Building Software In-House: Too Much Control and Flexibility

Practicum

Ivan Ruchkin
Institute for Software Research
Carnegie Mellon University

May 9, 2012

Abstract

As domain-specific software becomes more available, businesses face a dilemma: whether to acquire commercial off-the-shelf (COTS) enterprise management systems or to build them in-house. Companies choosing to create a product internally are often rewarded with flexibility and control over their development process and its results. However, when expanding, they can outgrow their ability to support the developed software.

Working as a programmer at a medium-sized logistics company, Si-Trans, in 2010, I witnessed the long-term implications of an initial decision to build an information system in-house. While this decision was appropriate in 1997 because COTS alternatives were scarce and inapplicable, it created a favorable climate for inconsistent, ad hoc management practices within the entire company, in particular, software creation and maintenance. These practices ultimately contributed to Si-Trans' inability to see an opportunity in the early 2000s when it was feasible and advantageous to adopt a COTS-based solution. This solution would have scaled better for the company's growth and would have helped avoid an outstanding technical debt in the old system. By the end of 2011, Si-Trans finally considered the acquisition of an off-the-shelf information system, after having suffered substantial financial losses from the protracted in-house development.

Contents

1 Si-Trans: Context and Business

With the advent of powerful and flexible business-oriented enterprise management systems, companies that do not possess considerable software engineering expertise got a viable and often practical alternative of acquiring commercial-off-the-shelf (COTS) systems instead of building in-house ones [17]. Commercial acquisition promises lower costs of ownership and lower development-associated risks at the price of losing control over what is developed [12]. At the same time, in-house development allows an organization to have total control over the created software, but it comes with typically higher costs and a risk of outgrowing the organization's processes [6].

My experience is related to a company, *Si-Trans*, that experienced the hardships of in-house development. Having developed an internal information system over a period of 15 years, Si-Trans started out from reaping a competitive advantage from the system and ended up almost incapable of sustaining the development. In 2010–2011, I worked in the company as a programmer and closely observed the implications of building an internal system for a long timespan, in which circumstances and requirements had changed. After having lost money on fighting through the old system's technical debt [1], the company decided to transition to a COTS alternative in late 2011.

This practicum goes through the chronological story of Si-Trans and evaluates its management's strategic decisions in software development and procurement. Common understanding of the "buy versus build" issue, as it is expressed in software engineering literature, serves as a basis of the evaluation and reflection.

Before the narration dives into details, it is crucial to know the business model and organizational structure of Si-Trans, as these were major factors that determined successes and failures of the in-house development. The rest of this section is devoted to the description of the Si-Trans business.

1.1 The Business Model

Si-Trans is an international logistics company established in 1995–1996 in Moscow, Russia[1]. It rose from the "primordial soup" of companies in the 1990s Russia when the idea of individual capital was spreading among the wide masses of Russian people. The founders of Si-Trans realized that they could exploit Russia's geographical position and the global economic situation of that time. Specifically, they took advantage of Russia's intermediate location between China, where tons of goods were produced, and Europe, where tons of goods were consumed. To earn money from connecting this demand and supply, Si-Trans started transporting cargo between China and Europe. As the company grew, it developed expertise in related areas, such as insurance and customs services. By 2010, Si-Trans provided the following services:

- *Multimodal*[2] *transportation of goods:* businesses in Western Russia or Europe ordered a run of goods from a warehouse in China (e.g., in Hong-Kong, Beijing or Shanghai) to their location in Europe (e.g., in Germany, Romania, or Italy)[3]. Then a Si-Trans employee analyzed modes and schedule to suit the client's needs. After a contract was signed, a payment was processed

[1] Formally, the legal entity of Si-Trans was established several years later, aggregating a few smaller companies. But for simplicity we will refer to them as Si-Trans as well.

[2] *Multimodal* transport involves different carriers, or *modes*; for example, plane, train, truck, or ship.

[3] Alternatively, certain goods might go from Europe to Russia. However, for this practicum we assume that all transports go from China to Russia and Europe.

accordingly, and the goods were shipped from China. The usual obligations of Si-Trans were finished when the goods were delivered to a warehouse of the client's choice.

- *Customs brokerage:* as Si-Trans carried goods over borders of several countries, it gained skill in customs operations: handling documentation, estimating costs and schedules, and negotiating time constraints with government agencies. Si-Trans offered these services to other companies that would otherwise have to struggle with custom clearances themselves.

- *Cargo insurance:* insuring cargo that travels through several countries was and is not trivial because of local differences in legislation and common practice. After Si-Trans had built up related knowledge and connections with local insurance companies, it became possible to sell insurance services that help other companies avoid an overhead from handling individual country-based insurance agencies.

- *Storage:* Si-Trans owned and rented warehouses in China and Russia. Since extra storage space was usually available, it was possible to make money by subletting it to other companies. Si-Trans offered it as a separate service as well.

Si-Trans was built around the core service of international logistics, so geographical distribution of operations was inherent to the company from its very beginning in around 1995. The 2010 distribution of offices is shown in Figure 1. Although I do not have precise data about how Si-Trans was spread over countries before 2010, anecdotal evidence suggests that it was distributed among countries from the beginning.

Figure 1: Si-Trans' locations and transportation examples as in 2010.

1.2 The Organizational Structure

The organizational structure of Si-Trans persisted over the years: regional divisions in Europe, China, and Russia reported to top management. Several independent units, such as accounting and finance, also reported directly to management. Figure 2 shows the structure of Si-Trans as of 2010. Unfortunately, I have no accurate information about what the structure of Si-Trans was before the early 2000s, but the source code and the database design I worked with assumed the same structure as in Figure 2.

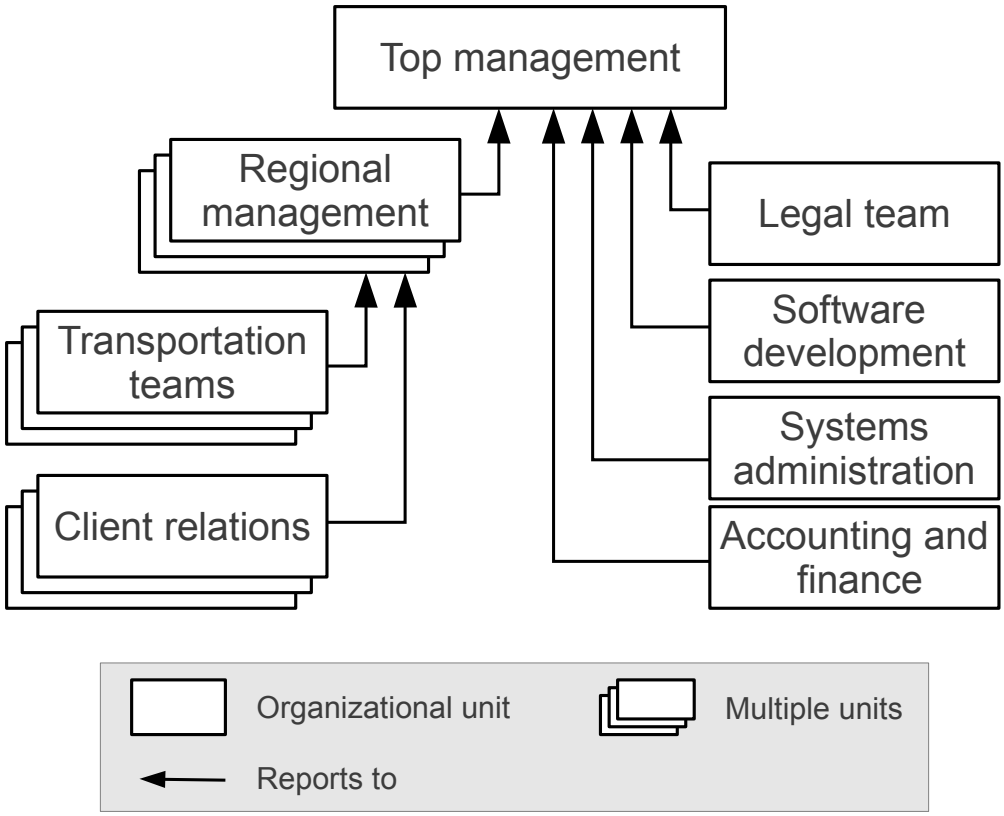

Figure 2: The organizational structure of Si-Trans in 2010.

Top management was located in the central headquarters in Moscow—they were several non-technical executives making strategic decisions. Legal and systems administration teams dealt with issues in all locations, and they were able to succeed because of their low workload.

Regional divisions handled everyday issues related to the main service—logistics. Transportation staff made sure goods passed customs, traveled in acceptable conditions, and went into the right hands. Client relations staff handled all communication with clients, analyzed transportation tradeoffs, and planned routes.

2 The Buy vs. Build Issue

This section describes the conventional engineering wisdom on the topic of buying or building software for business.

2.1 Approaches

Over the last decade, researchers and practitioners accumulated plenty of guidance on whether to buy software or to develop it internally [13]. We focus on the case when a company's business goals are best accomplished with software, but the company does not plan to take advantage of selling the software. The general approaches with respect to buying or building are as follows [4, 14]:

- *Build:* create software *organically*—with a staff team of engineers. This is a fully internal option: the company performs all development phases itself, from analysis to deployment. This direction will also be called building software in-house.
- *Buy:* completely adopt an externally developed system. Although financial details might differ[4], in the scope of this practicum we will call this approach COTS acquisition or buying the software[5]. It is essential to perform search, analysis, and selection for this strategy.
- *Outsource:* have a 3rd party develop software. In this case, a company creates a specification for the needed software and goes though a process of finding a contractor who would implement and potentially maintain the software.
- *Hybrid:* combine the above techniques at different stages. For example, a company may design an architecture and split the desired system into components. Afterwards, the company may acquire available components, develop the crucial ones in-house, and outsource the rest.

Clearly, there is a spectrum of choices that differ in how much software the company produces itself. Acquiring COTS for all needs and building everything internally are at the opposite ends of this spectrum. Choosing an exact point is a strategic decision for the company: it determines the company's competitive advantage, expenses, and flexibility of other choices for many years into the future [11, 10].

2.2 Making Decision

The literature on how to choose between buy and build offers guidance on three aspects of this decision:

- What the common benefits and pitfalls of building and buying are [12, 17].

  Building software in-house promises more flexible development, easier change of goals, and adaptation for an environment. However, building is generally considered to take more time and more resources to deliver a product, which might be not sufficiently tested or interoperable with other products on the market. Buying is believed to be less flexible because of a potential mismatch between an acquired system and the company's needs, but it is also considered to have lower costs of ownership in a long term, shorter deployment schedule, and generally more reliable software.

- How fitness of a company/product pair for either buying or building should be estimated [15].

  The commonly known factors to consider are following:

[4]For example, a company might acquire an open source solution, which would not formally be commercial.

[5]We do so because many aspects of adopting open source and commercial software are similar if not identical. The difference in initial cost of adoption is the biggest difference between adopting COTS and open source.

- *Required time to market* [6]: if a system is needed in the short term, buying looks more attractive because typically customizing and deploying a ready system takes less time than developing one from scratch.
- *Presence of internal expertise* [17]: a company that wants to build the system should have or hire staff qualified enough to do so. It is not only about technical expertise, but also about domain knowledge and management capabilities: adequate requirements have to be elicited and put down as a specification, and software should be assembled according to the specification.
- *Availability of solutions* [9]: COTS products in the market have to cover at least the most critical requirements to be considered. The more of them are available and the longer they have occupied the market niche, the bigger benefits from buying are. A higher degree of competition in the niche is considered a sign of more mature solutions provided.
- *Risks of building vs. risks of buying* [3]: failing to develop a working solution in-house is evaluated against over-relying on another company. Some sources state that in-house development is more predictable than relying on another company because all internal data is available. For example, a rate of development progress, staff working on a system, and the total cost of development are known when creating an organic system. At the same time, this data may be unknown about a company that provides COTS. Hence, that other company's failure is harder to anticipate and accommodate.
- *Stability of requirements* [2]: if requirements for the system are expected to change and are not likely to be covered by available solutions in the market, building internally might be a better solution. In the organic case it is at least theoretically possible to evolve a system for the new needs, while many commercial solutions cannot be changed sufficiently.
- *Presence of standards* [12]: the more a subject domain is standardized, the less risky buying is. When standards are present, emerging software is more likely to adhere to the standards and interoperate with the acquired system. Also, standards are known to be a suitable basis to create a hybrid solution on.

- How each potential option affects a company [7].

  To build software, a company may have to go through an *organizational change*: it may need to form requirements engineering, programming, and testing teams and provide for their communication with other units. Buying software may lock in certain processes and communication paths; therefore, the organization might change to follow the processes ingrained in the acquired software [12]. This might pose a risk for the company's business goals because the ingrained processes might enforce not an optimal way for the company to be organized.

  Apart from a potential organizational change, acquiring 3rd party software results in the *dilution of control* [17]: many companies contribute to the success of a system's exploitation, so the control pattern becomes complicated. Since vendors make ongoing decisions, effort is required to keep the system in adequate operation. An apparent impact on the software procurer is the reduction of his control over the evolving software. This impact is often considered negative because the company needs to account for someone else's interests and actions when planning its own strategy.

3 Initial Decision to Build In-House

During its first years, Si-Trans needed an information system to share information between its distributed branches. In 1996, a decision to build the system organically was made, as it is shown in Figure 3. This section relates this decision to the global environment in which Si-Trans operated.

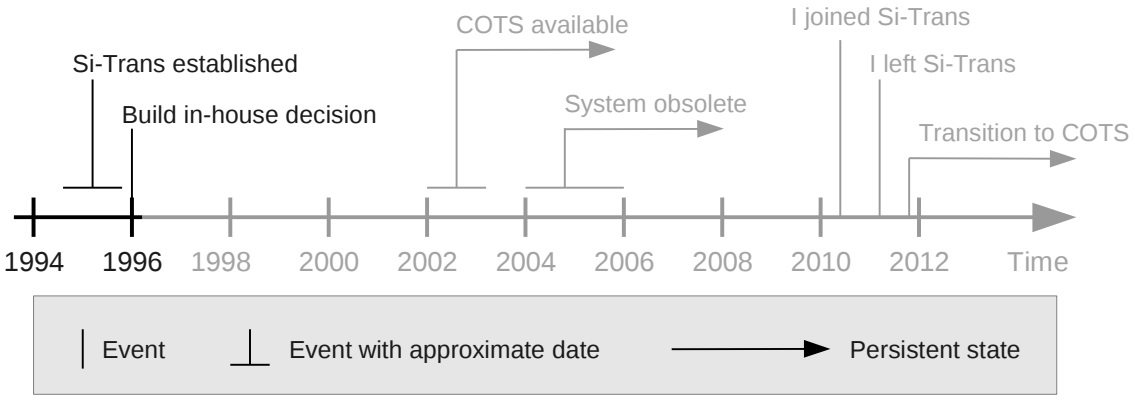

Figure 3: The timeline: 1995-1996. The early days of Si-Trans.

3.1 External Factors in 1990s

When communism was officially dismantled in 1991, Russia faced a serious challenge of adopting a market economy. Not being culturally and organizationally ready for a sharp change in this direction, Russia had to go through a period of economic turmoil that lasted until the early 2000s. The country saw vast corruption, hyperinflation, decline of whole industries, problematic property privatization that gave rise to super-rich oligarchs, and national currency devaluation. In addition to its economic trouble, Russia went through several political crises. Not going too deeply into details, tension between the Executive and Legislative branches shook the whole government apparatus. The Judicial branch lagged behind the fast changes of the two other branches. Hence, a serious disconnect between the letter of the law and the actual way it was administered was typical of the 1990s.

This economic and political instability had strong ripple effects on individual businesses. It affected two major aspects of Si-Trans' business routine. First, frequent changes in legislation and regulations altered, sometimes unpredictably, how everyday business issues such as taxes, certifications, and customs clearances were handled. Second, the transportation market was not stable: turnover of companies was very high, often due to illegal action from their competitors, e.g., corporate raids or arranged criminal cases. It was impossible to predict which of Si-Trans' clients would stay or would go the next month.

The turmoil in Russia did not prevent Si-Trans from looking into business optimization. And so the idea of an information sharing system for remote collaboration was conceived.

3.2 Need for Information System

Due to its geographic distribution, Si-Trans needed a means of coordination between teams. Use of information processing software promised a huge leverage over telephone communication and

emailing. There was also a potential to automate routine processes, such as report generation. Moreover, a system might store operations data for future reference and provide analysis and planning tools.

The quality attributes of concern were, presumably, stability—uninterrupted operation, security—data confidentiality and integrity, and performance—supporting an interactive mode for operators. The level of these attributes that Si-Trans needed was, as it often happens, "just good enough". The company just needed a system that could help with running operations and not hurt the business.

An important requirement was that the system adhere to Russian practices and regulations: personnel management, customs forms and declarations, service contracts with other companies, tax reports, handling finances, and so on. Also, Si-Trans could not afford to hire regular staff fluent in English[6] in order to work with enterprise management systems developed abroad, such as SAP products, so localization was essential.

The alluring promises of business informatization pushed Si-Trans towards making the first step in obtaining a transportation support software system.

3.3 Decision to Build

In around 1996, Si-Trans adopted the approach of in-house development. The development of an information system started with a single developer. According to common knowledge among Si-Trans employees, this person did not have software engineering training.

The system had a thick client architecture: a single central SQL database was established in the headquarters in Moscow; thick clients were distributed over locations in different countries so that Si-Trans employees could exchange data through the database. A thick client was a Windows application with a traditional point-and-click GUI. Thick clients of Si-Trans had complicated business logic wired in their source code. The big picture, though, was quite simple: clients sent SQL queries to the database and received datasets in return. In the beginning, the system offered a basic feature of viewing and editing several database tables with cargo movement information.

3.4 Interpretation: Building In-House as an Adequate Response

My overall interpretation of the 1996 events is that, despite lacking[7] a clear vision of requirements and alternatives, top management of Si-Trans intuitively went in a reasonable direction for the context they were in. There was no explicit decision of committing to internal development as opposed to relying on another company to provide an enterprise management system.

Choosing between adopting COTS and in-house development was easy because there was no adequate commercial solution present on the market. Had the solution been present, it would have made more sense to go with in-house software production anyway. There are several reasons:

- *An unstable and volatile environment:* the evee-changing environment produced a lot of requirements drift. How many clients do we serve? What rate of transportation will we have in 6 months? What regulations on customs clearances will be passed in the next month? How will political climate change Russia's relations to neighbors? All these questions might have

[6] Back in 1990s, even pathetic English speaking and writing skills used to cost a lot on the job market.

[7] Si-Trans did not possess any solid software engineering expertise in 1996.

had unpredictable answers, so the company needed a lot of flexibility. In-house development provided the flexibility.

- *Absence of strict time limits to produce the solution:* the company was left to its own devices. Given the chaotic spirit of that time, there would have been no disaster if the system had been deployed 3 months later. The information system project started as an experiment, without the desire of top management to invest up-front into it.

- *An immature domain of enterprise management systems in Russia:* to my knowledge, no enterprise management system was an accepted leader at that time. No related standards were recognized either.

The architecture chosen for the homegrown information system was, in my opinion, adequate for the needs the company had in 1996 because of:

- *Existing technology support for the thick client architecture:* popular implementation frameworks of that time favored the client-server paradigm. Hence, the initial development investment was low, and the system was quick to deploy.

- *Few employees at Si-Trans:* there were no more than 20–30 system users in the company. Therefore, the homegrown system could easily meet the performance needs.

- *Small implementation size:* the system's code was, apparently, not large in the beginning. Also, the size did not grow fast with only one staff programmer. As a result, stability issues with the system were manageable.

- *Simplicity of the initial functionality:* in the 1990s, the internal system provided only basic support for information exchange with only few complicated business rules. Therefore, these rules were easy to change.

Software development in Si-Trans started in a constantly changing environment with little guarantees. Even existing laws and regulations were not guaranteed to have been observed. All that gave a rise to the management style of Si-Trans that, I believe, defined the future of the company. There are several characteristics to this style:

- *Inconsistencies in management:* a decision made on day X may contradict another decision made on day X+1. For instance, one top manager decides to deploy a system at a new location immediately, while another top manager overrides in the next day saying there is no hurry, but the first manager does not know about this decision. This situation produces utter confusion among software and administration teams.

- *Strive for flexibility in decisions:* if several alternatives arise, it is likely that the one that minimizes commitment and cost will be taken, not the one that is advantageous in the longer term.

- *Experimentation:* many initiatives are started by top management not out of need to cut expenses or meet business goals, but because of their fear of missing an unknown opportunity. This leads to spending time in experiments that spread resources, but can find something new or missing in the context of a volatile environment.

- *Insufficient planning:* Si-Trans' management was very reluctant to do long-term analysis, estimation, and planning. Many decisions were reactions, not preventions, and they were aimed at fixing situational problems that often were symptoms of deeper issues.
- *Strive for control over fine-grained details:* managers looked for absolute control over low-level details of how Si-Trans worked. Specifically, they exercised a lot of fine-grained control over the software process. For example, a top manager might have envisioned a particular UI (that actually contradicted the conceptual model of the application) and did his best to make sure the UI was implemented, with little respect for software engineers' reasons for objecting.
- *Reluctance to invest into long-term benefits:* many decisions were as if the company would exist for merely several months. For example, no documented analysis of Si-Trans' computing needs has even been carried as it was viewed as a "too fundamental" process that would yield no immediate profit to the company's main revenue activity—selling transportation services.

4 Changes in the COTS Suitability

Si-Trans survived the chaos of the 1990s and was successful in its business during the 2000s. As the early 2000s passed, changes occurred in Si-Trans, its environment, and the information system. This section describes events that happened in the interval approximately in 2002–2006, as shown in Figure 4. Further statements about the history of Si-Trans are based on the substantial anecdotal evidence obtained from Si-Trans employees and on information extracted from code. However, the changes described below occurred over several years, so I will not give any precise time reference for them.

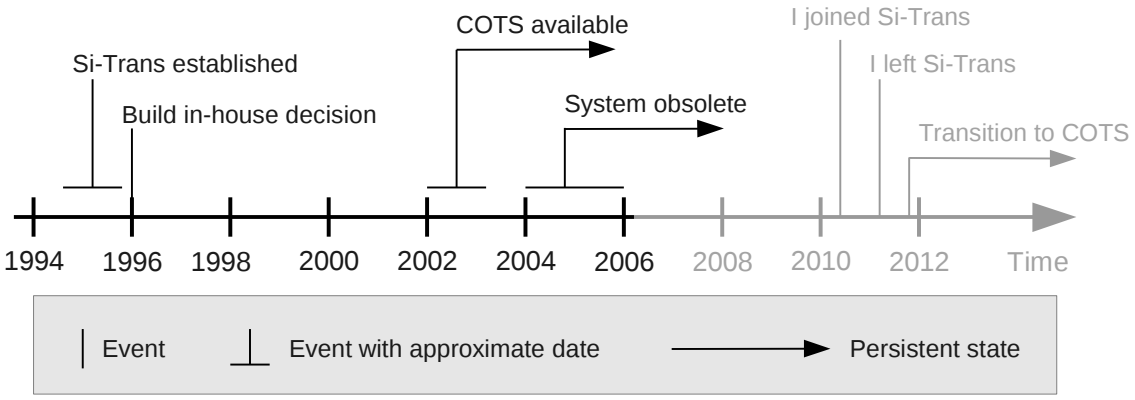

Figure 4: The timeline: 1995–2006. Change of circumstances.

4.1 Changes in the Business Environment and the Company

In early 2000s, the legal environment of Si-Trans became less fluid and more predictable as a result of Putin's stabilization strategy. Common practices at customs, tax agencies, and financial audit agencies solidified. The main outcome was that report forms became fixed and, therefore, they could be generated automatically by software. The risk that forms will radically change decreased.

The market for logistic services also became more solid: transportation companies in Russia and Eastern Europe were either driven out of the market or made a solid position there. The set of Si-Trans' clients included several companies with permanent relations. Therefore, it became much easier to predict type and volume of demand. Competitors and their strategies also became more apparent and amenable to analysis and prediction.

By 2005, the Si-Trans staff had grown to approximately 100 people. They were organized into the same structure as described in Figure 2: regional divisions handled client interaction and transportation; other concerns were handled centrally in Moscow. Every employee who worked with clients and transportations had to use the information system, so the number of system users grew drastically.

Despite the company's growth and the stabilization of the major external factors, the management culture remained the same in the company: top management acted as if there was still the same degree of uncertainty as in the 1990s. Even in 2010–2011 I observed the same style as the company seemed to have in the 1990s (see Section 3.4).

In addition to the business and organizational context, the information system's evolution is another important factor in the COTS suitability.

4.2 Changes in the Internal System

The organic information system grew steadily over 10 years. The situation in 2005 was:

- Apart from the essential information support on transportations, customers, and employees, the system offered more complicated functionality to exchange data about contracts, payments, money transfers, and warehouses.
- The number of system users increased to approximately 100.
- Regardless of the implementation growth, no documentation apart from source code comments was created. The high-level design knowledge was carried in the heads of software engineers.
- The architecture remained the same: a single database handled requests from thicks clients spread over the continent of Eurasia.
- The technological platform[8] on which the system was built grew old; new versions offered attractive features (e.g. transaction support), but migration attempts were not successful because of technical difficulties.

In addition to these internal changes, it is significant that a COTS alternative to the internal information system emerged.

4.3 Changes in COTS Availability

In 2002–2003, a significantly advanced version of a COTS product for enterprise management, **1C:Enterprise** [5], was released. It featured:

[8]The information system was based on the ODBC code to connect to a Microsoft SQL database and on Borland C++ Builder UI forms for the graphical interface.

- Ready-to-deploy components for personnel management, accounting and finance, wares management, contact management, document management, and tax report generation. Their flagship component was the bookkeeping support, which was widely used and recognized in Russia.
- Different options for client-side applications. Thick, thin, and web-based clients were available.
- A more scalable architecture. It was possible to decouple data storage and client request processing. Also, it was possible to have several databases and client processing servers in order to distribute requests between them.
- A platform that comprised a domain-specific language (DSL), an API for common functions, and a development toolkit. All of that could be used to develop new components and to integrate them into the 1C system. 1C components could communicate with other 1C components through calls in the DSL as well as with components based on other technologies, such as .NET, through COM interfaces.

By 2004–2005, many Russian companies have been making wide use of the 1C platform. As a result, the demand for 1C programmers grew and drove their salaries up. The job market replied with a higher presence of 1C expertise by 2005. Nevertheless, Si-Trans ignored the success of 1C:Enterprise and continued to develop the internal system.

4.4 Interpretation: Missing Out on a Beneficial COTS Alternative

Over these 10 years, critical factors that influence the "COTS vs. build" decision changed. If Si-Trans had to make this decision back in 2005–2006, it would have been more reasonable to base the information system on the 1C COTS for the following reasons:

- The legal and business environments have stabilized. There was no point of being extremely flexible anymore. Moreover, long-term investments into COTS were much less risky in the light of knowledge that Si-Trans will not cease to exist in the next month.
- Si-Trans' internal information system became obsolete because it reached its architectural limits. The size of the user base grew to exceed the ability of the architecture to support the required performance.
- The in-house system grew beyond the software team's ability to support it. As the system increased in size, its technical debt piled up because of a number of factors that we will discuss further below. The source code had grown too big to be effectively handled by a small team that was under constant pressure from the management.
- The 1C platform was reliable and tested enough to have been adopted. It also took a place of a de-facto ERP standard in Russia. Had Si-Trans bought into 1C in 2006, they would have (a) accommodated for the company's growth, (b) avoided subsequent issues with the system developed in-house, and (c) taken the control over software development from the hands of the chaotic top management.

According to the software engineering wisdom outlined in Section 2, these facts indicated that in 2005–2006, it had been appropriate to switch to the 1C software. However, top management made a different choice—and chose to stick with the initial in-house approach. The reason for that was, in my opinion, that the transportation services, which were the core of Si-Trans, lacked any support in 1C: it had no data model or user interface to operate transportation. And the management probably decided that implementing it there would be too much work. Nevertheless, I think that the other alternative—to continue maintaining a lot more modules than just the transportation one in-house—was far less practical at that point.

5 Protracted In-House Development

This section describes my personal experience in Si-Trans during 2010–2011, as shown in Figure 5. By that time, the internal system did not work acceptably, and the management was exploring an opportunity to cut the technical debt.

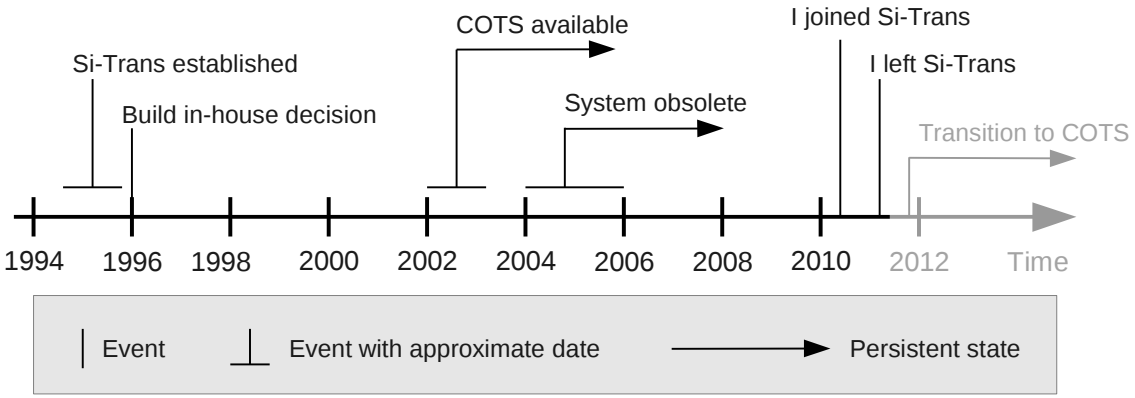

Figure 5: The timeline: 1995–2010. My experience at Si-Trans.

5.1 Accumulated Technical Debt

I joined Si-Trans in 2010 as a software developer. I had two main responsibilities: to develop the old information system, which was almost 15 years old at that time, and to prototype a new one. Priorities between these two activities were not specified up-front. Looking back now, I ended up spending much more time on the old system.

The relatively wide spectrum of duties allowed me to communicate with different departments of Si-Trans and find out the details that this practicum is based on. At the time I got familiar with the state of the in-house development in Si-Trans, a number of technical issues revealed themselves:

- *Unmanageable source code:* the internal system's code had grown unsuitable for reading, understanding the high-level intent, and business logic extraction. This was true due to the increased size, approximately 300 KLOC including comments, and because the structure of code modules was plain. If a developer tried to comprehend the source code organization, he faced many same-level entities that interacted in many subtle ways. Even by the end of my employment at Si-Trans, I could still hardly understand the role of half of the code modules without scrutinizing them.

- *An eroded database schema:* After years of carelessly managed evolution of the database, many tables and fields were obsolete or used in ways that nobody remembered. Several tables captured similar concepts—e.g. city, office, and warehouse—and needed many-to-many mappings between each other because redesigning in a better way would take too much effort. One particularly poor table contained transportations data: it had over 100 fields and over 30 thousand records, which followed different semantic assumptions about fields. To my knowledge, there was no single person in the company who understood what each field in this table meant.

- *Low performance:* the system did not meet end users' performance needs. As the number of users grew to about 160 in 2010, the database became a performance bottleneck. This issue introduced noticeable waits for the users. It was not possible to sufficiently speed up the operation under the constraints of the thick-client architecture.

- *Poor runtime stability:* subtle implementation bugs did not get sufficient treatment for long periods of time. The toughest problems arose from conceptual mismatches between parts of the system. One example was different assumptions about currency cross rates[9] in financial transactions. Some parts of the code treated it as just a derivative of individual currency rates. At the same time, there existed code that treated a cross rate as an independent characteristic of a transaction and, thus, violated the assumption of the former code. Such semantic mismatches led to corrupting database tables and crashing all thick clients company-wide.

- *Insufficient computing environment flexibility:* the internal system could be deployed on a small variety of hosts. Thick clients could plausibly run only on Windows-based machines with a high bandwidth and a stable Internet connection because they sent relatively big queries and datasets over network. Si-Trans management wanted to deploy the system at warehouses in China to enable more precise tracking of containers with goods, but it was not possible since only an intermittent and low-bandwidth Internet connection was available at the warehouses.

- *Difficulties with updating thick clients:* our software engineering team could not find an appropriate technical solution for updating thick clients. It needed to be a compromise between shutting down all clients immediately as soon as the update was available and letting clients run as much as they want. Therefore, it was often the case that users were distracted by non-critical updates; also, many users missed critical updates and sometimes corrupted the database, making it incompatible with the newer version of clients.

So, by 2006 the architecture and the codebase did not meet Si-Trans' requirements. Apart from the technical trouble, I observed the weak management practices of Si-Trans' top executives that are exemplified in the next subsection.

[9] A *cross rate* between currency A and currency B through currency C is a ratio of the A-to-C rate to the B-to-C rate. For example, a cross rate of the euro to the Chinese yuan through the Russian ruble would be the euro-ruble rate divided by the yuan-ruble rate.

5.2 Management Style

Top management of Si-Trans was a central point of control for all branches of the company: operations, accounting, legal support, and so on. The software engineering team reported directly to top management as well. I was very uncomfortable with their practices: top management was inconsistent, chaotic, and noncommittal, just as described in Section 3.4.

Two concrete scenarios that I personally witnessed depict the general style of management. These scenarios exemplify how flawed the interaction of management and engineering was at Si-Trans. You can find them in Tables 1 and 2.

Outset	Half an hour before a work day ends, a top manager shows up in person at the programmers' office and requests that one feature be implemented before the end of the day. The feature is highlighting all transports that have been stuck at customs for more than 10 days. The motivation for the urgency is that branches in China start daily operation earlier and would like to use the functionality to detect late transports the next day.
Events	The feature is quickly implemented as a *complicated condition* in thick clients that infers how much time a transport spent at customs from its other attributes. Then an update is distributed among locations before the end of the day. This business rule about highlighting transports was not documented anywhere except the code. Several weeks pass, and the knowledge about the rule's existence and location in code leaves the heads of software developers. The manager who remembers that piece of business logic turns up and asks a different developer to change the condition to 15 days based on the feedback from other employees. Not having time to elaborate on his request, the manager leaves the engineers' office. This time, the engineers decide to approach a problem in a more principled way and define a *database field* "late" for transport. The attribute is populated by a database trigger, and highlighting is implemented based on this new field. The original rule remains in the code because nobody remembered about it, nor could anyone find it quickly enough to remove.
Outcome	Inconsistent display of transport status confuses users. The development team later spends days to find the root of this inconsistency and remove the first implementation that was based on a condition in clients.

Table 1: Management practices scenario 1: undocumented conflicting patches.

5.3 Reimplementation Effort

I was hired to help reimplement the old information system in a new architecture, labeled *V2.0*. The new architecture introduced a middle tier between clients and data storage. The middle tier would receive requests from clients, translate them into database queries, apply business rules, and return results to clients. The clients were supposed to become thinner because they did much less computing and they were independent from the database schema. The new architecture aimed to remove the performance bottleneck at the database and simplify the update procedure. A new data

Outset	A top manager has a very specific GUI in mind and wants it implemented as soon as possible. The GUI will allow users to see and edit the chain of cities through which a transportation goes. No time is given to developers to realize the global impact of this particular GUI on the system.
Events	The GUI and the corresponding data model are implemented and deployed as quickly as it was possible. Several weeks pass, and users attach chains of cities to a number of transportations. After the GUI has been implemented, it turns out that it has a conceptual mismatch with the system: it should have used the concept of "branch" instead of the "city". The initial intent of the GUI was to provide traceability about which branch participated in handling which transportation. This analysis is impossible when chains of cities, not branches, are recorded.
Outcome	Programmers spend a lot of time to spot all code that has been written under the incorrect assumption, to change it, and to convert the data in the database. Being under constant time pressure, programmers leave bugs behind.

Table 2: Management practices scenario 2: conceptual mismatch.

model was expected to use the insights obtained over 15 years and to get rid of obsolete parts. Also, there was an intent to develop the new system in a more disciplined way that would include strict coding conventions, design documentation external to code, automated testing, and traceability of requirements to code.

Another decision was to build the new information system with a modern user interface framework—Qt 4. It was more powerful than C++ Builder in two aspects: (a) Qt 4 offered a higher level of abstraction in day-to-day programming activities, with many convenience classes and syntactic sugar; (b) Qt 4 allowed a higher level of reuse for UI parts and behavior. Using Qt 4 promised higher development speed, fewer bugs, and the integrity of user experience.

After one year, our team finished a prototype with only a small part of the functionality of the original system—creating, displaying, and marking containers with goods. We devoted all our remaining time to supporting and extending the old system according to the management's requests. Shortly after I left Si-Trans, top management stopped allocating resources to the reimplementation, despite the eagerness of the software development team.

5.4 Interpretation: Peak of Technical Debt

The technical issues described in this section came about, I believe, for two immediate reasons:

- *The thick client architecture stopped meeting the changing requirements:* the architecture could not support needed performance, stability, and easiness of updates because it was overly database-centric and, at the same time, placed too much responsibility on client applications. An enormous technical debt separated this architecture from the requirements, and this gap was impossible to bridge in 2010 by just putting more time into the old system. A complete redesign was needed.

- *The source code evolved into a big ball of mud* [8]: the codebase was a mess of low-level memory management, business logic, network communication, database queries, user interaction,

workarounds of "bugs with design-level significance", outdated or just dead instructions, and irrelevant comments. It was hard to extract the design intent from the source code, and even harder to modify it without introducing an implicit inconsistency with other parts of code.

The root cause of these two technical concerns was the management style of Si-Trans' top management. Not only did this style accelerate the accumulation of technical debt, as shown in the examples, but it also caused the company to miss the opportunity to adopt COTS when the in-house system became obsolete. In my opinion, this style's fundamental problem was *too much control and flexibility* of the organic development process. The mere existence of programmers placed an obligation on managers to "keep them occupied" no matter what. Active involvement of the managers who did not possess much skill or knowledge about software development was very harmful for the project because they had absolute power to alter any requirements and control the developers. In other words, I believe that the mentality of "we pay them—they do whatever we want" was a source of the most trouble for Si-Trans' homegrown software.

One might argue that the absence of a software architecture-centric development process could have been the primary reason for the project's failure. I cannot agree with this statement for two reasons. First, software architecture is just *a* way to succeed, not *the* way. Si-Trans' in-house development would have fared well even without paying explicit attention to architectural design and documentation. The same, if not better, effect could have been achieved by merely letting the team invest more time into maintaining internal consistency of the original system or into the reimplementation later on. Second, I do not see how an architectural process could have been successfully set up in Si-Trans. Any such process would have required continuous investment into "invisible" aspects, as perceived by managers, which was not acceptable to them. Instead, they would have just forced implementing their "brilliant" ideas, which came to them at a high rate. So, it was not the lack of any particular process that made the development fail, but the disruptive behavior of top management.

The relationship between the major concepts of this practicum is shown in Figure 6. The chaotic management style led both to accumulating technical debt and sustaining an in-house development longer than needed. At the same time, the in-house development paradigm let the managers do whatever they wanted. Adapting the 1C COTS could have been a well-fit decision to break this vicious circle.

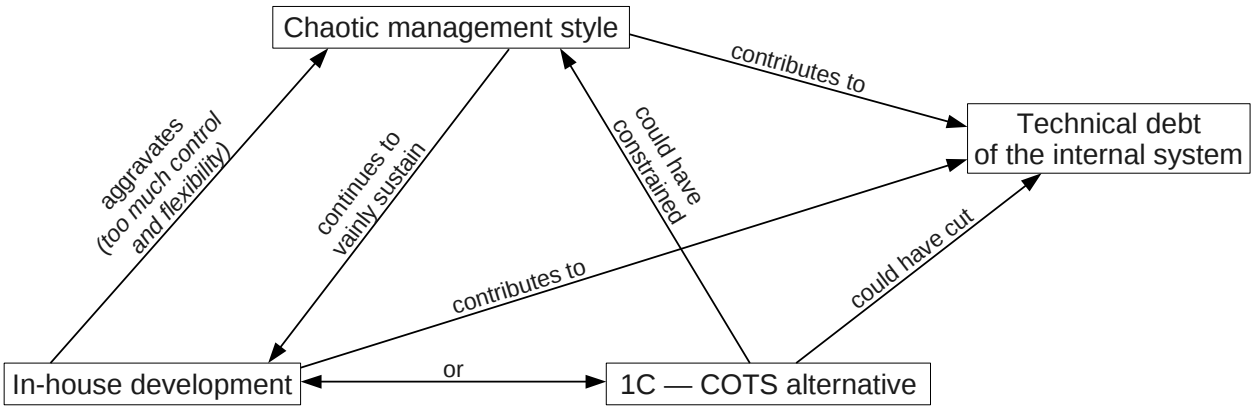

Figure 6: Interrelation of concepts for the buy or build question of Si-Trans.

The reimplementation effort aimed at the most serious issues of the old system. The more suitable 3-tier architecture of V2.0 could provide appropriate performance and reliability. The fact that a disciplined development started from scratch would mean that the problems of technical debt would not bother Si-Trans for at least several years. Additionally, the Qt 4 technology would have saved plenty of development efforts in the long run; also, Qt 4 could have had a positive effect on the usability because of potential reuse of UI artifacts.

In spite of all the advantages of V2.0, it did not go as far as it was expected to. The problem was that the implementation rate for the new information system was low since the old system sucked a lot of time from all programmers. They were not only distracted by fixing bugs in the old system, but also top management kept requesting new features there. Managers were not able to stop extending—not merely maintaining—the old one, even though the software team proposed that many times[10]. So again, a non-optimal strategy of the management intervened. The developers could not keep actively extending both systems at the same time, so the progress on V2.0 was too slow. It was obvious for both programmers and managers that it would take years to implement all functionality of the old system in V2.0. That's why top management lost their faith in V2.0.

5.4.1 Perspective of the Developers on COTS

One might think that the Si-Trans software developers were responsible for missing the opportunity to adopt 1C. This argument is supported by the fact that technical expertise is needed to point out a suitable COTS option. However, the developers never had an opportunity to do a COTS suitability analysis because they had been overburdened by management feature requests. And, of course, they had never received a request to evaluate COTS alternatives.

From the programmers' point of view, the 1C platform was too constrained and rigid to be adapted to the management's requirements. Moreover, it gave too much support to programming by eliminating technical challenges like memory management, which "made programming fun". 1C also featured a domain specific language with Cyrillic alphabet, writing in which was too much of a cultural shock for software engineers who were conventionally trained in C++ and Java. Finally, some engineers believed that 1C's promises of scalable architecture and suitability of the ready-to-deploy components were exaggerations[11].

So, the Si-Trans development team did not sincerely consider 1C worthy of closer analysis. The developers constantly informed the management of the system's poor state and, in fact, initiated the reimplementation effort from bottom-up. However, the adherence to the in-house approach did not serve them well because it was too much spoiled by management.

6 Transition to COTS

This short section describes the events that happened in late 2011–early 2012, after I had left Si-Trans.

[10] Features that management asked for were hardly an immediate business need for Si-Trans. Most of changes were UI tweaks and additions that added little value, but took much time to implement in the messy old system.

[11] Those engineers substantiated this claim with their limited experience with parts of 1C and folklore failure stories. Those with much less exposure to 1C, like me, just treated the claim as a valid temporary assumption.

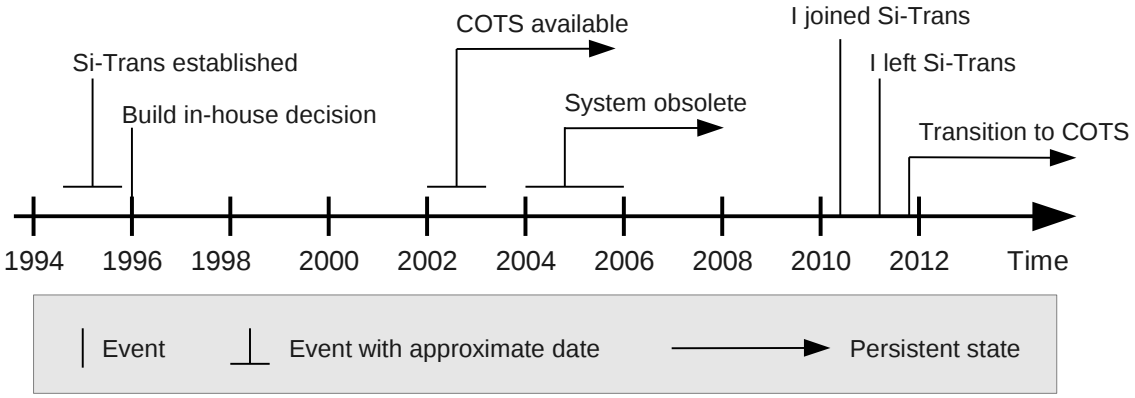

Figure 7: The timeline: 1995–2012. Embracing the COTS.

6.1 Aftermath

In November 2011, top management of Si-Trans decided to shift the software paradigm of the company to the COTS-based development. They fired the software engineering staff except for a team lead who had both deep domain knowledge and technical skill in building information systems. In December 2011, Si-Trans started joint analysis with 1C on adopting 1C:Enterprise—still by far the most widespread enterprise management system in Russia. Adaptation engineers from 1C started collaboration with the Si-Trans' higher executives to elicit the set of 1C ready-to-deploy components to use and the set of Si-Trans-specific components to implement.

The former development team lead left Si-Trans in March 2012. According to him, he did not want to participate in "yet another turn of this idiocy". As he was the last person I knew well enough at Si-Trans, this concludes the story of software development and procurement at Si-Trans.

6.2 Interpretation: On Right Track 6 Years Late

Top management of Si-Trans, despite their shortsightedness, were able to notice the enormous losses from developing the old information system. Stability and availability issues as well as the need to pay a team of 5 software developers every month were strong enough arguments to reconsider the whole paradigm of in-house software creation.

It is not clear whether top management identified the sources of problems—management style, outgrown architecture, and technical debt—or just intuitively decided to try another option because their current way of doing things did not yield enough benefit. I believe, however, that the implementation attempt was a convincing argument that, for whatever reasons, the company can barely handle the in-house development. Probably, based on this evidence top management decided to start a transition to COTS.

7 Lessons Learned

Si-Trans started developing an information system in-house back in 1996, when the business context required agility to adapt to the rapidly changing circumstances. As the information system evolved through mid-2000, its architecture grew less and less appropriate, while its codebase grew less and less manageable. At approximately the same time, an advantageous COTS alternative emerged,

but it was ignored by the top management of Si-Trans. When I joined the company in 2010, the information system was at the peak of its technical debt; adding to all that, detrimental management practices incessantly pulled the development down. Shortly after I left Si-Trans, top management drastically changed the software paradigm to COTS-based development. This long-needed change arrived 6 years late, but better late than never.

I draw the following conclusions from the story of Si-Trans:

- **Business logic changes and thick clients** — the thick client architecture is poorly maintainable when business logic (a) gets complicated and (b) changes frequently. This is true because the business rules are lost in source code. Development can continue only if all programmers know—or at least can quickly find—all the business rules, which is not true in case of the thick client architecture. When using this architecture, the technical debt piles up faster, and the software reliability drops.

  Si-Trans kept developing a thick client system for 15 years. Frequent changes in business rules eroded the code so that it would take much time to find and extract the rules from it. As the size of the system grew, it became increasingly harder to ensure integrity between the rules. As a result, occasional inconsistencies in the implementation amassed and severely damaged the reliability of the system.

- **Management and debt** — a volatile, chaotic, and noncommittal management style speeds up the accumulation of technical debt because development-level requirements change too often. Excessive workload pressure on programmers also makes technical debt grow faster: bugs are left behind, and new features are built as workarounds.

  There were three main ways in which the chaotic and inconsistent management of Si-Trans accelerated the rate at which the technical debt was created. First, the frequently changing, sometimes contradictory requirements broke programming abstractions, which resulted in the ball of mud source code [8]. Second, permanent time pressure on software developers resulted in leaving old issues in the system and building new ones on top of the low-quality code. Third, the reluctance of the executives to strategically invest man-hours into fixing bugs made engineers build the system on workarounds. This contradicted a common software engineering wisdom that proscribes creating new code assuming that old bugs exist [16].

- **Stability in management and environment** — for a software project without fixed requirements, there is a "degree of volatility" in management that defines how flexible a software development process is with respect to the requirements. There should be a match between this degree of volatility and the stability of the project's environment, which is capable of evolving and, hence, changing the requirements. Stable environments should be matched with a more fixed, rigid process, while volatile environments are better approached with a more agile development process.

  Si-Trans' software process can be characterized as very lightweight and flexible. This was appropriate in the unpredictable environment of 1996-2000. However, as the environment stabilized, the flexibility of process and the active search of new requirements did not give much advantage; conversely, it undermined deep domain analysis and disciplined code reuse.

- **Continuous monitoring of the buy vs. build question** — the issue of buying COTS or building software in-house should not be treated as a one-time question. Constant re-evaluation of fitness for buying or building helps not miss beneficial opportunities as an

environment changes. Engineers should be equipped with time and authority to collaborate with management on this question.

The sunk cost bias might have been one of reasons why Si-Trans missed a COTS acquisition opportunity. Top managers might not have considered acquisition at all because this would mean throwing away the organic system that had received huge investments: developers' salaries over many years; efforts of the management to govern the development; time lost by the system's users because of faults, low performance, and counterintuitive interfaces. Another reason might have been that the high executives were not aware of how suitable and extensible 1C was. The software team could not help because they lacked time budget to investigate 1C. In any case, an objective cost-benefit analysis in 2006-2011 would have revealed that adopting 1C had been a more advantageous option then sticking with the downhill in-house engineering.

- **Benefiting from control dilution** — if a company partly loses its control over software development as result of acquiring COTS, is often seen as a disadvantage of buying software [17]. However, the lack of control caused by COTS can prevent volatile management practices from eroding the software because COTS solutions preserve their assumptions and code. So, the lack of control over a software process may help in case of chaotic management practices.

  Many issues with the in-house development at Si-Trans came, in my opinion, from the excessive control of top management over the flexible development process. Had there been a force keeping the requirements from drifting, the old information system might not have had many of its problems. A COTS-based immutable architecture might have been such a force.

References

[1] Eric Allman. Managing technical debt. *Queue*, 10(3):10–17, March 2012.

[2] Carina Alves and Anthony Finkelstein. Challenges in COTS decision-making: a goal-driven requirements engineering perspective. In *Proceedings of the 14th international conference on Software engineering and knowledge engineering*, SEKE '02, pages 789–794, New York, NY, USA, 2002. ACM.

[3] Lisa Brownsword, Carol A. Sledge, and Tricia Oberndorf. An activity framework for COTS-Based systems. Technical Report CMU/SEI-2000-TR-010, Carnegie Mellon University, Software Engineering Institute, 2010.

[4] David Carney and Fred Long. What do you mean by COTS? Finally, a useful answer. *IEEE Softw.*, 17(2):83–86, March 2000.

[5] 1C Company. 1C:Enterprise 8 - the system of programs. http://v8.1c.ru/eng/the-system-of-programs/.

[6] Allen Eskelin. *Technology Acquisition: Buying the Future of Your Business*. Addison-Wesley Professional, 1 edition, June 2001.

[7] M. D. Feblowitz and S. J. Greenspan. Scenario-Based analysis of COTS acquisition impacts. *Requirements Engineering*, 3(3-4):182–201, March 1998.

[8] Brian Foote and Joseph Yoder. Big ball of mud. http://www.laputan.org/mud, 2000.

[9] Patricia K. Lawlis, Kathryn E. Mark, Deborah A. Thomas, and Terry Courtheyn. A formal process for evaluating COTS software products. *Computer*, 34(5):58–63, May 2001.

[10] Jingyue Li, Reidar Conradi, Odd Petter, N. Slyngstad, Christian Bunse, Umair Khan, Marco Torchiano, and Maurizio Morisio. An empirical study on Off-the-Shelf component usage. *In Industrial Projects. Proc. 6th International Conference on Product Focused Software Process Improvement (PROFES'2005)*, 13:13–16, 2005.

[11] Jingyue Li, Reidar Conradi, Odd Petter N. Slyngstad, Christian Bunse, Marco Torchiano, and Maurizio Morisio. An empirical study on decision making in off-the-shelf component-based development. In *Proceedings of the 28th international conference on Software engineering*, ICSE'06, pages 897–900, New York, NY, USA, 2006. ACM.

[12] B. Craig Meyers and Patricia Oberndorf. *Managing Software Acquisition: Open Systems and COTS Products*. Addison-Wesley Professional, 1 edition, July 2001.

[13] Abdallah Mohamed, Guenther Ruhe, and Armin Eberlein. COTS selection: Past, present, and future. In *Engineering of Computer-Based Systems, 2007. ECBS '07. 14th Annual IEEE International Conference and Workshops on the*, pages 103–114. IEEE, March 2007.

[14] Maurizio Morisio and Marco Torchiano. Definition and classification of COTS: a proposal. In *Proceedings of the First International Conference on COTS-Based Software Systems*, ICCBSS '02, pages 165–175, London, UK, 2002. Springer-Verlag.

[15] Vijay Sai. COTS acquisition evaluation process: The preacher's practice. In *Proceedings of the Second International Conference on COTS-Based Software Systems*, ICCBSS '03, pages 196–206, London, UK, 2003. Springer-Verlag.

[16] Joel Spolsky. *Joel on Software: And on Diverse and Occasionally Related Matters That Will Prove of Interest to Software Developers, Designers, and Managers, and to Those Who, Whether by Good Fortune or Ill Luck, Work with Them in Some Capacity*. Apress, August 2004.

[17] Kurt Wallnau, Scott Hissam, and Robert C. Seacord. *Building Systems from Commercial Components*. Addison-Wesley Professional, 1 edition, August 2001.