Towards the Competitive Software Development

Andrzej Zalewski and Szymon Kijas[1]

[1] Warsaw University of Technology,
Institute of Automatic Control and Computational Engineering
{a.zalewski,s.kijas}@elka.pw.edu.pl

Abstract. The concept of competitive software development is founded on the observation that the system's owner and development companies have not only common interests, but also conflicting interests as well. This is particularly true in the case of large-scale software systems. Competitive development is an answer to the syndrome of large-scale software systems evolution and maintenance being monopolised by the companies that originally developed these systems. Competitive development is founded on the idea that the entire system is divided into smaller units, which are independently developed by different companies, i.e. no co-operation is assumed between various development organisations and there is no communication between them. However, strong and efficient co-ordination has to be performed on behalf of the system's owner in order to make such an approach successful. These assumptions are radically different to the typical collaboration assumption for agile development. This prevents one software company from making a system owner entirely dependent on its services. We show that such demonopolisation can save large sums of money, making the prices of software development considerably lower than they would be in the case of a single software development company. Our experiments show that, if efficiently co-ordinated, such a distributed, competitive development requires a similar effort to traditional approaches.

Keywords: Software process, software process improvement, empirical studies.

1 Introduction

This paper has been inspired by the work done for Public Procurement Office in Poland, supervising purchases for public or publicly-owned institutions. One of its responsibilities is to ensure that public money is spent on services rendered and goods supplied by competing contractors. Large-scale software systems are a very specific subject of public procurement.

These are the largest and the most complex systems, tailored to the specific requirements of a commissioning organisation. Their evolution and maintenance becomes usually monopolised by the company that originally developed the system. This enables them to act in the condition of limited competition, allowing them to expand their profit margins above the market average, which is exactly the situation that every company is striving for.

D. Caivano et al. (Eds.): PROFES 2011, LNCS 6759, pp. 103–112, 2011.

Such a monopolisation syndrome occurs whenever any part of the system becomes too large, i.e. too complex to freely hand over its development to some other development company (competitor). This is studied in detail in part 2. We argue that, to prevent monopolisation, an antimonopoly function should be included in the development, maintenance and evolution processes for large-scale software systems.

This approach is the core concept presented in this paper – part 3. Its aim is to enable competing software companies to work independently on the decoupled parts (macro components) of the same software system, while assuring necessary coordination. No co-operation is assumed between different software development companies. The process has been validated experimentally, as described in part 4. The impact of such a novel approach on the various aspects of software development and its cost have been discussed in parts 5 and 6 respectively. A summary of results and ideas for further research are given in part 7.

2 The Software Monopolisation Syndrome

From a business point of view, software products are supposed to overcome market competitors, attract clients, gain a certain market share, and retain and attract clients to buy newer and newer versions or other software from a given company. At the same time, they should be hard to substitute by market competitors. What makes a software product successful depends on lots of factors such as the target group of the product, the software properties, market conditions etc. etc.

Large-scale software systems are a very specific kind of software product in this context. They are built at the commission of a single entity and tailored to its specific requirements. There is usually no other organisation that could use that system. Their extreme complexity makes them very difficult to substitute by a system provided by another software company. They are also difficult to evolve by making changes and extensions. This effect has theoretically been described in [1], [2], [3] and attributed to the high level of coupling between system components.

As a result, such systems are typically perceived as a kind of a natural monopoly (similar to water supply and sewage system) in which maintenance and further development can only be carried out by the company that originally developed the system. The lack of pressure from competition results, as with any other monopoly, in the expansion of profit margins and increased cost of software maintenance and evolution.

The tendency of large IT systems to become monopolised comes from:

- the very nature of software systems i.e. their complexity, enhanced by the architectures dominating modern business applications designs,
- the inefficiency of system documentation and the means of knowledge management,
- the lack of motivation to minimise the monopoly,
- legal / regulatory / policy issues (e.g. intellectual property rights, special purpose systems like military ones).

We now delve deeper into details of the first three reasons, leaving behind the legal ones. To modify any IT system the developers need to know its construction in sufficient detail. Monopolisation is generally caused by the inability to obtain or transfer design knowledge on complex software systems efficiently in terms of time and cost.

This, in turn, is a result of the limitations of software documentation [4] and design knowledge management methods and tools [5], which, despite many years of research, have not yet been overcome.

If the given organisation does not have a team of its own shadowing the activities of the developing company, knowledge on the system and software design can only be obtained from system documentation, configuration and source code. If a system is kept small enough, the owner can always (given sufficient intellectual property rights to the source code and documentation) change the current developer with negligible disturbances to the system's operation. In such a case, knowledge on the system construction can be obtained by the new developers within a short time and at minimal cost.

However, when a system becomes too big, design knowledge cannot be obtained within a reasonable time and money by the new developers. This means that disturbances in the system operation cannot be avoided when the developing company is replaced with another one, or the necessary preparations would take a very long time (tens of months) and cost a lot (reports indicate 50% or even more of the entire maintenance cost [4]) before the change takes place. Even if the owner succeeds, the system is monopolised again – this time by the new developers. Another important factor is that, in the case of large-scale software systems, such a step is generally very risky in business and technical terms. Thus there is no real incentive for the owner to undertake such a challenge. If the system becomes too large, the owner cannot avoid the monopolisation of maintenance and evolution.

The problems of over-dependence on a certain system, and its developers, have not passed unnoticed. It currently takes two extreme forms:

- A single subsystem becomes too complex – this is the basic version of the syndrome presented above – companies try to resolve the problem by acquiring systems from various vendors (often overlapping existing functionality!) and integrating them with EAI solutions such as Enterprise Service Bus or Process Management Systems;
- An integration system (e.g. ESB) becomes too complex – this is the opposite situation encountered by companies that exaggerated with the extent and complexity of an integration solution – it is clear that integration solutions can easily be monopolised by the companies that developed them as the system becomes too complex and its owner grows too dependent on its operation.

3 Demonopolising Software Development

To draw up a remedy for the monopolisation syndrome we need to look more into the technical details of the monopolisation problem. The extreme cases of the monopolisation syndrome indicate that the source of the problem is located in a common component of a system on which large parts of the software depend. It was an integration solution in one of the extreme cases. However, in most cases, such a component is usually a database on which most of the components of a given software system depend. Most of the modern business systems use a layered architecture, which is imposed by all the most popular implementation technologies, like J2EE or .NET. Hence, modern business software is built on the foundation of a common database. This means that a change made to a single component can possibly impact all the other components, as illustrated in fig. 1.

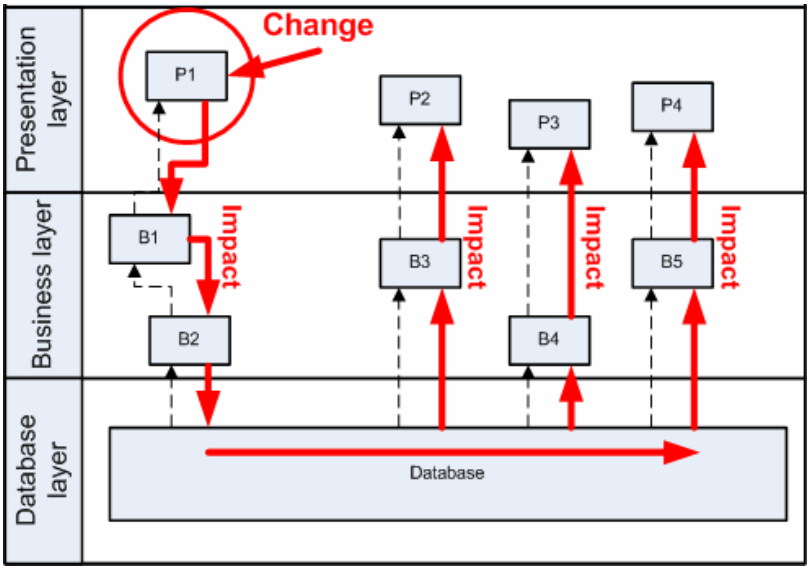

Fig. 1. Coupling in a three-tier software system

To eliminate, or at least to minimise, the possibility of monopolisation, we need to decouple strongly coupled components. It means that software should be split into parts that plug into an existing database without altering its structures common to a greater number of software units (components, etc.). Such macro components (applications, subsystems) could be developed independently of the work done by the developers of the other subsystems. Obviously changes to common database structures cannot be entirely avoided, but a careful consideration of the database can minimise such needs and their impact. Developing organisations can develop data structures of their own, but they should be treated as private unless the owner decides to include them in a common database schema.

This task has to be allocated to a skilful team, known as a "co-ordinating team", which should work on the system owner's behalf to ensure conditions in which various development teams can work independently without excessive interference. The organisation of competitive development has been illustrated in fig. 2. Its interesting property is that the teams can use different development processes.

The co-ordination team's role comprises:

- Common database management – the development and maintenance of the database structures common to larger parts of the system, negotiating changes to database structures with the software development companies and assessing their impact, providing information on the common database structures;
- Development organisation and co-ordination – this is part of the co-ordination effort needed to define and manage system decomposition, to place all the independently parts together, to create software releases, and to co-ordinate independent work;

- Integration management – the implementation of integration solutions necessary to integrate subsystems resulting from splitting subsystems that have grown too large, or from purchases of ready-to-use configurable systems (such as ERP);
- Complexity control – this means that whenever any part of the system becomes too complex, it is split into manageable, independently developable parts (this is a typical antimonopoly action similar to those undertaken by state competition protection agencies), or when co-ordination becomes too complex then separate systems could be merged.

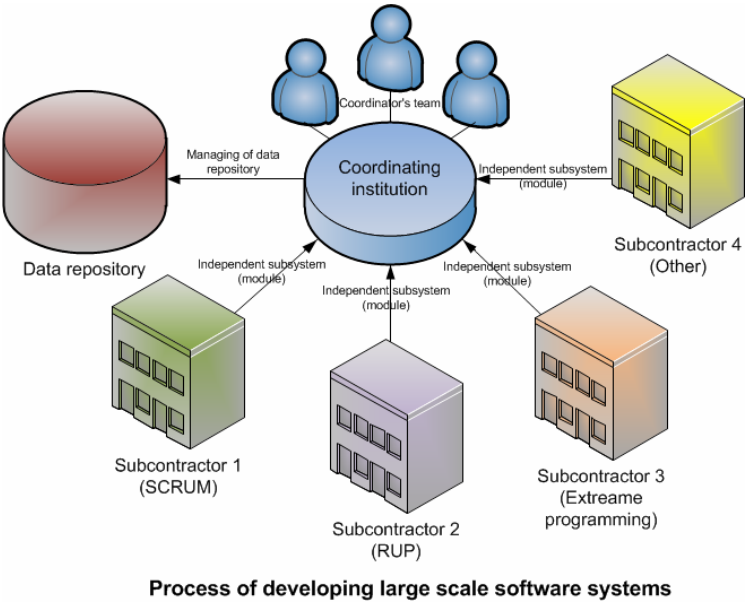

Fig. 2. Competitive development

4 Experimental Assessment

An experiment was performed to show that separated teams working independently under central co-ordination will not increase the amount of work done by the programmers compared to a single team working on the same project. The experiment was performed on the modular system for managing studies at our faculty. This system comprised three modules using the same database:

- **Module one:** Student module – an application used by the students for semestral declaration, course registration, browsing results, etc.
- **Module two:** Teacher module – web application used by teachers for subject descriptions, grading etc.
- **Module three:** Dean and dean's office module – GUI application used for configuration, approval for course registration, student promotion, final exam procedures support etc.

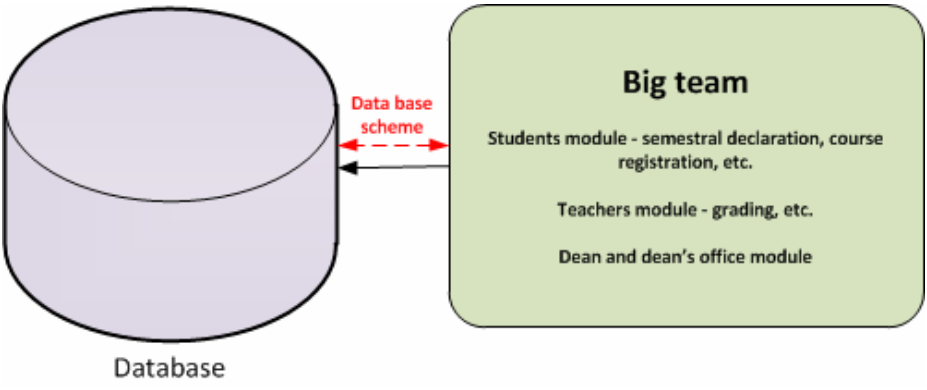

Fig. 3. Scheme of system developed by a single big team

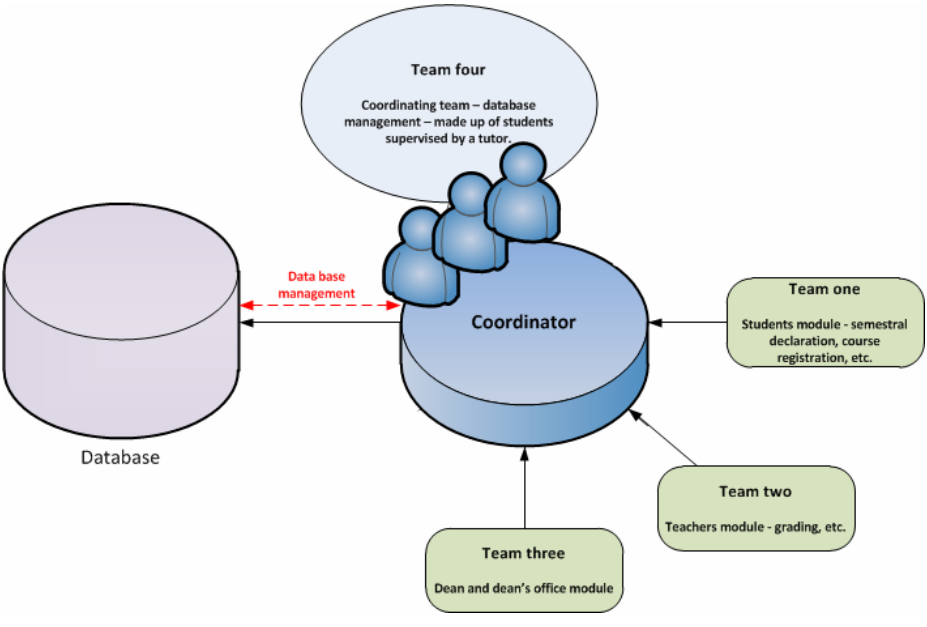

Fig. 4. Scheme of system developed by four small teams

These modules share database structures and their functionality partially overlaps.

Two teams of eight students were chosen for every time the experiment was conducted. One team was working together to develop the required functionality (fig. 3) without a predefined internal structure, but with a certain amount of co-ordination ensured by an early draft of a common data model. The other team (fig. 4) was divided into small teams consisting of two people each. Three of the sub-teams were assigned to the development of selected modules (one independent module for each group) and one became the co-ordinating group (responsible for co-ordinating the work and managing the common database structure).

All of the teams used Extreme Programming. The experiment was supposed to last for three working days (24 hours per person - which makes 192 man-hours altogether). The development times of individual modules and the entire system were measured and compared.

Table 1. Results of the experiment

First measurement				
	Module 1	**Module 2**	**Module 3**	**Total**
One large development team	49 man-hours	55 man-hours	98 man-hours	202 man-hours
Developing in small teams	55 man-hours	58 man-hours	107 man-hours	220 man-hours
Second measurement				
	Module 1	**Module 2**	**Module 3**	**Total**
One large development team	51 man-hours	49 man-hours	97 man-hours	197 man-hours
Developing in small teams	43 man-hours	51 man-hours	95 man-hours	189 man-month

The development effort measured during the experiments is presented in table 1. The analysis of the results and of monitoring the teams' activities revealed the following observations:

- General observation:
  - The level of effort required to develop the whole functionality turned out to reach similar levels in both approaches. This indicates that competitive development should not increase effort compared to conventional approaches (agile, RUP, waterfall).
- Observations concerning the single team exercise:
  - Weak co-ordination of work on the database structure was a large problem, due to many clashes between changes introduced by different people.
  - Team members trying to reuse code developed by the others lost a lot of time on debugging immature solutions of their colleagues. In many cases it required more time than developing the whole thing from scratch.
- Observations concerning the independent teams exercise:
  - Independently working teams turned out to be less effective in the first run of the experiment, because of the poor quality of co-ordination and database structure developed by the co-ordination team. This improved considerably during the second run. This indicates that the efficiency of competitive development depends strongly on the quality of co-ordination, and in particular on careful database design;
  - Team separation eliminated clashes on the changes to database structures, as well as error propagation resulting from the reuse of poorly verified software.

5 Discussion: The Impact of Competitive Development

The idea of a number of teams working together on the same project under some sort of co-ordination is obviously nothing new. It can easily be related to current achievements of global software development [7], [8], which raises issues of efficient communication and co-ordination between geographically separated teams.

However, the idea of competitive development is based on quite opposite assumptions: instead of striving to resolve the challenge of multi-party communication between separate teams [6], it assumes that this should not take place, as all the conflicts should be resolved by the co-ordinator. This obviously makes sense only for strongly cohesive and large enough modules like applications or even subsystems, otherwise the co-ordinator has the heaviest task and has to make the greatest effort.

The proposed approach assumes that the interests of the organisation commissioning the system and the interests of the developing companies are partially conflicting and only partially consistent. This is far removed from the agile approaches, which assume close co-operation between the parties and implicit sharing of the same interests. This synergy does not necessarily take place in the case of large-scale software systems.

Competitive development apparently ignores the advantages of reusing code, though this applies only to the reuse of code created during systems development, which in many cases turns out to be error prone. At the same time, each of the separate development teams can reuse proven COTS or libraries.

6 The Savings from Software Demonopolisation

Competitive development is designed for the development and evolution of large scale software systems. Modern IT systems evolve perpetually throughout their whole lifetime, never reaching a stable state. Market surveys show that the cost of system maintenance and evolution incurred during the entire life cycle is often twice or even three times as high as the initial cost of system construction. Table 2 contains a sample of large-scale software systems developed for public organisations in Poland over the past 10 years. They belong to the group of the largest systems developed in our country and across Central and Eastern Europe, and are all affected by the software monopolisation syndrome.

Table 2. A Sample of Large-scale Software Systems Developed for Public Organisations in Poland

Name of the system or application domain	Approximate Development and Maintenance Costs Incurred so far (in USD)
Comprehensive System for Social Security	$1.2 bln
Integrated Administration and Control System (IACS) /Agricultural Aid Programmes Administration and Supervision (IACS)	$0.6 – $1 bln
Cars and driving licences register	$80 mln (development only)
Energy Market	$50 mln
EU Aid Management (SIMIK)	$6 mln

Different internet sources, like [9], indicate that increased levels of competition can make prices in groceries cheaper even by about 10%, for electronic devices it can even be 30% off [10]. As the profit margins assumed in the software industry are rather high, due to the high level of risk, we expect that it should be possible to save at least 20% of the development cost if competition is properly ensured. In the case of the largest systems mentioned in table. 1, i.e. systems Nos 1 and 2, it means that at least $20 mln could be saved each year on maintaining each of these systems (assuming an annual maintenance cost of $100 mln for each of these systems).

7 Conclusion

Large-scale software systems are a kind of natural monopoly (like water supply or sewage system) in the genre of IT. Recent history shows that some natural monopolies were broken by splitting up large organisations into a number of smaller ones, or through the introduction of some sophisticated co-ordination of common resources usage as in the case of the electric energy market. The concept of competitive development is assumed to be the first step toward breaking monopolies on the development of large scale software systems.

Competitive development supports development process diversity, as different processes can co-exist in the development of the whole system. This approach is radically different to the existing ones, as it assumes conflicting interests rather than consistent interests of the parties taking part in the development of large-scale software systems.

The further development of this concept should include mechanisms of efficient co-ordination of independent developments, demonopolisation activities undertaken by the co-ordinator, as well as the techniques of efficient management of common databases. Obviously, further validation on a greater number of development cases is needed and is planned for the future.

Acknowledgement. This research has been supported by a grant from the Ministry of Science and Higher Education of the Republic of Poland under grant No 5321/B/T02/2010/39.

References

1. Simon, H.A.: The Architecture of Complexity. Proceedings of the American Philosophical Society 106(6), 467–482 (1962)
2. Ethiraj, S.K., Levinthal, D., Roy, R.R.: The Dual Role of Modularity: Innovation and Imitation. Management Science 54(5), 939–955 (2008)
3. Ethiraj, S.K., Levinthal, D.: Modularity and Innovation in Complex Systems. Management Science 50(2), 159–173 (2004)
4. Corbi, T.A.: Program understanding: Challenge for the 1990s. IBM Systems Journal 28(2), 294–306 (1989)
5. Ali Babar, M., et al.: Architecture knowledge management. Theory and Practice. Springer, Heidelberg (2009)
6. Lanubile, F., Ebert, C., Prikladnicki, R., Vizcaino, A.: Collaboration Tools for Global Software Engineering. IEEE Software 27(2), 52–55 (2010)

7. Damian, D., Moitra, D.: Global Software Development: How Far Have We Come? IEEE Software 23(5), 17–19 (2006)
8. Boden, A., Nett, B., Wulf, V.: Operational and Strategic Learning in Global Software Development. IEEE Software 27(6), 58–65 (2010)
9. Jarman, M.: Grocery competition brings shopper savings, http://www.azcentral.com/community/westvalley/articles/2010/10/12/20101012arizona-food-prices-decline.html
10. Tough Competition Brings Down Smart Phone Prices, http://www.iwabs.com/content/tough-competition-brings-down-smart-phone-prices