.dsmweb.org/Scrap_book/Climate.htm

Case Study – Apache Ant

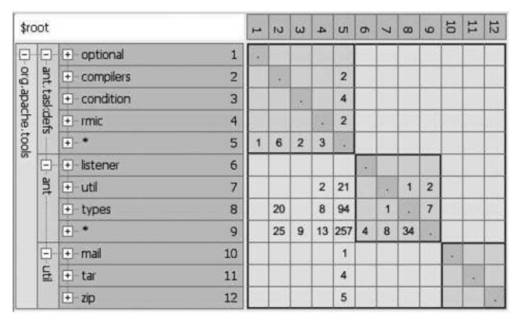
- Ant is an open-source, Java-based build utility:
  - The ant framework reads XML files and runs a series of tasks
  - Ant capabilities are extended through the creation of tasks, that are intended to be modular and separate from the framework



Ant Package Structure

DSM For Apache Ant 1.4.1

- Initially, Ant framework and set of tasks relatively small



DSM For Apache Ant 1.6.1

- Ant framework has grown in complexity and the number of tasks has increased.

$root				1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
org.apache.tools	ant.taskdefs	email	1	.					1		3										
		cvslib	2		.																
		compilers	3			.			2												
		condition	4				.		12				1			3	2				
		rmic	5					.	2												
		*	6		5	10	3	4	.	3							7				
	ant	loader	7							.											
		listener	8								.										
		input	9						3			.					4				
		types	10	3	2	19	1	7	197				.		20	8	12				
		helper	11						1				1	.			1				
		filters	12						3				21		.	1					
		util	13	1	1	3	1	3	72	1	2	1	20	4	11	.	17				
		*	14	11	11	26	23	15	368	1	5	3	98	24	11	21	.				
	util	mail	15	1							1							.			
		bzip2	16						4										.		
		tar	17						4											.	
		zip	18						8				2								.

Conclusion

- Software systems acquire complexity as they evolve
- Design erosion and architectural drift is inevitable
- Design rationale *may or may not* be a useful tool in steering the evolution process
- The ability to discover architectural patterns from the software, and enforce those patterns within the solution, can be a useful tool for managing evolution
  - The DSM approach provides one such solution