

Legacy Systems of the Past, Present, and Future

Conrad Weisert
Information Disciplines, Inc., Chicago
www.idinews.com

Abstract

We generally associate the designation "legacy system" with the most inflexible and unmaintainable sort of application developed by the most unenlightened programmers for an obsolete mainframe computer. Today, however, despite decades of dramatic breakthroughs in software development methodologies, many organizations are surprised and disappointed to discover that they have replaced those old applications with expensive new ones that are just as costly to maintain. We're still developing legacy applications!

Responsibility for this alarming situation is shared by software vendors, academic programs, fad methodologists, and contract development firms. Fortunately, a remedy is still well within the reach of disciplined management in user organizations.

Background: What is a Legacy System?

Everyone knows that legacy systems are bad. We feel pity for any colleague who gets stuck with maintaining or converting one. We may feel smugly superior to the unenlightened developers of the past who created those nightmares.

Although *legacy system* has no officially accepted definition, we usually infer these characteristics:

- Its user interfaces are unfriendly and error prone.
- It's poorly documented.
- Its programs are disorganized, inflexible, hard to understand, and very expensive to change.
- Its database contains lots of inconsistencies and redundancies.

Creating a Legacy System

We often assume that such an application was designed and originally developed more than a decade ago by a team of developers who lacked knowledge of the tools, techniques, and methodologies that today's professionals take for granted. That assumption, however, is invalid. Growing evidence from large and small organizations shows that many newly developed applications, even those that exploit the latest breakthrough methodologies, exhibit those same characteristics.

Naturally, that comes as a surprise and serious disappointment to the managers who sponsor those projects in a user organization. Not only does the organization fail to get the high-quality system it expected, but also:

- Such projects often unfairly tarnish within the organization the reputation of a new methodology or approach. "We tried object-oriented design," an exasperated manager confides, "and what a fiasco that was!"
- Even if management views the project as an initial success, the cost of maintenance and unreliability over the system's life span will surely have a serious impact upon the organization.

What accounts for the continuing development of legacy systems a quarter century after the structured revolution was supposed to have laid a solid floor under software quality? The blame is shared by at least these three sources:

- Academic institutions, especially Computer Science departments.
- Software-development organizations and contractors, especially their management.
- Dramatic breakthrough methodologies and their zealous promoters.

Let's examine how each of them contributes to the problem.

Academic contributions to legacy systems

Ignoring software quality

Most software developers today have a degree or at least some formal background in computer science or management information systems. Employers criticize academic programs for placing insufficient emphasis on real-world applications, but that's by no means their worst shortcoming. Many institutions, including some of the most prestigious, place little emphasis on software *quality*, often none at all.

Students become accustomed to getting an **A** grade on any program that runs to completion, produces the right answer, and uses the prescribed algorithm. Those students may later be shocked to discover that there's far more to software development. Worse, they may never discover it, pursuing remunerative careers in body-shop contractor firms where their shoddy end products do little short-term damage to their employer's bottom line, but inflict immense long-term harm on their employer's customers.

Some instructors are themselves unaware of essential quality criteria. Students not only report getting astonishingly naïve guidance, but also complain of being penalized by their instructor (more likely by a teaching assistant) for applying long-established *good* practices.

Even those instructors who are well aware of good techniques, confronted with huge enrollments, seldom have time to evaluate each student's work critically. As a result, many students get no feedback on their work except "points off" for deviations from some orthodox solution.

Bypassing good practice

Many academic programs fail to practice what they preach. For example:

- Software engineering courses teach students the great benefits of *reusable components*. Yet few university computer science departments support a central library to which faculty members or others contribute modules for their colleagues or students to use.
- Systems analysis courses teach students that rigorous *requirements specification* is essential to project success. Yet instructors in other computer science courses continue to assign vaguely stated programming problems that violate nearly every good practice for such specifications.

In many institutions tenured professors openly reject cooperation in such areas. Pure computer scientists often view systems analysis with disdain and ignore the *information systems* side of their institution's offerings.

Although many graduates of such programs go on to do high-quality work, others, not surprisingly, join the ranks of mediocre developers using new tools and techniques to produce new legacy software.

Development Organizations' Contributions to Legacy Systems

In many organizations today management's concern is more and more on extremely short-range performance. They avoid making investments that yield payback beyond the current quarter.

That emphasis has led to a dismantling of the information systems infrastructure that was originally intended to foster quality and productivity in system development. Many younger managers lack understanding of the value of such infrastructure, viewing it as bureaucratic red tape. They can easily earn their bosses' approval by eliminating such "overhead" from their budgets, before moving on to another role in the organization. Their successors will pay a big price, but no one in the organization ever associates cause and effect.

Some organizations that had established a sensible date representation standard in the 1970's, for example, ended up with costly Y2K compliance problems after a later manager discarded the earlier standards.

What ever happened to QA?

In the past few years, the term "quality assurance" in recruiting advertisements has evolved to mean little more than "testing" or finding bugs ("defects") in a nearly complete software product. The discredited 1960s cliché "Any program that works is better than one that doesn't" is making an amazing comeback.

Of course, finding and correcting bugs has little to do with the actual quality of software, in terms of its future maintainability and long-term robustness. Indeed, an urgent patch to a discovered operational defect may well undermine the structure of a program or a database, and nudge the software further toward a legacy-like status.

In-house methodology infrastructure

In the 1960s, just about every programming organization had its own set of *programming standards* for techniques to be used or avoided in Cobol or Fortran. Today Java, C++, and Visual Basic (VB) offer the programmer 100 times more choices (ways to go wrong) than Fortran or Cobol did. Nevertheless very few organizations have invested in similar in-house standards for Java, C++, or Visual Basic programming techniques.

The explanation for the surprising indifference to programming standards lies partly in naive expectations about the benefits yielded by newer technologies:

- "Since it's object-oriented, it's surely well organized and reusable."
- "Since it's Java, we can just throw it away and do it over quickly."
- "Since VB generates most of the code automatically, there's just not much need for standards."

The few organizations that have issued standards for programming techniques in a modern language have tended to repeat the mistakes of the earliest naïve data processing managers:

- They emphasize rigid rules and strict uniformity, over flexible guidelines (the "military approach to standards").

- The standards are often based on the personal preferences of one or two in-house gurus.
- They minimize participation by rank-and-file staff in originating or reviewing proposed standards, conventions, and guidelines.

Of course, programming standards are just one important corner of development methodology. An organization that develops (or even specifies) systems draws upon a huge range of processes, standards, conventions, guidelines, and tools. Those methodology components span the subject matter of project planning & control, systems analysis & specification, data definition, data representation, data analysis, program design, programming languages, component libraries, application generators, network architectures, and many other areas. Whether an organization has ten developers or several thousand, it's sure to keep producing poor-quality *legacy* applications in the absence of solid infrastructure for choosing, disseminating, and supporting its development methodology.

Contractors' contributions

The explosion of new methodologies has spawned a corresponding growth in the number of contract development firms claiming to specialize in those methodologies. When your organization engages the world's greatest experts in client-server application architecture to design and develop a client-server application, you ought to feel confident that the software you'll get will reflect the vendor's reputation.

You'll be sorely disappointed, however. Command of the details of some methodology is quite different from expertise in organizing and documenting high-quality software. The contractor's staff may consist of people who have impressive detailed knowledge of some set of facilities but hardly a clue how to put them all together to produce maintainable software.

The recent failures of many such firms are due more to financial overreaching and incompetent marketing than to customers' reaction against their poor-quality products. Assuming that the delivered application initially works, it may take years for a customer to realize that their contractor has delivered a brand-new legacy system.

Methodology Contributions to Legacy Systems

Every year or two, we're treated to yet another major dramatic breakthrough (MDB) in software development methodology. MDBs fall into two categories:

- Many valuable MDBs are introduced as evolutionary insights built upon already established good practices.
- Other MDBs claim to be *revolutionary* changes, demanding a new "mindset" and rejecting certain established practices. Sometimes partisans of a new technique scornfully disparage not only a particular older approach but also its "dinosaur" practitioners.

The latter type of MDB has the effect of both attracting and intimidating managers and professionals. Trade journal articles and conference presentations make us feel that we're hopelessly obsolete and unenlightened if we haven't embraced the MDB. That leads to unthinking instant demand for practitioners, and thus to an instant market for courses and textbooks, which further convey the impression that the world is passing us by.

Some recent MDB's actually return to some of the worst discredited practices of the distant past! Younger developers who have no memory of 1959's user-programmer iterative collaboration or of

1969's "Victorian Novel" requirements specifications are now eagerly embracing remarkably similar "innovations".

Here are three MDBs of the recent past that, along with some positive impact, are directly contributing to the ongoing production of legacy software.

1. *Unstructured Muddling Language?*

Having been blessed by the Object Management Group (OMG), the Unified Modeling Language (UML) has become so widely accepted that many organizations consider it to be a requirement for every project and they demand UML fluency from every job applicant. Uncertainty remains, however, over just what problems UML solves.

The main issue centers on confusion between analysis and design, an old problem that UML makes worse. Although UML provides a rich repertoire of modeling tools for one software developer to communicate to other software developers, it's practically useless for communicating a system specification to the sponsoring end users. Instead UML enthusiasts emphasize two mechanisms for documenting the user's requirements and securing the user representatives' concurrence:

- Use-cases¹
- Unstructured want lists²

Although UML claims to be object-oriented, there's nothing at all object-oriented about either of the above. You can do object-oriented analysis (OOA) without use-cases, and you can prepare use cases without mentioning objects. Apart from superficial notation, use-cases take us back to the discredited sequential narrative technique of the 1960's, called by DeMarco³ the "Victorian Novel approach" to system specification.

A serious practical problem with both use-cases and want lists in particular and UML in general is that there's neither a clear starting point nor a clear end to the analysis (detailed requirements or external specification) phase. The systems analysts just keep writing use-cases and drawing related diagrams until they can't think of any more. Then they hope that the developers can build a coherent application system based on the resulting pile of documentation and that the sponsoring users can make sense of it.

The lack of a definite end to systems analysis is actually cited as an advantage by a sizable community who denigrate formal specifications in favor of trial-and-error incremental iteration.

Although the UML is put forth as an industry standard, it has now become closely associated with one dominant vendor of computer assisted software engineering (CASE) tools. The three pioneering contributors to UML, called "the three amigos" by insiders, have now all gone to work for that company.

The amigos place stress on distinguishing between a language⁴ (UML) and a process (life-cycle methodology), asserting that UML is independent of the choice of life-cycle. They then go on to put forth a companion life-cycle methodology⁵, UP (Unified Process), that further steers a project toward a new legacy system. In particular:

- UP makes no provision for evaluating, selecting, and purchasing packaged application software, even though that alternative has become the mainstream option in many organizations. Consequently sponsoring users often conclude that buying an application software product exempts them from all discipline in defining their requirements.

- There is no phase in UP that produces as its end product a coherent user requirements specification. Indeed, their “elaboration” phase naively mixes analysis and design, two entirely distinct disciplines⁶ requiring distinct skills. That confusion is traditionally one of the most common causes of project failure.

2. XP: help or hindrance?

One of the latest fad methodologies is so-called *extreme programming* (XP), a disciplined form of iterative incremental development.⁷ It came out of the Smalltalk community, and reflects a style and approach that worked well for projects suited to Smalltalk’s strengths and weaknesses.⁸

XP returns to the dominant style of 1950s application development, based on close collaboration between a programmer and a problem-sponsor user, with little if any formal written specification.

Such incremental cooperative development has been shown to work well for producing single-program applications with simple database designs, even where the single program is large and complicated. For more complicated systems, however, XP’s shortcuts lead both to likely loss of project control and to ongoing maintenance nightmares.

An alarming element of XP is the euphemism “refactoring” to describe what we do after we discover that we’ve developed a lot of code based on a faulty design structure. Of course, programmers should always be open to undoing a bad design choice, but when you don’t know where you’re headed at the start, regrettable dead-ends are inevitable.

3. The Java universe

The past half-dozen years have brought us two unforeseen phenomena:

- The rapid rise of Java to become the dominant platform for many kinds of modern application.
- The stunning expansion and proliferation of Java-based technologies, especially for distributed deployment. Java began as a simple programming language, and has evolved into a monstrously complex operating platform.

Recruiting advertisements now demand fluency in EJB, JSP, JINI, JDBC, JFC, and other fragmented technologies we hadn’t heard of in 1995. At the core lies the original badly flawed⁹ Java language that imposes unreasonably high maintenance costs.

Java’s most destructive impact has been a growing distortion of the essence of object-oriented programming, OOP. A novelty 15 years ago, object-orientation has justifiably taken its firm place as today’s mainstream approach to design and programming. Although Java partisans proclaim that Java is a *pure* object-oriented language (“In Java *everything* is an object!”), when we examine a typical Java program we nearly always find:

- Failure to model real-world (application domain) data types as classes. For example, many Java programmers routinely store and manipulate numeric data items as floating-point numbers, just as if they were using Fortran.
- Dozens if not hundreds of quasi-objects representing internal program structures having nothing to do with the application domain. Before OOP we called such objects “housekeeping data”; now the code to manipulate them often overshadows the actual application logic.

- Frequent, if not routine, undermining of OOP discipline through overpermissive *accessor* functions that expose the internal representation of objects.

As a result, most large programs written in Java fail to exploit the benefits of OOP while at the same time paying a huge price in unnecessary complexity.

Those characteristics are exhibited not only in typical programs developed by in-house staff or outside contractors, but also in articles and textbooks from Java experts. Indeed, even the vendor's official Java libraries contain more than their share of misguided design and atrocious code.

MDB Summary

Although we've cited three popular approaches that contribute to producing new legacy systems, UML, XP, and Java all offer many useful and positive features that developers can and should draw from. Each of them rests, more or less, on the object-orientated foundation, which is now firmly established as a mainstream approach to software development.

Note that UML, XP, and Java usually work well for *small* applications. Of course, almost any approach works if the problem is small enough. Very few of the legacy systems that torment us now or will torment us in the future, however, are small.

Recipe for success

Some cynical managers have given up. It's just the nature of software development and maintenance, they concede, to be utterly unpredictable and uncontrollable.

A few aspects of software development are indeed beyond the control of an organization of professional developers. We can bring only slight influence upon academic institutions. We can't predict when the next MDB will arrive or what its impact will be. We feel helpless

Let's firmly reject such defeatism. *Our* organization can be the exception. What it takes is management discipline in applying proven techniques throughout the application development life cycle. Here are a few things we should do:

- We must reinstate (or establish for the first time) supporting infrastructure, including:
 - Written standards and methodology, along with ways of proposing, approving, and disseminating them.
 - Quality assurance / review.
 - Professional staff continuing education.
- Before embracing any new miracle cure or MDB we must assess it critically. Naturally we should draw upon its good ideas. However if we can't see how a new approach contributes to quality or productivity, or if we can't fully understand how it works, then we shouldn't commit to it in its entirety and it shouldn't replace established proven techniques.
- We must apply the same quality standards to custom software developed under contract by an outside vendor that we apply to software developed by our own internal staff. The contract must call for compliance with our standards, and is not complete until the end products have been shown to comply.

- We must be especially wary whenever we're tempted to *discard* existing methodology. Many organizations require return-on-investment analysis for adopting proposed *new* methodology. We should insist upon equally rigorous justification for getting rid of existing standards or processes that are (or should be) serving us well.
- We must respect our professional staff¹⁰ and encourage their participation in proposing and evaluating additions and changes to our organization's standards and methodology.¹¹

Legacy systems, as that pejorative term is commonly understood, never were inevitable. We have to struggle harder and harder to avoid them, but it's a struggle we can win.

1 **Ivar Jacobson**: *The Object Advantage*, 1995, Addison-Wesley, ISBN 0-201-42289-1.

2 **James & Susan Robertson**: *Mastering the Requirements Process*, Addison-Wesley ISBN: 0-201-36046-2.

3 **Tom DeMarco**: *Structured Analysis and System Specification*, 1985 (facsimile reprint), Prentice Hall; ISBN: 0138543801.

4 **Grady Booch, James Rumbaugh, & Ivar Jacobson**: *The Unified Modeling Language User Guide*, 1999, Addison-Wesley, ISBN 0-201-57168-4.

5 **Ivar Jacobson, Grady Booch, & James Rumbaugh**: *The Unified Software Development Process*, 1999, Addison-Wesley, ISBN 0-201-57169-2.

6 **Conrad Weisert**: "Analysis & Design Considered Harmful", www.idinews.com/analysisDesign.html.

7 **Kent Beck**: *Extreme Programming Explained*, 2000, Addison-Wesley, ISBN 0-201-61641-6.

8 **Wilf Lalonde**: *Discovering Smalltalk*, 1994, Benjamin Cummings, ISBN 0-8053-2720-7.

9 **Conrad Weisert**: "The Dark Side of Java", www.idinews.com/darkside.pdf

10 **Conrad Weisert**: "Instilling Professionalism in a Software Development Organization", Proceedings of the 1988 ACM SIGCPR Conference on the Management Of Information Systems Personnel, College Park, pp. 192-197.

11 **Conrad Weisert**: "Methodology Development and Dissemination in a Decentralized Organization", Proceedings for COPE IT '93, Copenhagen, pp. 701-709..