Architecture and Design

Guest Lecture for
COMP 180: Software Engineering
Tufts University
Fall 2006

John Klein

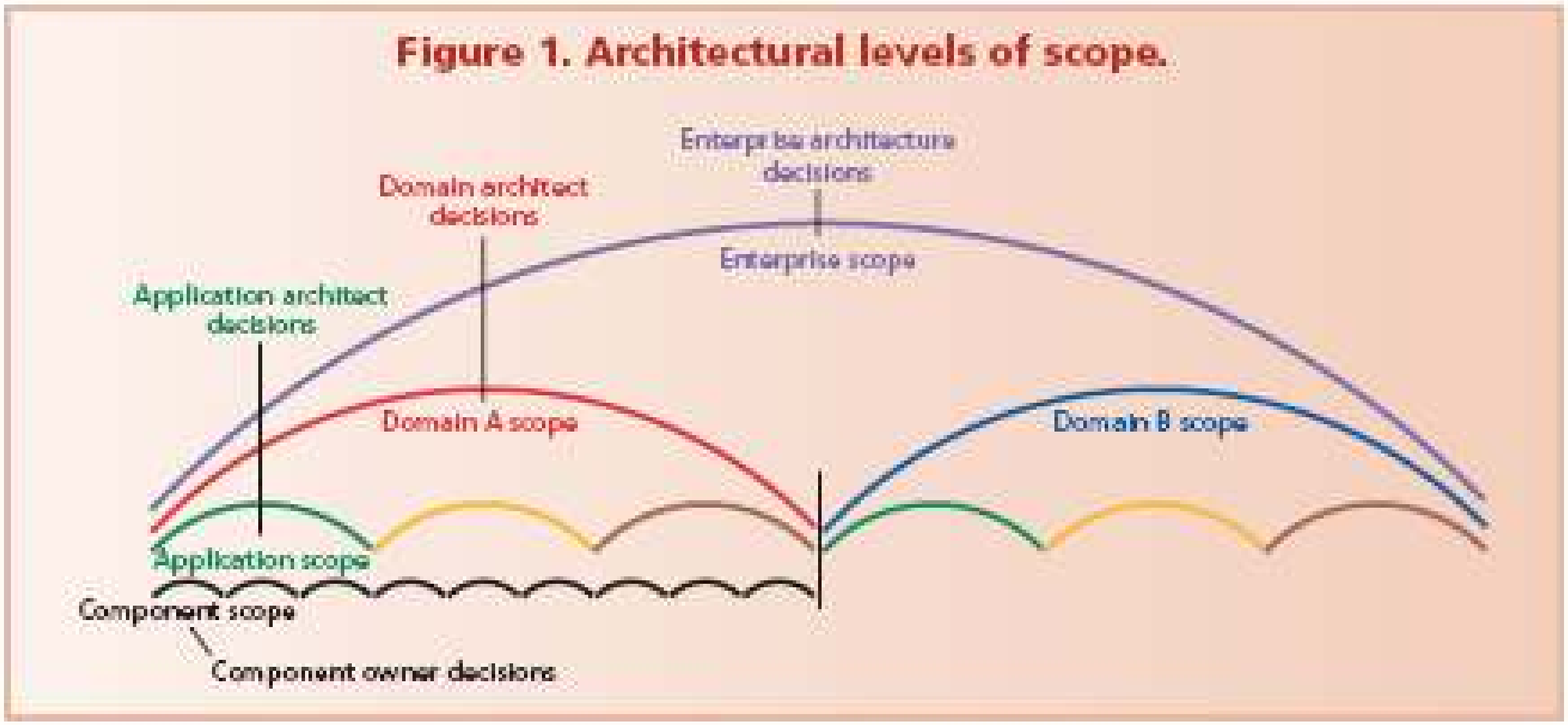
Architecture – Definitions

- http://www.sei.cmu.edu/architecture/definitions.html
- *The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.*
  - Bass, Clements, Kazman, Software Architecture in Practice (2nd edition), Addison-Wesley 2003.
- *The structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time.*
  - Garlan, Perry, "Introduction to the Special Issue on Software Architecture," *IEEE Transactions on Software Engineering*, April 1995.

Practitioner Definitions

- *"Software architecture is the set of decisions which, if made incorrectly, may cause your project to be cancelled."*
  - Eoin Woods
- *"Decomposition of the problem in a way that allows your development organization to efficiently solve it, considering constraints like organizational structure, team locations, individual skills, and existing assets."*
  - John Klein

Architecture/Design Decisions



from Malan,Bredemeyer, "Less is More with Minimalist Architecture", *IT Pro*, Sept/Oct 2002, p. 48.

Frequently Used Tools in the Software Designer's Toolbox

- Abstraction
- Separation of Concerns

- Patterns – Architecture, Design, Language & Technology
- Organizational Patterns (see Coplien and Harrison, *Organizational Patterns of Agile Software Development*. Prentice Hall, 2004.)
- Notations – UML, SDL, Traces, Formal Specification Languages (Z, CSP, …), Predicate Logic

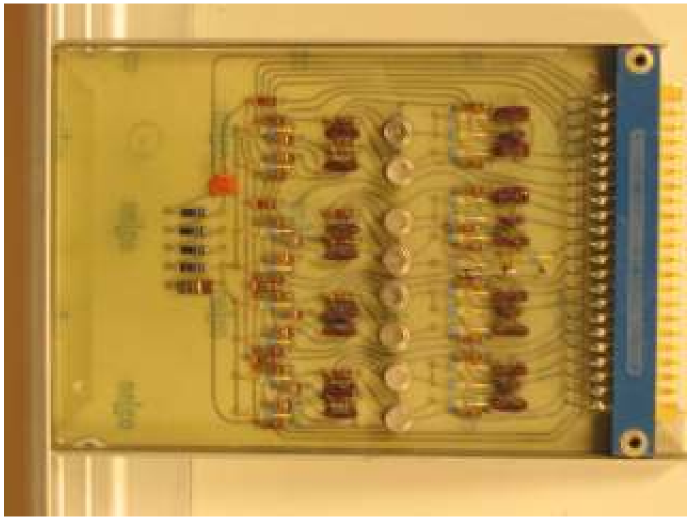
Abstraction

All Software is an Abstraction –
Underneath it all is electrons flowing through semiconductors

- Logic gates…

		MIN	TYP	MAX	UNIT
V_{IH}	High-level input voltage	2			V
I_{IH}	High-level input current (@ V_I = 2.4V)			40	μA
V_{IL}	Low-level input voltage			0.8	V
I_{IL}	Low-level input current (@ V_I = 0.4V)			-1.6	mA

- Early 4-bit counter (DEC PDP-6, circa 1965)



Abstractions

- Assembly Language

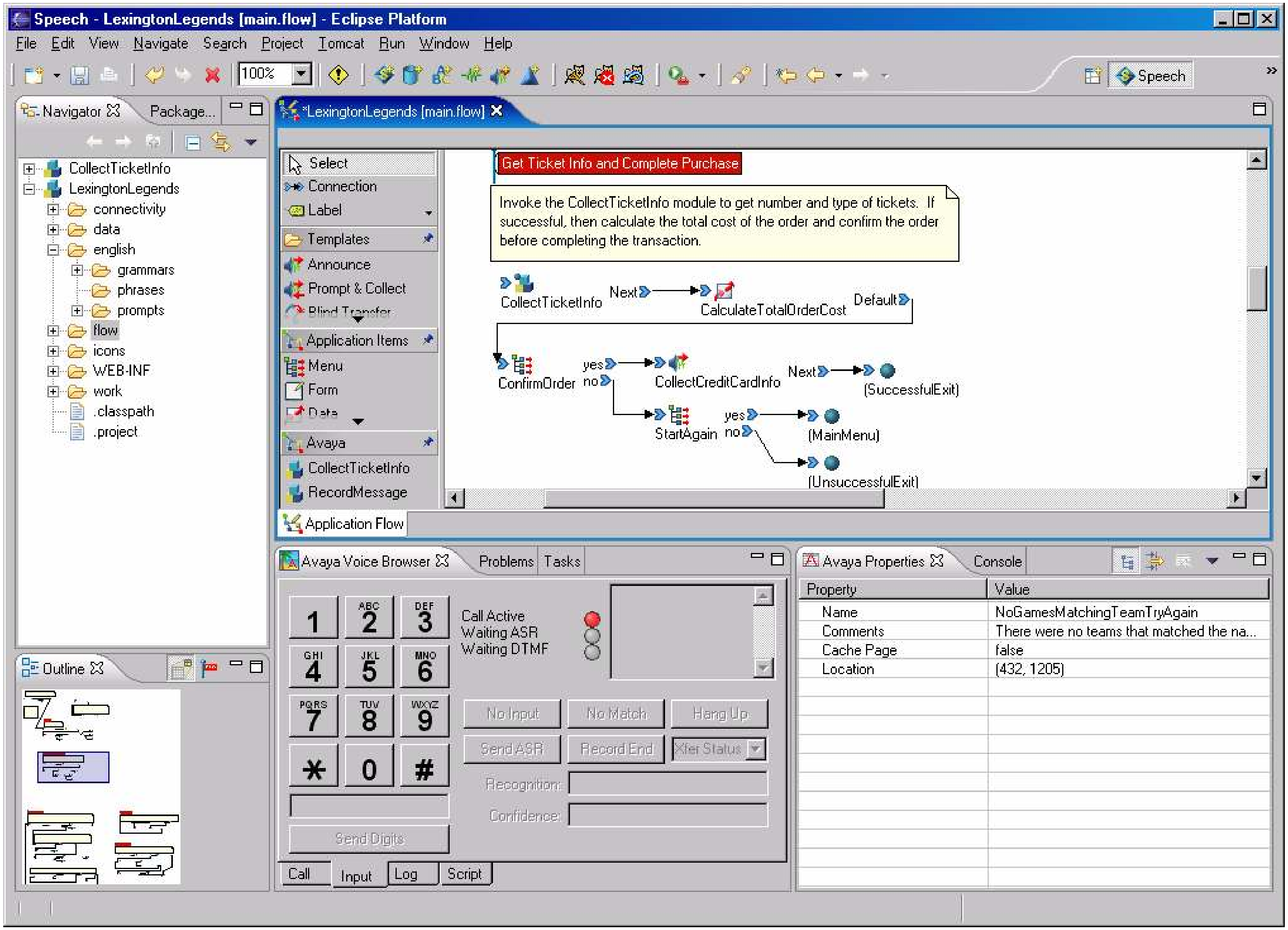
```
 ALIGN     4                                             ; 0
           PUBLIC   _main
_main      PROC NEAR
@B1@8:                  ; preds: B1.3

        mov       edx, DWORD PTR 12[ebp]                        ; 2
        mov       eax, DWORD PTR 8[ebp]                         ; 2
        cmp       eax, 2                                        ; 10
        mov       edx, DWORD PTR [edx]                          ; 8
        movsx     ecx, BYTE PTR [edx]                           ; 8
        je        @B1@1         ; PROB 5%                       ; 10
```

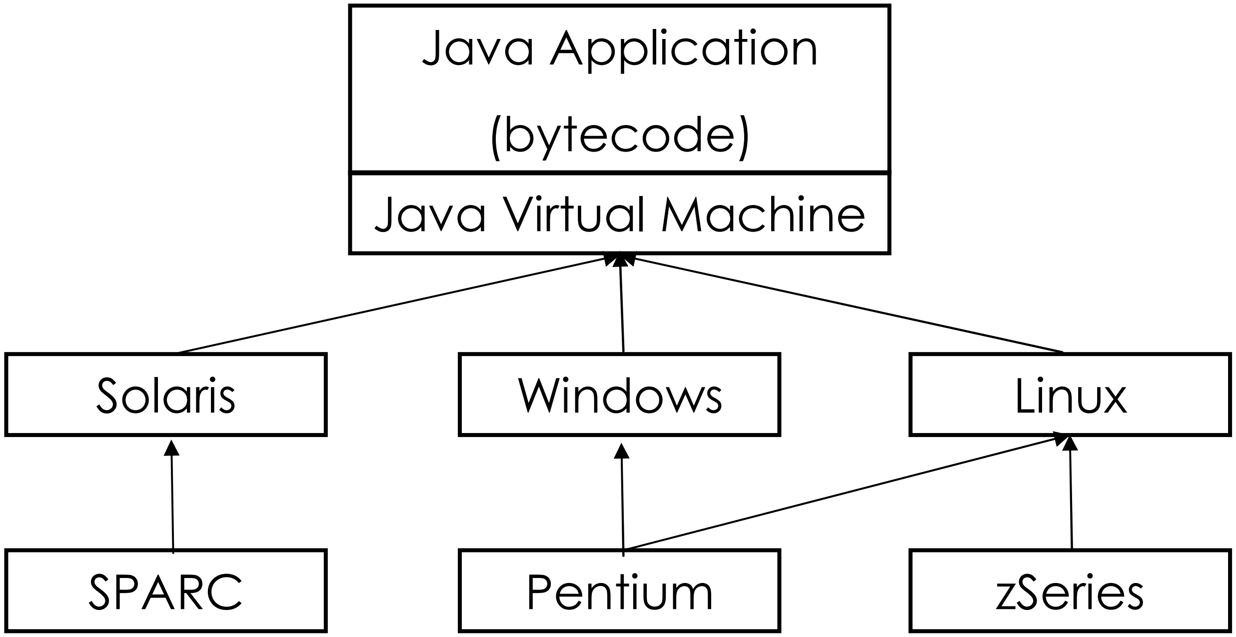
- High-Level Languages

```
public TerminalLogger(Provider theProvider, ILog log, String extension) {
       logDest = log;
       try {
           try {
               myAddress = theProvider.getAddress(extension);
               myTerminal = theProvider.getTerminal(extension);
           } catch (Exception e) {
               tracer.error("Looks like a bad extension");
               throw (e);
           }
           myTerminal.addCallObserver(this);
       } catch (Exception e) {
           tracer.error("TerminalLogger constructor caught " + e);
           return;
       }
   }
```

And even higher abstractions…



An Abstraction is a "many-to-one" mapping



- An abstraction is a "virtual machine" than removes some of the <u>unneeded</u> details and complexity from the base machine
- Reduce the "impedance mismatch" between the base machine and the problem to be solved
- The challenge for the designer is to know when to stop…
  - *Everything should be as simple as possible, but no simpler*

    - Albert Einstein
- Different problems have different sets of "unneeded details"

Separation of Concerns

Separation of Concerns – Examples

- Performance
  - Inter-process communication
  - End-to-end latency
- Security
  - Hardened external interfaces
  - Flow and persistence of cleartext data
- Maintainability
  - Components can be changed independently
- Testability
  - Data can be injected and recorded from individual subsystems
  - Subsystems can run independently

Concerns lead us to choose "Structures" to describe the architecture

- "Software architecture…is the structure or structures of the system…"
- A structure is a binary relation*
  - Define the elements
  - Define the rule
- Examples:
  - Inheritance hierarchy: Element = Class, Rule = "is a subclass of"
  - Pipe and Filter Structure: Elements = Filters and Pipes, Rule = "attached to"
  - Implementation Technology: Elements = Modules and Programming Languages, Rule = "is implemented in"
- "What do the boxes and lines in that diagram mean?

*This description of structures as relations is based on a presentation by David Weiss.

Putting it all together

Using Abstraction and Separation of Concerns to create an Information-hiding Module Structure

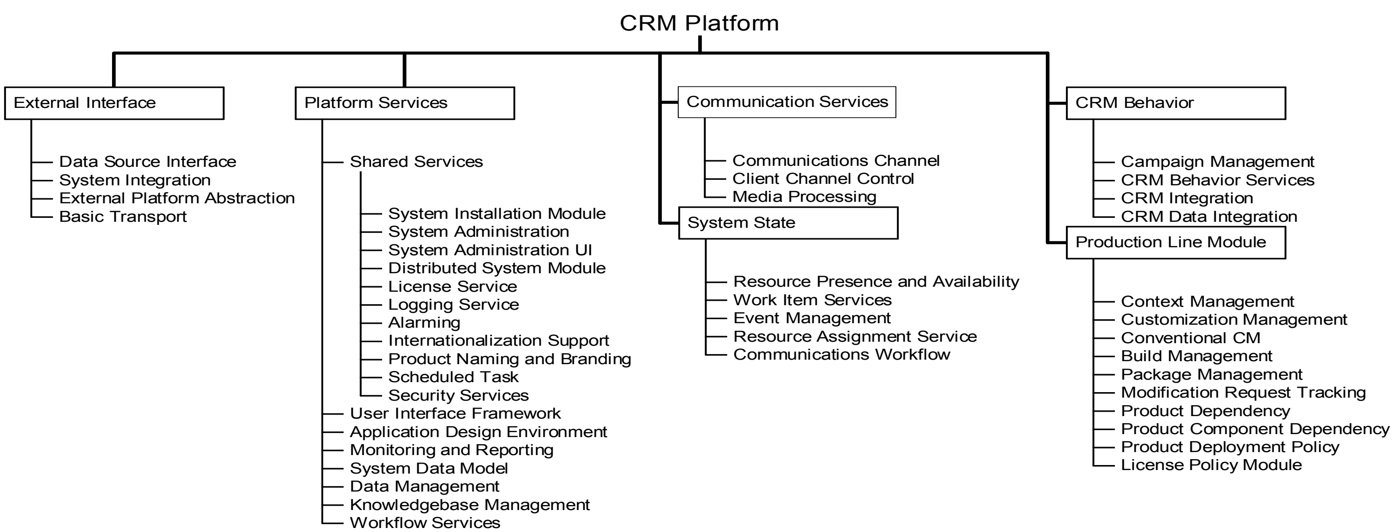
- For tonight…Module = Work Assignment
- Concerns –
  - How do you divide the system so that each module can be built by an individual or team?
  - How do you partition the system so that parts can be changed independently?
  - How do you minimize the risk of "unknowns" at the start of a project? How do you deal with "TBDs" in the requirements?
- David Parnas (1972) proposed that using an "information hiding" criteria to decompose the system into modules will satisfy all of these concerns
- Later work by Baldwin & Clark showed the economic value of modularity, and Sullivan, *et al* showed that Parnas's criteria was optimal in an economic sense.

Information-hiding criteria

- The information here is "design decisions", especially those that are likely to change
- Each module hides (ideally) a single decision
  - The "secret" of the module is the design decision that can change without affecting any other module
  - Hide the secret behind an interface
- The interface defines an abstraction
- Modules are "write-time" entities, not "run-time" entities
  - Hide design information, don't minimize run-time data exchange
- Typical decisions – data representation, persistence mechanism, algorithm implementation, hardware platform, COTS packages (different rates of change)

Information-Hiding Structure

- Element: Modules
- Rule: “refines the secret of”
- This is not O-O…don’t think inheritance
- Example:



Modules exist in many forms

- A set of programs and shared data
- Abstract interface and implementation
- A state machine
- A class
- An abstract data type
- An abstraction
- A collection of macros and preprocessor directives

From David Weiss, "Information Hiding", Avaya Labs Research Report ALR-2002-031

Decision Binding Time

- When do we make each decision?
- Model for decision times is system and technology specific
  - When? How?
- Examples:
  - Specification-time
  - Architecture-time
  - Design-time
  - Code-time
  - Compile-time
  - Link-time
  - Package-time
  - Install-time
  - Configuration-time
  - Run-time

Example from a Java System Under Development

Binding Time	Examples of decisions	Binding Method
Architecture-time	Feature content Platform - programming language, application server, inter-process communication mechanism	Product Specification Architecture Specification
Development-time	Definition of extension points (events and filters)	Code and Metadata
Package-time	Selecting components for a module	Installation-builder scripts
Install & Deploy-time	Selecting which modules to install Setting default parameters Entering initial parameters	Installation options Metadata Setting values in Enterprise Database
Customization-time	Associating a handler to an extension event	Setting values in Enterprise Database
Run-time	"Plug-and-play" automatic selection of a component based on environment	Java dynamic class loading

Reality Check

- Real world is imperfect
  - It is hard to isolate each design decision in a single module
  - It is hard to define interfaces that reflect the desired abstractions
- Why?
  - Reuse of existing assets
  - Mergers and acquisitions
  - Organizational constraints – people, processes, locations
  - Conway's Law – The structure of the system reflects the structure of the organization that builds it - "…[an] organization had eight people who were to produce a COBOL and an ALGOL compiler…five people were assigned to the COBOL job and three to the ALGOL job. The resulting COBOL compiler ran in five phases, the ALGOL compile ran in three."

So what's an architect to do in the "real world"?

- Do the best you can
  - Focus on modularizing high-change/high-risk areas of the system
  - It's OK to have a top-level module that is less cohesive and collects the "leftover" unrelated set of decisions
- There is still value in an "information hiding" and "separation of concerns" mindset
  - Basic tools for dealing with complex systems
  - Examples – Quality Attributes, General Scenarios, Architecture Tactics
- Track the deviations from ideal, know the weaknesses of your design
  - What is going to be hard to change?
  - Is the deviation worth the risk?

Review

- Architecture is the structure or structures that describe the system
- Structure = Mathematical Relation = Elements & Rule
- Abstraction – One-to-Many mapping
- Separation of Concerns – Concerns tell us what structures we need to describe
- Information Hiding – Hide each design decision within a module
- Decision Binding Times
- Reality gets in the way

References

- D.L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules", CACM., Vol 15, no. 12, pp. 1053-1058, Dec. 1972. Reprinted as Chapter 7 in Software Fundamentals: Collected Papers by David L. Parnas, Hoffman and Weiss, eds. Addison-Wesley, 2001.
- D. Parnas, "Designing Software for Ease of Extension and Contraction." IEEE Trans. on Software Engineering, March 1979, pp. 128-138. Reprinted in Software Fundamentals: Collected Papers by David L. Parnas, Hoffman and Weiss, eds. Addison-Wesley, 2001.
- D.L. Parnas, P. Clements, D. Weiss, "The Modular Structure of Complex Systems", IEEE Trans. Software Eng., Vol 11, no. 3, pp. 259-266, March 1985. Reprinted as Chapter 16 in Software Fundamentals: Collected Papers by David L. Parnas, Hoffman and Weiss, eds. Addison-Wesley, 2001.
- Baldwin and Clark, "Modularity in the Design of Complex Engineering Systems", Harvard Business School Working Paper,
- Sullivan, et al, "The Structure and Value of Modularity in Software Design", ESEC/FSE 2001, Vienna, Austria.
- J. Coplien and N. Harrison, Organizational Patterns of Software Development. Prentice Hall, 2004.