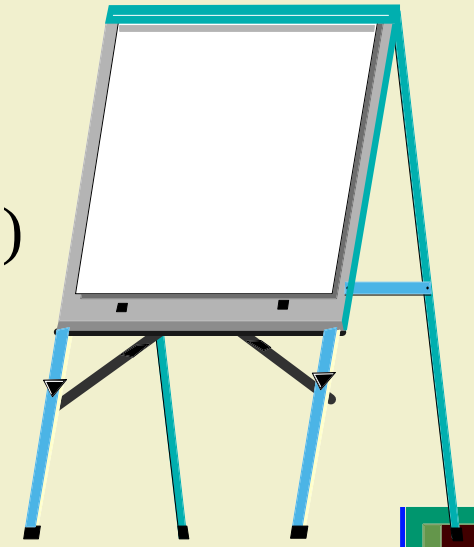
Legacy Systems

of the past, present & future

for NY SPIN

by Conrad Weisert (**www.idinews.com**)

November 8, 2005



What's a *Legacy System*?

An application with these characteristics:

- User interface is unfriendly and error-prone
- Maintenance documentation is non-existent or hopelessly out of date
- Programs are disorganized, inflexible, hard to understand, and very expensive to change
- Database is full of inconsistencies and redundancy
- *What else?*

Other common characteristics

- May run on obsolescent platform
  - mainframe computer system
  - dedicated online network
- May depend on old development tools
  - programming languages
  - source-code library managers
  - C.A.S.E

Role of *Legacy Systems*

- Legacy systems are **bad.**
  - Costly, risky, and unpredictable to maintain
- But an organization may depend on them for:
  - Mainstream applications
  - High security operation
  - High volume transaction processing
  - High performance number crunching

such as. . .?

Where did Legacy Systems come from ?

"The traditional approach"

- Conventional Wisdom:
  - Developers of the 1970s and earlier were dismally unenlightened compared with us, who practice modern methods & technologies
  - Written in a *dinosaur* programming language
  - Ongoing maintenance under deadline pressures makes a given system worse and worse.
- We feel sympathy for anyone stuck with maintaining or operating one.

So, what's the problem?

"Those ancient applications are being rapidly replaced by modern systems, aren't they?"

"Yes, we're replacing our old systems, but we're creating new legacy applications that are just as unmaintainable as the old ones!"

Why? How could this happen?

The Blame Game:
Contributions to new legacy systems

1. **from academic institutions**
2. from in-house management
3. from fad *breakthrough* methodologies
4. from contract developer firms

Academic contributions to new legacy systems

- **Conventional wisdom** criticizes curricula for lack of *real-world* orientation.
- But that's rarely the *only* problem -- CS and MIS departments often exhibit serious:
  - Faculty shortcomings
  - Infrastructure shortcomings

Faculty contributions

- Many faculty, esp. in computer science:
  - Don't practice what they preach
  - Are indifferent to (or ignorant of) ***quality*** issues
  - Fail to encourage ***critical thinking***
  - Are naive about
    - development *projects*
    - the role of the programmer
    - organizational environments

Practicing what we preach

example 1: requirements and specifications

- In **systems analysis** courses instructors emphasize the vital importance of rigorous external specifications (detailed user requirements).
- Yet in **programming** courses instructors hand out dreadful problem specifications in assignments

Dreadful in what ways?

Practicing what we preach

example 2: module library

- In *software engineering* courses, instructors stress the value of component **re-use**
- But few universities support a **library** of re-usable components that students and faculty can draw on and contribute to.

Faculty indifference to quality

- Many advanced programming students are shocked the first time they don't get an **A** on an assigned program that:
  - runs to completion
  - gets the right answer
  - uses the prescribed algorithm

i.e. that's free of "defects"

- That has never happened to them before! *So they claim -- are they right?*

A real-world grading policy

- **Any correct and complete problem solution earns at least a C.**
- **Work that is unusually thorough, well-organized, nicely presented, innovative, or in some other way superior to a minimum required solution, earns at least a B.**
- **A grade of A is earned for work that is *outstanding* in some way.**
- **Partial or flawed problem solutions can earn any grade A through F depending on the nature of the omission or errors and on the quality of the work that's turned in.**

from `www.ece.iit.edu/~cweisert/gradepolicy.html`

Is that policy reasonable? realistic?

Why don't more C.S. instructors teach and demand quality?

- Some are unaware of it themselves

  What can we do about that?

- Many are too busy.
  - They have to rely on teaching assistants to grade students' work.
  - They hope to avoid arguing with students over qualitative (or "subjective") aspects of grading.

Role of the programmer

- Has evolved over a half century.
- In 2005 it is usually (pick one):
  - Given a specification, code and test one or more programs to satisfy it.
  - In collaboration with potential users, develop software that satisfies them.
  - Given a well-defined problem, produce a (usually computer-based) solution.
  - All of the above

Role of the programmer

- Has evolved over a half century.
- In 2005 it is usually (pick one):
  - Given a specification, code and test one or more programs to satisfy it.
  - In collaboration with potential users, develop software that satisfies them.
  -  Given a well-defined problem, produce a (probably computer-based) solution.

Who defines the problem?

The programmer as problem solver

- Must we always write code?
  - **Packaged application** product solution
  - **Re-used components** solution
  - **Spreadsheet** solution
  - **Manual process** solution
  - . . . etc.
- How can we
  - measure productivity?
  - reward performance?

Critical thinking skills

- Do universities teach them?
- Do managers encourage (or tolerate) them?

Blind acceptance and uncritical thinking

- Q: "Why are you drawing that diagram?"
- A: "Because our methodology requires it."

Uncritical thinking 1

- "Design and develop a function to determine the prime factors of an integer."
  (first week exercise a second programming course)

- Student responses:
  - ~1/3 completed a reasonable solution
  - ~1/3 asked
    "How?" or "Should I use an array for the results?"
    What does that tell us?
  - ~1/3 submitted a seriously flawed solution
    What was unsatisfactory about them?

Uncritical thinking 2

- "During the break, write a program fragment to display the sum of the first 50 integers."

workshop exercise in a C++ programming course immediately after reviewing loop-control constructs

Uncritical thinking 2

- "During the break, write a program fragment to display the sum of the first 50 integers."

- A naive solution:

```
total = 0;
for (ctr = 1; ctr <= 50; ++ctr)
  total += ctr;
cout << total;
```

Uncritical thinking 2

- "During the break, write a program fragment to display the sum of the first 50 integers."

- A more "elegant" naive solution:

```
for (ctr=1, total=0; ctr <= 50;
                     total += ctr++);
cout << total;
```

Uncritical thinking 2

- "During the break, write a program fragment to display the sum of the first 50 integers."

Was that a trick question?

- A *reasonable* solution:

```
cout << 1275;
```

- A somewhat more thoughtful solution:

```
const int n = 50;

cout << (n * (n + 1)) / 2;
```

Faculty naiveté about development environments

- Some faculty are unaware of **roles** in an organization:
  - programmer
  - systems analyst
  - data administrator
  - database manager
  - quality assurance staff
  - project manager
- *their interrelationships*
- *measures of performance*

For more on academic shortcomings see
www.idinews.com/academic.html

Contributions to new legacy systems

1. from academic institutions
2. from **in-house management**
3. from fad "breakthrough" methodologies
4. from contract developer firms

Misconceptions by naive managers

- "Silver bullet" tools & technologies
- Programmers (or "developers") are fungible
- "I.T. projects are hopeless, anyway"

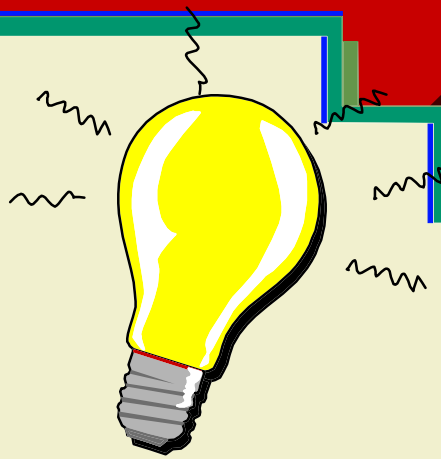
Decline of methodology infrastructure

- Adopting new tools, methods, support functions, etc. usually demands justification (ROI)
- But discarding old ones doesn't!
- Standards and methodology are perceived as bureaucratic "red tape", especially after a merger or reorganization.

Contributions to new legacy systems

1. from academic institutions
2. from in-house management
3. from **fad *breakthrough* methodologies**
4. from contract developer firms

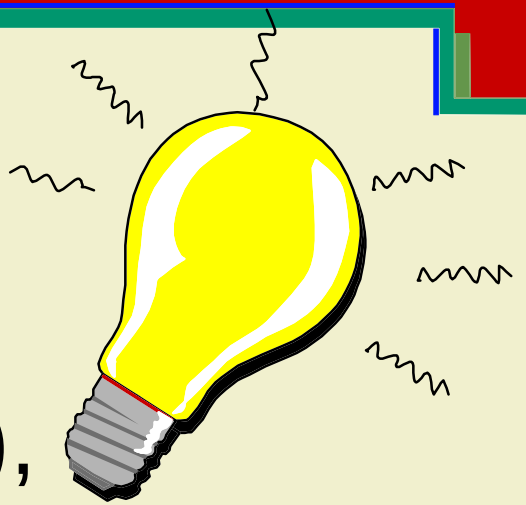
Major Dramatic Breakthroughs



- Every 3 or 4 years someone devises a *major dramatic breakthrough* (**MDB**) in software development.
- Some MDBs are **evolutionary**, others **revolutionary**.
- Each MDB claimed productivity gain between 1.5x and 10x.

Therefore . . .

MDB cumulative impact



- Therefore (conservatively), we are now developing software with less than 1/1000 the effort required in 1960.

or

- We're now routinely developing programs 3 orders of magnitude bigger and more complex.

Why didn't that happen?

- Half-hearted adoption
- Guruism
- Obfuscation and intimidation by insiders.
- Overhyped fad MDBs -- Some may even be harmful! *Which ones?*
- *"If it ain't broke, fix it anyway!"*

Which **fad methodologies?**

We'll look at three examples:

1. A **systems analysis** methodology: UML with UP and use-case requirements
2. A **programming** methodology: Extreme programming (XP) & similar "agile" methods
3. A **platform**: Java

Each of them has positive aspects and offers benefits to discriminating users.

For more information about impact of methodologies see **`www.idinews.com/fixit.html`**

The Requirements Crisis

- Large projects that try to follow UML/UP often experience a serious deficiency in gathering, organizing, understanding, and approving the **users' requirements**.
  - Abandonment of structure, in particular:
    - Where do we begin?
    - How do we know when we're done?
  - Overwhelming detail
    - Compare with DeMarco's "Victorian Novel" approach to system specification

How did we come to abandon requirements structure?

A chronology from ~1991

1. Object-oriented analysis (OOA) is *good.*
2. UML (Booch-Rumbaugh) is *standard* for OOA.
3. But sponsoring users and other non-technical audience can't understand UML reqs. specifications.
4. Jacobson adds use-cases to UML.
5. Users can't understand use-cases either.
6. Unstructured "want lists" substituted.

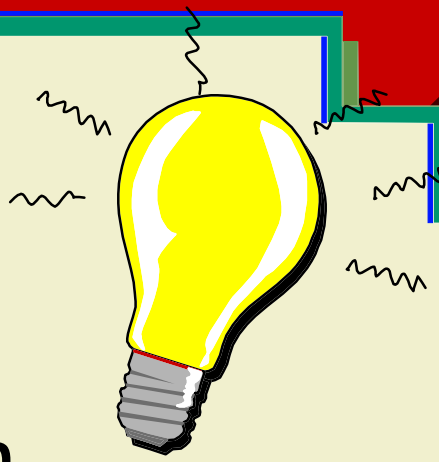
How did we come to abandon requirements structure?

A chronology from ~1991

1. Object-oriented analysis (OOA) is *good.*
2. UML (Booch-Rumbaugh) is *standard* for OOA.
3. Sponsoring users and other non-technical audience can't understand UML reqs. specifications.
4. Jacobson adds use-cases to UML.
5. Users can't understand use-cases either.
6. Unstructured "want lists" substituted.

Reject / ignore anything associated with "traditional" or "structured" systems analysis (SA)

"UML is a language, not a process (life-cycle)"



- Immune from most criticism
- Works with *any* "methodology" that's
  - use-case centric
  - iterative and incremental
- By the way, here's one: **UP**
  - muddles analysis and design ("elaboration phase")

Clearing it all up

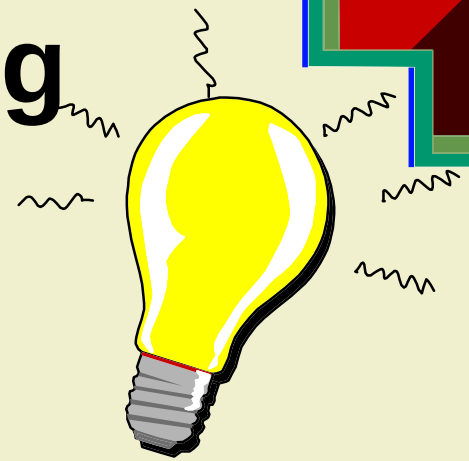
"During the elaboration phase, as we have already noted, we build the architecture. We identify the use cases that have a significant impact on the architecture. We realize these use cases as collaborations. It is in this way that we identify most of the subsystems and interfaces -- at least the ones that are architecturally interesting. Once most of the subsystems and interfaces are identified, we flesh them out, that is, write the code that implements them. Some of this work is done before we release the architectural baseline and it continues throughout all of the workflows."

- Ivar Jacobson

Responding to the requirements crisis

- Give up in favor of "iterative and incremental development"
- What's the impact on:
  - estimating time and cost
  - return on investment
  - system quality

Extreme programming (XP)



- What kinds of end-product do projects develop?
  a. A custom application
  b. A software product
  c. One or more reusable ("utility" or "generic") components

Which of those are suited to XP?

What else is there?

A common policy for application system development

"We will develop custom application software only when it is shown that no suitable packaged software product exists."

- Not a variant, but the mainstream in many, probably most, organizations in 2005
- This policy is foreclosed when requirements emerge by iteration.

Another common policy for application system development

"When we do develop custom software, we draw upon existing components wherever we can and we try to contribute new reusable components.

- Component reuse provides huge benefits in productivity, quality, and reliability.
- This policy is foreclosed by **YAGNI**.

What's that?

Limitations of extreme programming (XP)

- XP works only when we're certain at the start that the solution will not call for buying packaged application software products.

- XP discourages ongoing development of an organization's library of reusable components.

So, what's it good for?

Extreme programming

- Can work well for:
  - single-program applications
  - multiple-program applications with very simple databases
- But what about the *quality* of the end products?

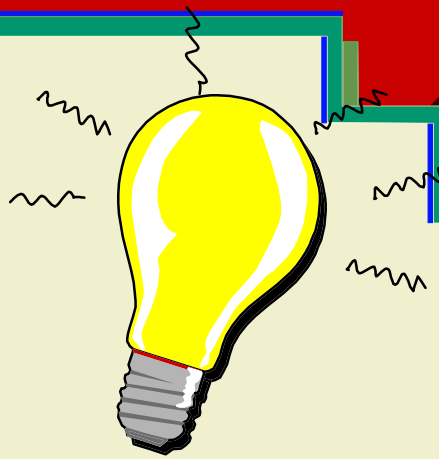
A sample result of XP

```
public class PrimeFactorizer {
 private static int factorIndex;
 private static int[] factorRegister;

public static int[] factor(int multiple) {
  initialize();
  findPrimeFactors(int multiple);
  return factorRegister; }

private static void initialize() {
  factorIndex = 0;
  factorRegister = new int[100]; }

private static void findPrimeFactors(int multiple) {
  for (int factor=2; multiple != 1; factor++)
     for (; multiple % factor) == 0; multiple/=factor)
        factorRegister[factorIndex++] = factor; }
}
```

Where did this come from?

What's wrong with it?

Java: What is it?



- Is Java
  - a fad methodology (MDB),
  - an operating platform,
  - or just a programming language?

Java! Origin

- A simple programming language for *embedded programming* in devices
- Later a simple language for running applets in web browsers
- The emphasis was on ***simple***. An experienced (esp. in C++) programmer could learn it in an afternoon.

Java: Evolution

- Operating platform consisting of;
  - General-purpose programming language
  - Huge families of library components
- With the essential library classes that an application programmer must master, Java is now the ***largest and most complicated*** development tool in the history of programming.

A surprising phenomenon

- Typical Java programs in 2005:
  - Avoid using objects for many application-domain data items.
  - But heavily use objects for internal (housekeeping) artifices.
  - Package non-OO code in pseudo-classes.
- Java, as applied by many practitioners *undermines* object-orientation!

Contributions to new legacy systems

1. from academic institutions
2. from in-house management
3. from fad *breakthrough* methodologies
4. from **contract developer firms**

Contract developers

- 1990s Proliferation of firms specializing in:
  - object-oriented technology
  - UML
  - client-server architecture
  - web-server deployment
  - Java technologies

Many have now gone out of business

- Customers assume the contractors are experts and know what they're doing.

Wrong!

Common contractor failings

- Little or no interest in or understanding of **quality**
- Little or no incentive to facilitate future **maintenance**
- Open *hostility* to **productivity** (esp. by brokers and hourly contractors) *Why?*
- Just deliver something that works (free of "defects") and collect our fee for this contract.

21st Century Expertise

- One can be a world-class expert on UML and know next to nothing about *systems analysis.*
- One can be a world-class expert on Java and know very little about *programming.*
- Being a world-class expert makes some impatient, arrogant, and intolerant of dissent.
- *This is illustrated by some "big name" gurus/authors/consultants*

That was really depressing. Is there a cure?

- The key to avoiding new legacy systems is ***management discipline***. *But that's hard!*
  - Stick to unwavering commitment
  - Reject management by wishful thinking
- The #1 element of methodology infrastructure is ***organizational memory***.

What about **CMM?**

SWEBOK?

How can we avoid creating new legacy systems

- Establish (or resurrect) **infrastructure**:
  - Written standards & processes
  - Processes for proposing, approving, and disseminating them (preferably participative)
  - Quality assurance / review
  - Professional staff continuing education
  - Component library

  Is this a top-down or bottom-up process?

- Demand justification for getting rid or any part of the infrastructure.

Thank you!

- For more information or to continue the discussion:
  - See my web page
    www.idinews.com
  - Send me E-mail
    cweisert@acm.org
  - **Phone me**
    (773) 736-9661

21s century system development