ABSTRACT

Why do software development projects fail, and what can be done about it? This paper, chock full of case studies, identifies eight reasons, or anti-patterns, for unsuccessful software development and software project management. The author walks through what went wrong on numerous projects, and how things could have gone better. From “nontechnical manager leading a team of junior programmers” to “using small system methodology to build a large system” to “the agile charade” to “Rockefeller requirements,” the author combines experience, insight and wit to guide proposal teams, program managers, requirements analysts and technical leads to success.

Keywords: software development, project management, best practices, software engineering

INTRODUCTION

Recognizing that the software design discipline contains stereotypical needs with common solutions, Gamma et al. formalized these into design patterns.¹ Books and articles sprang up documenting anti-patterns—failure modes—in both software development and project management.²

This paper identifies a set of anti-patterns observed by the author over a career in design, development and project management. In thinking about why software development projects fail—either cannot deliver software or cannot deliver customer satisfaction—a handful of patterns dominate the landscape.

Frederick Brooks observed that developing software is like casting a magic spell—it must be done perfectly for it to work. Software is one of the few human activities that demand perfection.³ The fact that these common anti-patterns span management, process and technical issues reinforces the difficulty of software development. For the project to succeed, the whole team must perform superbly, not just the coders!

This article is aimed at proposal teams, program managers, requirements analysts and technical leads in hopes of creating more successes that become referenceable projects leading to yet more successes.

The terms “Client” and “System Owner” refer to a project’s sponsor. “IT” and “the Team” refer to the IT shop, normally organizationally distinct from the sponsor.

All of the case studies actually happened, in some cases more than once.

¹ Gamma, Erich et al. Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley Longman (1995)
² For example, Brown, William et al. AntiPatterns in Project Management, John Wylie & Sons (2000)
ANTI-PATTERN 1: NONTECHNICAL MANAGER LEADING A TEAM OF JUNIOR PROGRAMMERS

This is how the majority of failed projects fail. Here’s how it works.

1. The company bids capable managers and, to remain cost-competitive, a low-cost technical team.

2. The team divides up the work and begins working.

3. When it’s time to integrate, the pieces don’t fit together.

4. Everyone works long hours to make integration happen. But as the software comes together, it becomes increasingly clear that it doesn’t add up to the envisioned business solution.

5. The project fails to deliver, or delivers an incomplete and defect-ridden product.

The deadliest aspect of this anti-pattern is that the team is in over its head from day one, but without senior technical staff, no one suspects that there’s a problem until the team can’t deliver. After all, each developer is capable of doing his or her own piece, and that’s what management hears.

What’s lacking is a custodian of the big picture—a senior architect. Here are some examples of how this plays out in practice.

CASE STUDY 1: THE SCIENTIFIC DATA SUBMISSION PROJECT

The System Owner tasked IT with developing a system that scientists could use to enter research results into a database over the Web. To save everyone’s time, the application would validate both the syntax and the internal consistency of the data before approving the submission.

The System Owner thought that a Java applet made a lot of sense, and the Team agreed. Scientists would view a web page and invoke the applet to validate the data. The applet would display the data, critique it, and submit it after successful validation.

The requirements looked simple to the mid-level Web developers on the Team. Each developer took ownership of a portion of the requirements. The syntax validation requirements required some scientific knowledge to implement, but the team mastered them one by one. What no one realized was that the internal consistency requirements could not be implemented piecemeal by different developers. In fact, taken as a whole, they comprised a classic problem in computer science.

As delivery neared, the Team began sending prototypes to the scientist-users for evaluation. The scientists grew indignant that the Team never seemed to get the consistency validation right. Relations grew so strained that the System Owner demanded that the application undergo full regression testing daily in her presence.

There was another problem. As processing complexity grew, the code volume and CPU demands outgrew what an applet could support. Performance problems became severe, even on data sets of moderate size. Yet data set sizes were growing exponentially as data collection technology improved.

As the ever-growing cost and schedule overrun strained the Client relationship, the Team realized that there was something fundamentally wrong and brought in an architect to consult. The true nature of the technical problem was identified. A suitable algorithm was implemented to solve the problem implicit in internal consistency-related requirements. The Client was persuaded that an applet was too lightweight, and the system was reengineered as a downloadable, stand-alone application. Building on the technical success, management was able to repair the relationship with the System Owner.
CASE STUDY 2: THE USER REGISTRATION SYSTEM

Registering the user is the first step on many Web-based systems and is generally the least of anyone’s worries. User registration is to the Web what payroll is to the mainframe: well-trodden ground, mine swept and quicksand-free.

Registration was all that the User Registration System (URS) system did. User registration data were extracted by the Registrars and used to provision accounts for several other systems. The other systems were outside the scope for URS.

The Team thought they were avoiding Anti-Pattern 1 when they appointed the most senior team member as architect. The problem was that the most senior member on a team of junior programmers was still a junior programmer.

The URS requirements specified that the system be fully configurable by non-technical end users. The architect saw that the relational database model was a poor fit, because tables have fixed definitions—end users can add rows but not columns. So the architect designed a database with two tables: a transactional table and a metadata table. The metadata table contained a row for each cell in the transactional table, to specify what the cell contained.

Schedule pressure forced the system into production before load testing could be done. The system worked. But performance was excruciating, since literally thousands of database transactions were required per business transaction.

The architect lacked the experience to know that:

- Implementing non-relational data models in a relational database management system exacts a heavy price; and,
- If you try to make a system fully configurable by the end user, you must give the user something comparable in complexity to a programming language. Such a requirement is a well-known project-killer that should be fought to the death.

The System Owner was displeased at the delay, and the need to purchase a lot of unplanned hardware. The Team is concerned that future projects may go elsewhere.

CASE STUDY 3: THE WAR/PEACE DECISION SUPPORT SYSTEM

The System Owner gave the Team a large, complex project to support the decision to go to war or not in any given circumstance. Recognizing the importance of the project, IT assigned several senior technical members with great reputations to the Team.

The risk and rewards for project success lay mainly on the head of the Project Manager. He was the only Team member whose job was actually on the line. So the PM considered it reasonable to have the final say on make-or-break decisions. To this end, the PM held a weekly roundtable at which the technical team voiced their opinions about the architecture, and the PM made the final call. The PM found software development more rewarding than project management, if only because the outcomes were marginally more controllable.

The PM’s experience was with small systems, and naturally, he found himself agreeing with developers who had a similar background. The senior technical staff made him feel uneasy because he didn’t always fully understand the terminology they used, and also, their ideas were more complicated and therefore riskier.

After a year, all the senior technical staff had left for other projects, which the PM thought was good a good thing, because the Cost Performance Index was shrinking. The PM believed that the project had been bid top-heavy, as proven by the fact that he was spending too much time in meetings
defending his technical opinions. Two years into the project, the Team was 8 months behind schedule.

The System Owner had seen this before. Figuring that the schedule would continue to slip, forcing a cost increase later on, he cancelled the task.

SUMMARY
Case studies 1-3 ran aground because they implemented Anti-Pattern 1 *No Custodian of the Big Picture* as follows (Table 1):

<table>
<thead>
<tr>
<th>Case Study</th>
<th>Implementation Strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>No Architect</td>
</tr>
<tr>
<td>2</td>
<td>Incompetent Architect</td>
</tr>
<tr>
<td>3</td>
<td>Disempowered Architect</td>
</tr>
</tbody>
</table>

Table 1. Anti-Pattern 1 Implementation Methodology

MORAL
Every software development project needs a full-time senior architect who can:

1. Select the technical stack (or validate the feasibility of a client-mandated stack) and create a Decision Analysis Report, typically a balanced scorecard, to explain and justify the results.

2. Bring or rapidly acquire a true understanding of the business needs that the system will support—as opposed to the typical developer's lawyer-like approach of meeting the letter of the requirements one by one.

3. Create a solution architecture that defines the system as a set of interacting components. Show how the components add up to a business solution by tracing the requirements to component functionality.

4. Make buy-versus-build decisions about the components and create a Decision Analysis Report.

5. Select the design patterns (e.g., Model-View-Controller) to be used for custom components. Work with the System Architect to turn the technical vision into Class and ERD diagrams that the developers will implement.

ANTI-PATTERN 2: USING SMALL-SYSTEM METHODOLOGY TO BUILD A LARGE SYSTEM

In his book *Augustine's Laws*, Norm Augustine, the former CEO of Lockheed Martin, plots the number of U.S. weapons systems over time. The result is a straight line heading down. At some future date, the U.S. will have exactly one weapon, which Augustine calls the Death Star. In the spirit of Augustine's book, let's call such systems *Death Stars*. They're the one system that does it all.

Clients order up Death Stars for the best of reasons. A typical scenario: A new CIO sees that the business is awkward and inefficient and wants to fix it to the extent that budget, technology, and bureaucratic politics will allow. The System Owner had built numerous spot solutions—silo systems. While those looked good at the time, their benefits are now overshadowed by duplicate data entry.

---

data and process inconsistency, inability to see the history of interactions with an individual across all lines of business, and the cost and bother of maintaining many small systems.

The new CIO will tidy all this up into a single system with a cleansed and consolidated database, eliminate paper, fix the usability problems and unsupported requirements that users have been complaining about for years, reduce maintenance cost, and move on to a better job at a bigger-name organization.

Death Stars support many activities associated with a line of business, or even the enterprise as a whole. Such systems enable people to request services over the Web; provide extensive guidance on who is eligible to receive them; scan documents when requests arrive on paper; track the location and disposition of paper records (to comply with archiving policy); coordinate payment data across multiple modalities such as bank deposits, credit cards and payment at cashier windows; prioritize and schedule the workload to service the requests; automate routine decision-making where outcomes are clearly dictated by policy; request information from external systems and progress the case through the workflow as responses trickle in. This is a fraction of the list, but if you’ve worked on such a system, you can fill in the rest from memory.

How do you build a Death Star?

In *The Society of Mind*, Marvin Minsky argues that human intelligence isn’t as complicated as it seems. Complex mental processes are simply orchestrations of simpler processes. The simpler processes can be decomposed, in turn, into lower-level activities. If you take this far enough, you arrive at tasks that are simple enough to be carried out by neurons.

Whether or not Minsky is correct about the human mind, functional decomposition is a great way to tame complexity in computer programs. Functions and subroutines have been with us since the days of assembler. But sometimes we don’t take advantage of the powers of decomposition.

**CASE STUDY 4: THE SWISS ARMY KNIFE**

The System Owner assigned the Team a three-year project to build a system to do everything for everyone. IT selected a system architect/development lead who had successfully built a modest web-based system for the same client, and assembled a team of 15 capable developers and six strong requirements analysts.

The lead prioritized the more than 900 requirements. He assigned the most urgent requirements to the developers, who designed data models and worked with requirements analysts to understand the business logic. Each developer was assigned a requirements analyst, who worked with end users to design intuitive and ergonomic screens.

Before the work began, the Project Manager and Development Lead divided the requirements into batches to be implemented each month over the three-year period of performance, and agreed that the batch sizes were doable.

The Team was supposed to be a well-oiled machine—an assembly line. The raw materials (requirements) entered the line in batches. At the first station, analysts spent a month working with users to clarify the batch of requirements and design some screens. Then the conveyor belt carried the batch over to the developers, who implemented the functionality. The month’s batch of functionality went to the testers, and when the testers were happy, to project management for a client demo.
After a year, only one screen had sufficient functionality behind it to be demonstrable. The System Owner became exhausted and demoralized from seeing it over and over at monthly demos. That wasn’t supposed to happen. What went wrong?

**What Went Wrong**

- **No Domain Expert** – The requirements analysts were smart guys who had worked on similar projects before. But prior projects experience doesn’t guarantee that you really understand the current project’s domain. Moreover, it was hard to get the users to sit down and explain everything about the 900 requirements, or even to discuss screens in detail. Requirements analysis fell further and further behind.

- **Development Silos** – The development lead had organized the system—which was intended to replace dozens of silos—into silos. Each developer created database tables, logic and screens for a few specific requirements in isolation. The development lead was smart enough to allocate the requirements so that developers could work independently—they were working on different parts of the system. But next month’s batch of requirements always reframed last month’s batch in a way that required rework. Of course, the developers had read all the requirements—but didn’t fully understand them until the requirements analysts walked them through the details. And the requirements analysts didn’t have a comprehensive view either—they were working one month ahead of the developers to understand the full implications of the requirements.

Frederick Brooks, who wrote most of what is true and useful about software development, said that every system needs a surgeon. 6 The surgeon has a complete understanding of what needs to be done, and the rest of the team contains the specialists needed to execute the vision.

To be buildable, the system must fit inside someone’s head. The person at the technical helm must comprehensively understand what the system will do, what the components are, and how those components interact to create the solution.

Death Stars are too large to fit in even the smartest surgeon’s head. The system must be decomposed until the components are at a human scale.

*Programming by Contract* decomposes a system into components with well-defined interfaces (the contracts). Begin with a solution architecture—a high-level diagram that identifies custom and COTS components, and shows how they interact to create the solution. The solution architect should understand the system and how it works in totality, at the top level.

Component internals may be too complex to fit in a system architect’s head. Create a decomposed architecture diagram for each component, showing the sub-components and their interactions, and demonstrating that these interactions support all the interactions that must occur across the component’s boundary. Repeat as required.

The result should be a portfolio of mid-level architecture diagrams—below the level of “this is the application server and this is the database” but above the class level. The interactions among the components should be described in narrative text or pseudo code—sequence diagrams are very labor-intensive and can come later.

Components interact by well-defined interfaces—the boundaries are tight, components cannot reach into each other’s internals. The interfaces can be object-oriented, C-style APIs, web service definitions...any paradigm that the market supports. At this stage, requirements are allocated to use cases, and use cases are traced to message orchestrations across the components.

---

6 Ibid.
Organize the project into small teams of surgeons and assistants (senior and mid- or entry-level developers). Give each team a set of components to build. Give them complete autonomy over the implementation, as long as they adhere to the letter of the law concerning the interface.

Create a test harness in which the Test Team can exercise the components—calling the API, web services or whatever—in isolation from the test of the system, using a suite of unit tests.

This approach will yield:

- Improved quality because testing is done earlier
- Fewer integration failures because the system is well-understood at all levels of detail
- Terrific productivity. Nothing motivates developers like autonomy. Not even stock options.

MORAL
To build a Death Star:

1. The standard model has the requirements team learning from sponsors, users and other stakeholders. Cheat and hire a domain expert. Hire at least one for each and every functional area in the system. Get them early because they can help you size the effort.

2. This project acted as if complexity didn’t exist. If you organize the project into requirement/screen/code/data silos, better be sure that cross-silo interactions do not exist. Don’t sweep the complexity under the rug; it doesn’t stay there.


ANTI-PATTERN 3: THE AGILE CHARADE

The Iron Triangle’s vertices are cost, schedule and quality. One can control at most two of the three. (For software, the triangle is even more constricting—accelerating the schedule while maintaining quality by staffing up (adding cost) runs straight into the mythical man-month.

Many software contracts have an unstated clause regarding the Iron Triangle, although neither the System Owner nor IT will admit it. The clause reads, “We will deliver the functionality you want within cost. But since we’re all adults in the room and this is software, we know going in that the schedule will slip.”

The canny Program Manager stays awake nights when the client values on-time delivery over cost. Estimating the time to build software is not a science…or an art, or even black magic. It is guesswork at best, function points notwithstanding. This is the Perry Mason Principle of software estimation. Just as in a Perry Mason novel, the guilty party is the least suspicious person, even though you’ve already suspected the least suspicious person, so software always takes longer, even though you’ve planned for it taking longer than you think.

Agile development methodology stands the triangle on its head. There’s a new clause in the contract, and it’s explicit this time: “We will deliver working software at fixed time intervals at fixed cost. We will allow you to change and reprioritize requirements however you want. Consequently, we are unable to promise in advance what’s going into each release.”

This is anathema to people who monitor projects. First, Agile allows uncontrolled feature creep. The monitor knows that feature creep has killed fighter jets and air traffic control upgrades, and can easily kill this project too. IT usually has an incentive to push back against feature creep. Agile removes it. Also, it defeats the standard ways of measuring progress such as Earned Value
Management, which rely on knowing the percentage of work complete. It's hard to know the percentage of work complete when the scope is regularly revised as a best practice.

Agile is not all roses for IT, either. At first glance, Agile looks like a way to transfer risk to the Client. The catch is that every release must be high-quality—that’s a key incentive for the Client to agree to Agile.

Agile projects achieve quality rigor through continuous integration and automated regression testing. Every time a developer checks in a code change, the entire project is built and comprehensively tested. Any build or test failures are emailed to the developers for immediate action.

Modern tooling makes continuous integration simple. But the crucial ingredient—automated regression testing—rarely happens. (The author has seen it only in commercial product development and when someone’s life is on the line—e.g., manned space missions.) Automated regression tests are excruciatingly labor-intensive to create, they need to be redone whenever the UI changes in any fundamental way, and the skillsets for Rational Functional Tester or Quality Center are hard to find.

Here’s how this plays out:

1. The project commits to using automated regression testing.
2. The project purchases the test tool.
3. It takes months to hire or train someone to use the tool. A backlog of unimplemented test cases builds up.
4. Test scripts are implemented. By this time, there is enough UI for users to review, and they want significant changes. Half of the completed test scripts are now useless. The backlog swells.
5. The Program Manager realizes that it would take four testers to catch up. But it took months to hire the first one, and there’s no budget for that anyway.
6. The system is ready for delivery and the regression tests cover at most 10% of the functionality. The PM decides to claim credit for regression testing being done and move on.

Complexity makes Agile harder. Want to implement a new user story for the next release? In a simple system, one developer might implement a couple of new database tables, create a new screen, and write the business logic for data validation, storage and retrieval.

In a complex system, it’s not that linear. Data isn’t pipelined from database to UI in a closed path. New features have footprints all over the code. Agile encourages developers to do it quick and dirty and get the work done—elegance is optional. It can come later if at all.

Agile Charade is a benign anti-pattern. The System Owner and IT pay lip service. Use cases are called user stories. There are daily stand-ups at which progress is reported and tasks are shuffled. The System Owners are usually not invited, lest they try to change the requirements at no additional cost and without slipping any mandated delivery dates. There’s not much substance, but everyone loves to use Agile terminology.

ANTI-PATTERN 4: DATA LAST

This seductive anti-pattern creates systems that are glowing successes in every way but meeting business needs. To implement, start with a set of siloed applications. Consolidate the business processing and integrate the screens under portal. Put off data cleansing and consolidation until last.
CASE STUDY 5: THE FREELANDIA IMMIGRATION SYSTEM

Like many countries, Freelandia has several visa processing systems for adjudicating different types of visa requests, a border crossing system that records at least some of the entries and exits, and enforcement systems around detaining and deporting immigration violators.

The only interface among these systems was human. People in custody for overstaying their visa could be granted a bridge visa and then deported, because the only way Enforcement could know that the visa had been granted was to repeatedly log onto the visa system and check. The Enforcement officers were busy people with other things to do.

Freelandia tasked IT to fix the problem. IT built a great-looking web portal that seamlessly integrated the existing user interfaces. Immigration could not have been happier. They showed the portals to their employees, the Government, and the press. The Government held them up as exemplars of citizen service improvement and successful IT project management.

The system deployed. People kept getting deported in error. The problem, of course, was the portal gave the illusion of integration, while the data were still in silos. The Portal had no way of knowing that John A. Smith born on April 20, 1988 in the visa system was the John Smith born on January 1, 2000 in the enforcement system (the latter date was a placeholder for missing data).

IT decided it was time to integrate the data. They brought in a record matching product (aka master data manager, customer master file manager, et al.) to clean and consolidate the data. Like all others, the chosen product didn’t know how to integrate person data reliably out of the box—you have to tweak it for your mix of languages, reliability (verbal vs. documented information), etc. That took months and was an iffy proposition in the end—you tweak the parameters, look at the resulting matches—and how do you know if the matches are right or not?

Eventually IT and Immigration were satisfied with the (subjective) match quality and began data consolidation. Consolidating requires comparing every immigration record in Freelandia with every other immigration record, to see if they might refer to the same person. Even if the records have a visa number or other key, you still have to do the comparison because people and OCR devices don’t always get the number right.

This is when the realization set in that a billion squared is a trillion, and that the processing time could well exceed the Mean Time Between Failure of the processors. But it had to be done. IT set up a server farm (despite the gasp-inducing per-processor software licensing cost) and started cranking away. The server farm made inroads into the trillions of comparisons, and the processor heat kept IT personnel warm during the frigid Freelandia winters.

Even during the winter, though, many people visit Freelandia. The record volume grew at a linear rate over time, and the number of comparisons grew in polynomial time. IT never caught up, so the consolidated data were never deployed.

The press pilloried IT. The Government held up Immigration as an example of bad project management and cut their IT budget by 10%.

MORAL
Data first.

ANTI-PATTERN 5: JUST GET IT RIGHT THE FIRST TIME

A programmer sitting at a terminal writing code looks just like a bureaucrat sitting at a terminal writing policy. This accidental similarity causes untold mischief. System Owners, many of whom are bureaucrats, believe (and why should they not?) that writing instructions for a computer is much like
writing instructions for people. You just sit down and do it. And if someone would please proofread the document, it’s good to go.

This leads System Owners greatly to underestimate the complexity of software and the time required to create it. The false analogy also makes System Owners believe that software can be made close to perfect the first time, and can be perfected by one or two rounds of editing.

CASE STUDY 6: THE MILITARY MISSION EXPERT SYSTEM
The expert system was built on expensive LISP machines, before Moore’s Law made specialized symbolic processors obsolete. Mission Command provided a development suite at the remote IT site, and a demo/deployment site at headquarters. But the System Owner viewed a dedicated test suite as a budget buster. Their instructions to IT were, “Just do it right the first time. We in the military don’t get second chances, either.”

The Team didn’t always get it right the first time. Some of the demos embarrassed the System Owner in front of the Brass—couldn’t the Sponsor get it right? Didn’t they realize that lives were at stake?

The Military Mission Expert System was completed but never deployed due to the general perception that the benefits were not worth the cost, and the results would have to be hand-checked anyway.

MORAL
Just Get It Right the First Time may not be realistic, but it happens. When the System Owner refuses to provide the schedule or funding required for test, IT may have little recourse. Often, the root cause is a “hard” constraint imposed on the Owner by outside forces. IT will have identified this as a big risk in its Program Management Plan, but both System Owner and IT would rather hope for success than not attempt the project.

ANTI-PATTERN 6: ROCKEFELLER REQUIREMENTS
Someone once asked John D. Rockefeller how much money it takes to make a man happy. Rockefeller said, “More.” Rockefeller Requirements exist to impress. The system’s actual purpose is to advance the System Owner’s career, and the Owner has only a hazy idea of what it will take to do that. So when IT tries to obtain closure on the requirements, the Owner looks doubtful and says, “More!”

CASE STUDY 7: THE ACQUISITION MANAGEMENT SYSTEM PROJECT
As a project manager, Bertrand was aware how complex an endeavor it is to manage large acquisitions. He assembled a number of horror stories into a briefing to leverage funding for a support system that would walk inexperienced program managers through the steps. He wasn’t quite sure, himself, what the steps were, but knew that millions could be saved if they were followed.

IT hired an experienced acquisitions person as Program Manager. The Program Manager wasn’t convinced there was a recipe for a successful acquisition, but he was more than willing to implement whatever Bertrand and his Program Management Office had in mind.

Bertrand grew nervous when it became clear that the Program Manager didn’t have a magic recipe, either. The Team fleshed out the very generic requirements as best it could. The Team did a great job building screen after screen for scheduling, budget tracking and so on, and making them functional. But to Bertrand, the screens looked prosaic, and nothing like the amazing solution that he’d pitched. So he refused to sign the requirements specification, saying “More!”

This went on for five years. The Project Monitors began to ask questions and to press for the project to be terminated. Bertrand continued to oversell his system and found clever ways to keep it alive.

The Project Monitors found out about the clever ways, and called the Inspector General, who cancelled the program and forced Bertrand into early retirement.

MORAL

*Rockefeller Requirements* are the time for *caveat vendor*. It would be a wonderful thing if IT could protect the System Owners from themselves by refusing projects that are doomed to fail. When IT sees a requirements Ponzi scheme in play, the best strategy is to minimize the collateral damage:

- Ensure that execution is immaculate from a legal and contractual point of view.
- Don’t let IT become the public face of the oversell. Make Bertrand give his own briefings.

**ANTI-PATTERN 7: BRICKS WITHOUT CLAY**

“Data!” said Sherlock Holmes. “I can’t make bricks without clay!” The *Bricks without Clay* anti-pattern occurs when the System Owner attempts to solve a political or policy problem with technology.

**CASE STUDY 8: THE HUMAN RESOURCES ANTI-DISCRIMINATION SYSTEM**

A political hot potato, the system never stayed off Senior Management’s radar for long. Half of the organization wanted this system to do more than privacy laws allowed. The other half passionately wanted it to fail and go away.

As a result, HR got weekly phone calls from Senior Managers on both sides of the debate, demanding data about how the system was being used. Pro-hiring managers demanded data to prove that all candidates were being evaluated fairly. Anti-hiring managers wanted to prove that the system was rife with fraud.

HR relayed these requests to IT in the form of phone calls beginning with, “Quick! The Twelfth Floor wants answers!” But questions such as “What percentage of eligible candidates are applying?” and “How many of the resumes are fraudulent” can’t be answered from the system’s data. So IT would create ad-hoc queries to address questions that were answerable, write up the answers and all the caveats, and send them off.

Eventually HR wanted to run their own queries and tasked IT with building an automated fraud detection system. The problem was that due to privacy concerns, the system was prohibited from collecting the data required to detect fraud in any meaningful way. Due to privacy constraints, none of the common fraud methods left a data trail.

IT hired a fraud consultant, who developed a rather weak set of fraud indicators—about a dozen automated queries that identified cases that could be a starting point for a fraud investigation, but stopped far short of proving fraud.

Realizing that smart, aggressive Senior Managers would rip the results to shreds, HR began calling the Program Manager daily to ask, “Is that the best you can do?” The Program Manager would say, “You’re the experts, you tell me!” and HR would reply, “But that’s why we hired you!”
The System is currently in production. It’s used to develop leads for investigation, but is no help at all for its original purpose, to answer inquiries from the Twelfth Floor.

MORAL
This anti-pattern dooms the project that instantiates it. IT’s options are three:

1. Refuse the tasking. This is rarely an option.

2. Educate the Client about why the stated requirements are impossible to satisfy. (The Client is likely to respond, as HR did, that it doesn’t matter that it’s impossible because it’s politically imperative. Bureaucracies are comfortable with such situations.)

3. Re-purpose the system. Get the Client excited about what the system actually can do. (In the social sciences, this is called co-optation.) Co-opting the Client is really just an exercise in damage control usually, but it’s the best that IT can do.

ANTI-PATTERN 8: SYSTEM DEVELOPMENT AS WAR BY OTHER MEANS

The final anti-pattern, *System Development as War*, can occur when the Client is a mature bureaucracy in which internal goals have long supplanted the organization’s original mission. The system becomes a political football in a scrimmage between opposing factions.

CASE STUDY 9: THE CASE MANAGEMENT SYSTEM

The Department of Dysfunction,§ notoriously one of the least functional parts of the organization, tasked IT to build a case management system. The System Owner was the department’s CIO. IT knew the Department would be a difficult Client, but didn’t know that:

- The Quality Assurance Organization had the final say on whether a system was suitable for deployment, and intended to use this project to make the point to the Department of Dysfunction.
- The System Owner had worked in the IT organization many years ago and left under disagreeable circumstances.
- The IT Program Manager was a woman, and the Deputy System Owner had a deeply held cultural and religious belief that women should not be in the workforce.
- The Department’s CIO has previously tried to implement this system using an external contractor. It had been a spectacular failure. The CIO had no intention of letting IT succeed where he had failed.

The CIO, Deputy CIO, and Quality Assurance Organization did whatever they could to make IT’s life miserable, and to make their own project fail. IT appealed to the Deputy CIO, who responded that the system did not meet its functional requirements, despite the fact that none had ever been specified. IT tried putting functional requirements into successive versions of the requirements specification, and was rebuffed for usurping the System Owner’s prerogative.

By the time this Greek tragedy had run its clockwork course, several of IT’s staff were banned from entering the Department’s offices, a senior manager had been forced out of IT, and the Department had demanded nearly a million dollars in recompense.

---

§ It is accidental that the acronym is DoD. It wasn’t.
The Department terminated the task and gave it to a different external contractor. Two weeks later the Department of Dysfunction announced that remediation was successful, the Department had pulled off a miracle, and everyone involved was praised and rewarded.

Be of good cheer. In a bureaucracy this Hobbesian, one’s professional life is nasty, brutish and short. No one lasts very long. The CIO, Deputy CIO, and Director of Quality Assurance all moved on. With no organizational memory left, IT now had a happy relationship with the Department of Dysfunction.

AFTERWORD

All the projects in the case studies had skilled technical staff. Half of them had internal risk management programs. Three were headed by Program Managers with Program Management Professional (PMP) certifications. Two were certified as CMMI-3.

If conclusions can be drawn from the anti-patterns as a whole, they are these:

- Good people and good processes are necessary but not sufficient for success.
- You can’t ignore the human element. People use their professional life to express their personal agendas. This is not necessarily pernicious. In the best case, the agenda reads, “I’m going to go off like a supernova and be a star in my discipline.” Regardless of what the agendas are, the canny Project Manager is aware of staff and Client agendas and manages accordingly.
- Just like the military, IT wants to refight the last war and get it right this time. Some of the anti-patterns occurred because technical staff or management set out to apply lessons learned from their last engagement, whether they were relevant or not (e.g., applying best-practice small-system architectures and management techniques to large systems).
- In a Perfect Project with well-defined, well-understood requirements, ample schedule and budget, cooperative Client, superb staff and savvy management, two ineluctable difficulties remain, because they are inherent in software design and development.
  a. Requirement statements and use cases are written for human consumption (often by domain experts only). Like any natural language artifact, they are rife with unstated assumptions and information that can only be known from context. The ability meaningfully to turn the open-endedness and implicit complexity of requirements into ambiguity-free code is the rarest and most valuable talent in system integration.
  b. To ensure that the system coheres, everyone involved in design and development must continuously talk to everyone else. The number of communication paths scales as the square of the team size. One of the most effective ways to deal with this is to put everyone in a bullpen where all conversations can be overheard. It’s amazing what you learn that way.

Finally, as Machiavelli observed, sweat, prudence and virtue are half the game. The other half can come down to sheer luck. It’s considered unseemly to say this, since it sounds like whining, but management, technical staff, and even the Client have to play the cards they’re given.

Would you want all outcomes controllable? Do you still play Tic-Tac-Toe?

---

ABOUT THE AUTHOR

Rin Saunders joined CSC in 1990 and has worked on projects that apply pattern recognition, machine learning and quantitative methods to fields ranging from intelligence analysis to medical diagnosis. He received a B.A. from the University of California, Irvine, a Ph.D. from Princeton, an Innovative Applications award from the American Association for Artificial Intelligence, and a Technical Excellence award from CSC. rsaunder@csc.com
About CSC
The mission of CSC is to be a global leader in providing technology-enabled business solutions and services.

With the broadest range of capabilities, CSC offers clients the solutions they need to manage complexity, focus on core businesses, collaborate with partners and clients and improve operations.

CSC makes a special point of understanding its clients and provides experts with real-world experience to work with them. CSC leads with an informed point of view while still offering client choice.

For more than 50 years, clients in industries and governments worldwide have trusted CSC with their business process and information systems outsourcing, systems integration and consulting needs.

The company trades on the New York Stock Exchange under the symbol "CSC."

DISCLAIMER
The information, views and opinions expressed in this paper constitute solely the author’s views and opinions and do not represent in any way CSC’s official corporate views and opinions. The author has made every attempt to ensure that the information contained in this paper has been obtained from reliable sources. CSC is not responsible for any errors or omissions or for the results obtained from the use of this information. All information in this paper is provided “as is,” with no guarantee by CSC of completeness, accuracy, timeliness or the results obtained from the use of this information, and without warranty of any kind, express or implied, including but not limited to warranties of performance, merchantability and fitness for a particular purpose. In no event will CSC, its related partnerships or corporations, or the partners, agents or employees thereof be liable to you or anyone else for any decision made or action taken in reliance on the information in this paper or for any consequential, special or similar damages, even if advised of the possibility of such damages.