Meeting the Challenge of Distributed Real-Time & Embedded (DRE) Systems

Dr. Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

Professor of EECS
Vanderbilt University
Nashville, Tennessee

SATURN Conference, May 10th, 2012
Evolution in DRE Systems

The Past

Standalone real-time & embedded systems
- Stringent quality of service (QoS) demands
  - e.g., latency, jitter, footprint
- Resource constrained

The Present

Distributed real-time & embedded (DRE) systems
- Net-centric systems-of-systems
- Stringent simultaneous QoS demands
  - e.g., dependability, security, scalability, etc.
- More fluid environments & requirements

This talk focuses on technologies & methods for enhancing DRE system QoS, producibility, & quality
Evolution of DRE Systems Development

Mission-critical DRE systems have historically been built directly atop hardware, which is

• Tedious
• Error-prone
• Costly over lifecycles

Consequence: Small changes to legacy software often have big (negative) impact on DRE system QoS & producibility

Technology Problems

• Legacy DRE systems are often:
  • Stovepiped
  • Proprietary
  • Brittle & non-adaptive
  • Expensive
  • Vulnerable
Mission-critical DRE systems have historically been built directly atop hardware, which is
• Tedious
• Error-prone
• Costly over lifecycles

What we need are the means to
• Enhance integrated DRE system capability at lower cost over the lifecycle & across the enterprise
• Reduce cycle time of developing & inserting new technologies into DRE systems

Technology Problems
• Legacy DRE systems are often:
  • Stovepiped
  • Proprietary
  • Brittle & non-adaptive
  • Expensive
  • Vulnerable
What’s So Hard About DRE Software?

**Human Nature**

- Organizational impediments
- Economic impediments
- Administrative impediments
- Political impediments
- Psychological impediments

**Technical Complexities**

**Accidental Complexities**
- Low-level APIs & debug tools
- Algorithmic decomposition

**Inherent Complexities**
- Quality attributes
- Causal ordering
- Scheduling & synchronization
- Deadlock avoidance
- …

www.dre.vanderbilt.edu/~schmidt/reuse-lessons.html
Systematic Reuse Capabilities for DRE Systems

Frameworks

Software Product-lines

Patterns & Pattern Languages

Component-based & Service-Oriented Middleware

Model-Driven Engineering Tools
DRE System Case Study: Boeing Bold Stroke

- Systematic reuse platform for Boeing avionics mission computing
- Bold Stroke defined
  - reference standards
  - software interfaces
  - data formats
- protocols
- system services & reusable components

that enabled distributed computing & allowed distributed applications to coordinate, communicate, execute tasks, & respond to events in an integrated & dependable manner

splc.net/fame/boeing.html
DRE System Case Study: Boeing Bold Stroke

- Systematic reuse platform for Boeing avionics mission computing

- DRE system with 100+ developers, 3,000+ software components, 3-5 million lines of C++/C/Ada/Java

- Based on COTS hardware, networks, operating systems, languages, & middleware

- Used as an Open Experimentation platform (OEP) for DARPA PCES, MoBI ES, SEC, NEST, & MICA programs

splc.net/fame/boeing.html
Applying COTS to Bold Stroke

COTS & standards-based middleware, language, OS, network, & hardware platforms

- Real-time CORBA (TAO) middleware
- ADAPTIVE Communication Environment (ACE)
- C++, C, Ada, & Real-time Java
- VxWorks operating system
- VME, 1553, & Link16
- PowerPC

www.dre.vanderbilt.edu/ACE
www.dre.vanderbilt.edu/TAO
Benefits of Using COTS

- Save a considerable amount of time/effort compared with traditional approach to handcrafting capabilities
- Leverage industry “best practices” & patterns in pre-packaged (& ideally) standardized form
Limitations of Using COTS

- QoS of COTS components is not always suitable for mission-critical DRE systems

- COTS technologies address some, but by no means all, domain-specific challenges associated with developing mission-critical DRE systems

What was needed was a systematic reuse technology for organizing & automating key roles & responsibilities in an application domain.
Motivation for Software Product-lines (SPLs)

Legacy avionics mission computing systems are:

- Stovepiped
- Proprietary
- Brittle & non-adaptive
- Expensive
- Vulnerable

Consequences:

- Small changes to requirements & environments can break nearly anything
- Lack of any resource can break nearly everything
Motivation for Software Product-lines (SPLs)

- SPLs factor out general-purpose & domain-specific services from traditional application responsibility in DRE systems
- Manage software variation while reusing large amounts of code that implement common features within a particular domain
- SPLs offer many opportunities to configure product variants
  - e.g., component distribution & deployment, user interfaces & operating systems, algorithms & data structures, etc.
Overview of Software Product-lines (SPLs)

- SPL characteristics are captured via *Scope, Commonalities, & Variabilities (SCV) analysis*
  - This process can be applied to identify commonalities & variabilities in a domain to guide development of a SPL

- Applying SCV to Bold Stroke
  - Scope defines the domain & context of the SPL
  - e.g., Bold Stroke component architecture, object-oriented application frameworks, & associated components (GPS, Airframe, & Display)
Commonalities describe the attributes that are common across all members of the SPL family

- Common object-oriented frameworks & set of component types
  - e.g., GPS, Airframe, Navigation, & Display components
- Common middleware infrastructure
  - e.g., Real-time CORBA & Lightweight CORBA
- Component Model (CCM) variant called Prism
Variabilities describe the attributes unique to the different members of the family:

- Product-dependent component implementations (GPS/INS)
- Product-dependent component connections
- Product-dependent component assemblies
  - e.g., different packages for different customers & countries
- Different hardware, OS, & network/bus configurations

Patterns & frameworks are essential for developing reusable SPLs.

Applying SCV to the Bold Stroke SPL
Pattern-oriented domain-specific application framework

- Configurable to variable infrastructure configurations
- Supports systematic reuse of mission computing functionality
- 3-5 million lines of C++, C, Ada, & Real-time Java
- Based on many architecture & design patterns

Patterns & frameworks are also used throughout Bold Stroke COTS software infrastructure
Overview of Patterns

• Present solutions to common software problems arising within a particular context

• Capture recurring structures & dynamics among software participants to facilitate reuse of successful designs

• Help resolve key software design forces
  • Flexibility
  • Extensibility
  • Dependability
  • Predictability
  • Scalability
  • Efficiency

• Codify expert knowledge of design strategies, constraints, & best practices

The Proxy Pattern
Overview of Pattern Languages

Motivation

- Individual patterns & pattern catalogs are insufficient
- Software modeling methods & tools largely just illustrate *what/how* – not *why* – systems are designed

Benefits of Pattern Languages

- Define a *vocabulary* for talking about software development problems
- Provide a *process* for the orderly resolution of these problems
- Help to generate & reuse software *architectures*
Legacy Avionics Architectures

Key system characteristics

- Hard & soft real-time deadlines
  - ~20-40 Hz
- Low latency & jitter between boards
  - ~100 μsecs
- Periodic & aperiodic processing
- Complex dependencies
- Continuous platform upgrades

Avionics Mission Computing Functions

- Weapons targeting systems (WTS)
- Airframe & navigation (Nav)
- Sensor control (GPS, IFF, FLIR)
- Heads-up disSPLy (HUD)
- Auto-pilot (AP)

1: Sensors generate data
2: I/O via interrupts
3: Sensor proxies process data & pass to missions functions
4: Mission functions perform avionics operations
Legacy Avionics Architectures

Key system characteristics
- Hard & soft real-time deadlines
  - ~20-40 Hz
- Low latency & jitter between boards
  - ~100 usecs
- Periodic & aperiodic processing
- Complex dependencies
- Continuous platform upgrades

Limitations with legacy avionics architectures
- Stovepiped
- Proprietary
- Expensive
- Vulnerable
- Tightly coupled
- Hard to schedule
- Brittle & non-adaptive
# Decoupling Avionics Components

## Context
- I/O driven DRE application
- Complex dependencies
- Real-time constraints

## Problems
- Tightly coupled components
- Hard to schedule
- Expensive to evolve

## Solution
- Apply the *Publisher-Subscriber* architectural pattern to distribute periodic, I/O-driven data from a single point source to a collection consumers

## Structure

<table>
<thead>
<tr>
<th>Publisher</th>
<th>Event Channel</th>
<th>Subscriber</th>
</tr>
</thead>
<tbody>
<tr>
<td>produce</td>
<td>attachPublisher</td>
<td>consume</td>
</tr>
<tr>
<td></td>
<td>detachPublisher</td>
<td></td>
</tr>
<tr>
<td></td>
<td>attachSubscriber</td>
<td></td>
</tr>
<tr>
<td></td>
<td>detachSubscriber</td>
<td></td>
</tr>
<tr>
<td></td>
<td>pushEvent</td>
<td></td>
</tr>
</tbody>
</table>

## Dynamics

- : Publisher
- : Event Channel
- : Subscriber
- : Event

**Event**
- create
- receive

**Filter**
- filterEvent
Applying Publisher-Subscriber to Bold Stroke

Bold Stroke uses the Publisher-Subscriber pattern to decouple sensor processing from mission computing operations:

- Anonymous publisher & subscriber relationships
- Group communication
- Asynchrony

Implementing Publisher-Subscriber pattern for mission computing:

- Event notification model
  - Push control vs. pull data interactions
- Scheduling & synchronization strategies
  - e.g., priority-based dispatching & preemption
- Event dependency management
  - e.g., filtering & correlation mechanisms
Distributing Avionics Components

<table>
<thead>
<tr>
<th>Context</th>
<th>Problems</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Mission computing requires remote IPC</td>
<td>• Applications need capabilities to:</td>
<td>• Apply the <strong>Broker</strong> architectural pattern to provide platform-neutral communication between mission computing boards</td>
</tr>
<tr>
<td>• Stringent DRE requirements</td>
<td>• Support remote communication</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Provide location transparency</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Handle faults</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Manage end-to-end QoS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Encapsulate low-level system details</td>
<td></td>
</tr>
</tbody>
</table>
## Distributing Avionics Components

<table>
<thead>
<tr>
<th>Context</th>
<th>Problems</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Mission computing requires remote IPC</td>
<td>• Applications need capabilities to:</td>
<td>• Apply the Broker architectural pattern to provide platform-neutral communication between mission computing boards</td>
</tr>
<tr>
<td>• Stringent DRE requirements</td>
<td>• Support remote communication</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Provide location transparency</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Handle faults</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Manage end-to-end QoS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Encapsulate low-level system details</td>
<td></td>
</tr>
</tbody>
</table>

### Structure & Dynamics

![Diagram of the Broker architectural pattern](image)
Bold Stroke uses the *Broker* pattern to shield distributed applications from environment heterogeneity, *e.g.*,

- Programming languages
- Operating systems
- Networking protocols
- Hardware

A key consideration for implementing the *Broker* pattern for mission computing applications is *QoS* support

- *e.g.*, latency, jitter, priority preservation, dependability, security, etc.
Key Patterns Used to Implement Broker

- **Wrapper facades** enhance portability
- **Proxies & adapters** simplify client & server applications, respectively
- **Component Configurator** dynamically configures Factories
- **Factories** produce Strategies
- **Strategies** implement interchangeable policies
- Concurrency strategies use Reactor & Leader/Followers
- **Acceptor-Connector** decouples connection management from request processing
- **Managers** optimize request demultiplexing

www.dre.vanderbilt.edu/~schmidt/PDF/ORB-patterns.pdf
### Enhancing Broker Flexibility with Strategy

<table>
<thead>
<tr>
<th>Context</th>
<th>Problem</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Multi-domain reusable middleware Broker</td>
<td>• Flexible Brokers must support multiple policies for event &amp; request demuxing, scheduling, (de)marshaling, connection mgmt, request transfer, &amp; concurrency</td>
<td>• Apply the <em>Strategy</em> pattern to factory out commonality amongst variable Broker algorithms &amp; policies</td>
</tr>
</tbody>
</table>

**Diagram:**
- **Hook for the concurrency strategy**
- **Hook for the request demuxing strategy**
- **Hook for the event demuxing strategy**
- **Hook for marshaling strategy**
- **Hook for the connection management strategy**
- **Hook for the underlying transport strategy**
<table>
<thead>
<tr>
<th>Context</th>
<th>Problem</th>
<th>Solution</th>
</tr>
</thead>
</table>
| A heavily strategized framework or application | Aggressive use of Strategy pattern creates a configuration nightmare  
• Managing many individual strategies is hard  
• It’s hard to ensure that groups of semantically compatible strategies are configured | Apply the *Abstract Factory* pattern to consolidate multiple Broker strategies into semantically compatible configurations |

Concrete factories create groups of strategies
## Configuring Factories w/ Component Configurator

<table>
<thead>
<tr>
<th>Context</th>
<th>Problem</th>
<th>Solution</th>
</tr>
</thead>
</table>
| Resource constrained systems     | • Prematurely committing to a Broker configuration is inflexible & inefficient  
|                                  | • Certain decisions can’t be made until runtime  
|                                  | • Users forced to pay for components they don’t use                     | • Apply the **Component Configurator** pattern to assemble the desired Broker factories & strategies more effectively |

- Broker strategies are decoupled from when the strategy implementations are configured into Broker
  - This pattern can reduce the memory footprint of Broker implementations
Benefits of Patterns

- Enables reuse of software architectures & designs
- Improves development team communication
- Convey “best practices” intuitively
- Transcends language-centric biases/myopia
- Abstracts away from many unimportant details

www.dre.vanderbilt.edu/~schmidt/patterns.html
Limitations of Patterns

- Require significant tedious & error-prone human effort to handcraft pattern implementations
- Can be deceptively simple
- Leaves many important details unresolved, particularly for DRE systems

We therefore need more than just patterns to achieve effective systematic reuse

www.dre.vanderbilt.edu/~schmidt/patterns.html
Overview of Systematic Reuse Paradigms

Class Library Architecture
- A class is a unit of abstraction & implementation in an OO programming language, i.e., a reusable type that often implements patterns
- Classes are typically passive

Framework Architecture
- A framework is an integrated set of classes that collaborate to produce a reusable architecture for a family of applications
- Frameworks implement pattern languages

Component/Service-Oriented Architecture
- A component/service is an encapsulation unit with one or more interfaces that provide clients with access to its services
- Components/services can be deployed & configured via assemblies
### Applying Frameworks to Bold Stroke

**Framework characteristics**

- Frameworks exhibit “inversion of control” at runtime via callbacks.

- Frameworks provide integrated domain-specific structures & functionality.

- Frameworks are “semi-complete” applications.

www.dre.vanderbilt.edu/~schmidt/frameworks.html
Benefits of Frameworks

- Design reuse
  - e.g., by implementing patterns that guide application developers through the steps necessary to ensure successful creation & deployment of avionics software
Benefits of Frameworks

- Design reuse
  - e.g., by implementing patterns that guide application developers through the steps necessary to ensure successful creation & deployment of avionics software

- Implementation reuse
  - e.g., by amortizing software lifecycle costs & leveraging previous development & optimization efforts

```java
package org.apache.tomcat.session;
import org.apache.tomcat.core.*;
import org.apache.tomcat.util.StringManager;
import java.io.*;
import java.net.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

/**
 * Core implementation of a server session
 * @author James Duncan Davidson [duncan@eng.sun.com]
 * @author James Todd [gonzo@eng.sun.com]
 */
public class ServerSession {
    private StringManager sm = StringManager.getManager("org.apache.tomcat.session");
    private Hashtable values = new Hashtable();
    private Hashtable appSessions = new Hashtable();
    private String id;
    private long creationTime = System.currentTimeMillis();
    private long thisAccessTime = creationTime;
    private int inactiveInterval = -1;

    ServerSession(String id) { this.id = id; }

    public String getId() { return id; }
    public long getCreationTime() { return creationTime; }

    public ApplicationSession getApplicationSession(Context context, boolean create) {
        ApplicationSession appSession = (ApplicationSession)appSessions.get(context);
        if (appSession == null && create) {
            // XXX
            // sync to ensure valid?
            appSession = new ApplicationSession(id, this, context);
            appSessions.put(context, appSession);
        }

        // XXX
        // make sure that we haven't gone over the end of our
        // inactive interval -- if so, invalidate & create
        // a new appSession
        return appSession;
    }

    public long getCreationTime() { return creationTime; }

    void removeApplicationSession(Context context) {
        appSessions.remove(context);
    }
}
```
Benefits of Frameworks

• Design reuse
  e.g., by implementing patterns that guide application developers through the steps necessary to ensure successful creation & deployment of avionics software

• Implementation reuse
  e.g., by amortizing software lifecycle costs & leveraging previous development & optimization efforts

• Validation reuse
  e.g., by amortizing the efforts of validating application- & platform-independent portions of software, thereby enhancing software reliability & scalability

www.dre.vanderbilt.edu/scoreboard
Limitations of Frameworks

- Frameworks are powerful, but hard to develop & use effectively
- Significant time required to evaluate applicability & quality of a framework for a particular domain
- Debugging is tricky due to inversion of control
- Verification & validation is tricky due to dynamic binding
- May incur performance overhead due to extra (unnecessary) levels of indirection

We thus need something simpler than frameworks to achieve systematic reuse for DRE systems
Historically, mission-critical DRE apps were built directly atop hardware & OS

- Tedious, error-prone, & costly over lifecycles

There are layers of middleware, just like there are layers of networking protocols

Standards-based COTS DRE middleware helps:

- Control end-to-end resources & QoS
- Leverage hardware & software technology advances
- Evolve to new environments & requirements
- Provide a wide array of reusable, off-the-shelf developer-oriented services

Middleware is pervasive in enterprise domain & is becoming pervasive in DRE domain
Operating System & Protocols

- Operating systems & protocols provide mechanisms to manage endsystem resources, e.g.,
  - CPU scheduling & dispatching
  - Virtual memory management
  - Secondary storage, persistence, & file systems
  - Local & remote interprocess communication (IPC)

- OS examples
  - UNIX/Linux, Windows, VxWorks, QNX, etc.

- Protocol examples
  - TCP, UDP, IP, SCTP, RTP, etc.

---

**INTERNETWORKING ARCH**

```
RTP  TFTP  FTP  HTTP
DNS  TELNET  TCP
UDP  IP
Fibre Channel
```

**20th Century**

---

**MIDDLEWARE ARCH**

```
Middleware Applications
Middleware Services
```

```
Solaris  VxWorks
Win2K  Linux  LynxOS
```

**21st Century**
Host Infrastructure Middleware

- Host infrastructure middleware encapsulates & enhances native OS mechanisms to create reusable network programming objects
- These components abstract away many tedious & error-prone aspects of low-level OS APIs

Examples
- Java Virtual Machine (JVM), Common Language Runtime (CLR), ADAPTIVE Communication Environment (ACE)
Distribution Middleware

- **Distribution middleware** defines higher-level distributed programming models whose reusable APIs & components automate & extend native OS capabilities.

- **Examples**
  - OMG Real-time CORBA & DDS, Sun RMI, Microsoft DCOM, W3C SOAP.

---

Distribution middleware avoids hard-coding client & server application dependencies on object location, language, OS, protocols, & hardware.
Common Middleware Services

- Common middleware services augment distribution middleware by defining higher-level domain-independent services that focus on programming “business logic”

- Examples
  - W3C Web Services, CORBA Component Model & Object Services, Sun’s J2EE, Microsoft’s .NET, etc.

- Common middleware services support many recurring distributed system capabilities, e.g.,
  - Transactional behavior
  - Authentication & authorization,
  - Database connection pooling & concurrency control
  - Active replication
  - Dynamic resource management
Domain-Specific Middleware

- Domain-specific middleware services are tailored to the requirements of particular domains, such as telecom, e-commerce, health care, process automation, or aerospace.

Examples

Siemens MED Syngo
- Common software platform for distributed electronic medical systems
- Used by all Siemens MED business units worldwide

Boeing Bold Stroke
- Common software platform for Boeing avionics mission computing systems

Modalities
- e.g., MRI, CT, CR, Ultrasound, etc.
Applying Component Middleware to Bold Stroke

Product-line component model

- Configurable for product-specific functionality & execution environment
- Single component development policies
- Standard component packaging mechanisms
- 3,000+ software components
Benefits of Component Middleware

- Creates a standard “virtual boundary” around application component implementations that interact only via well-defined interfaces
- Define standard container mechanisms needed to execute components in generic component servers
- Specify the infrastructure needed to configure & deploy components throughout a distributed system
Limitations of Component Middleware

- Limit to how much application functionality can be refactored into reusable COTS component middleware
Limitations of Component Middleware

- Limit to how much application functionality can be refactored into reusable COTS component middleware
- Middleware itself has become hard to provision/use

Load Balancer
  FT CORBA
  RT CORBA + DRTSJ
  RTOS + RT Java
  IntServ + Diffserv

- Workload & Replicas
- Connections & priority bands
- CPU & memory
- Network latency & bandwidth

Mission Computing Services
Middleware Infrastructure
Operating System
Networking Interfaces
Hardware (CPU, Memory, I/O)
Limitations of Component Middleware

• Limit to how much application functionality can be refactored into reusable COTS component middleware
• Middleware itself has become hard to provision/use
• Large # of components can be tedious & error-prone to configure & deploy without proper integration tool support
Limitations of Component Middleware

- Limit to how much application functionality can be refactored into reusable COTS component middleware
- Middleware itself has become hard to provision/use
- Large # of components can be tedious & error-prone to configure & deploy without proper integration tool support
- There are many middleware technologies to choose from
Applying MDE to Bold Stroke

Model-driven engineering (MDE)

- Apply MDE tools to
  - Model
  - Analyze
  - Synthesize
  - Provision middleware & application components
- Configure product variant-specific component assembly & deployment environments
- Model-based component integration policies

www.isis.vanderbilt.edu/projects/mobies
Applying MDE to Bold Stroke

Formal mission specs, subsystem models, & computational constraints combined into integrated MDE tool chain & mapped to execution platforms

Interaction is based on mission-specific ontologies & semantics

CODE GENERATORS

C++
SMV
SPIN
Real-time Java
Ptolemy

XML

PowerPC
ACE+TAO
Bold Stroke

APPLICATION MODELING TOOLS

UML/ Rose
ESML/ GME
PI CML/ GME

EMBEDDED PLATFORM MODEL

Avionics Mission Control Architecture

Mission Computing Services
Middleware Infrastructure
Operating System
Networking Interfaces
Hardware (CPU, Memory, I/O)

ANALYSIS TOOLS

Stateflow
Statecharts
Ptolemy
Simulink
XML

CODE GENERATORS

www.rl.af.mil/tech/programs/MoBIES/
Benefits of MDE

- Increase expressivity
  - e.g., linguistic support to better capture design intent
- Increase precision
  - e.g., mathematical tools for cross-domain modeling, synchronizing models, change propagation across models, modeling security & other QoS aspects
- Achieve reuse of domain semantics
  - Generate code that’s more “platform-independent” (or not)!
- Support DRE system development & evolution
Limitations of MDE

Model & Component Library

Applications

- Modeling technologies are still maturing & evolving
  - i.e., non-standard tools
- Magic (& magicians) are still necessary for success
Ingredients for Success with Systematic Reuse

Key Technologies

- Standard Middleware, Frameworks, & Components
- Patterns & Pattern Languages
- Model-driven Software Development

Experienced Senior Architects

- Responsible for communicating completeness, correctness, & consistency of all parts of the software architecture to the stakeholders

Solid Key Developers

- Design responsibility (maintenance, evolution) for a specific architectural topic

Enlightened Managers

- Must be willing to defend the sacrifice of some short-term investment for long-term payoff

Accepted Business Drivers

- i.e., need a “succeed or die” mentality

It’s crucial to have an effective process for growing architects & key developers
Traits of Dysfunctional Software Organizations

Process Traits

- Death through quality
  - “Process bureaucracy”
- Analysis paralysis
  - “Zero-lines of code seduction”
- Infrastructure churn
  - e. g., programming to low-level APIs

Organizational Traits

- Disrespect for quality developers
  - “Coders vs. developers”
- Top-heavy bureaucracy

Sociological Traits

- The “Not Invented Here” syndrome
- Modern method madness

www.dre.vanderbilt.edu/~schmidt/editorials.html
Traits of Highly Successful Software Organizations

Strong leadership in business & technology
  • e.g., understand the role of software technology
  • Don’t wait for “silver bullets”

Clear architectural vision
  • e.g., know when to buy vs. build
  • Avoid worship of specific tools & technologies

Effective use of prototypes & demos
  • e.g., reduce risk & get user feedback

Commitment to/from skilled developers
  • e.g., know how to motivate software developers & recognize the value of thoughtware
Consequences of COTS & IT Commoditization

- More emphasis on integration rather than programming
- Increased technology convergence & standardization
- Mass market economies of scale for technology & personnel
- More disruptive technologies & global competition
- Lower priced—but often lower quality—hardware & software components
- The decline of internally funded R&D
- Potential for complexity cap in next-generation complex systems

Not all trends bode well for long-term competitiveness of traditional leaders

Ultimately, competitiveness depends on success of long-term R&D on complex distributed real-time & embedded (DRE) systems
Concluding Remarks

- The growing size & complexity of DRE systems requires significant innovations & advances in processes, methods, platforms, & tools
- Not all technologies provide precision of legacy real-time & embedded systems
- Advances in Model-Driven Engineering & component/SOA-based DRE system middleware are needed to address future challenges
- Significant groundwork laid in DARPA & NSF programs
- Much more R&D needed to assure key quality attributes of DRE systems

See blog.sei.cmu.edu for coverage of SEI R&D activities
Further Reading

ULS systems are socio-technical ecosystems comprised of software-reliant systems, people, policies, cultures, & economics that have unprecedented scale in the following dimensions:

- # of lines of software code & hardware elements
- # of connections & interdependencies
- # of computational elements
- # of purposes & user perception of purposes
- # of routine processes & “emergent behaviors”
- # of (overlapping) policy domains & enforceable mechanisms
- # of people involved in some way
- Amount of data stored, accessed, & manipulated
- … etc …

www.sei.cmu.edu/uls

See blog.sei.cmu.edu for discussions of software R&D activities
Focus of the report is on ensuring the DoD has the technical capacity & workforce to design, produce, assure, & evolve innovative software-reliant systems in a predictable manner, while effectively managing risk, cost, schedule, & complexity.

Sponsored by Office of the Secretary of Defense (OSD) with assistance from the National Science Foundation (NSF), & Office of Naval Research (ONR), www.nap.edu/openbook.php?record_id=12979&page=R1

See blog.sei.cmu.edu for discussions of software R&D activities