How Program Size Affects Construction

Stefan Ast

Software Construction Techniques: Writing Good Code
University of Tübingen

Abstract. Program size of software development projects varies greatly. Therefore there are different methods used for organization and development. Small projects usually use casual and instinctive methods. Large projects usually use rigorous and carefully planned methods. The size determines how difficult communication is, what type of errors to expect, how productivity will be affected and how big the share of single development activities will be. The main goal should be to keep big projects manageable and to avoid too much overhead for a small project. There is a lightweight approach for small projects and a more formal approach for big projects. The key is to find the right balance between them.

1 Introduction

It is easy to misjudge the effect project size has on a software construction project. A programmer who is used to work on small projects might have difficulties with the complexity of a medium to large project. A programmer who has only worked on large projects probably uses approaches that are too formal on a small project. This can even lead to failure of a promising project. So it is important that the programmer is aware of different approaches and methodologies for different project sizes [1].

2 How Program Size Affects Construction

2.1 Project Size

One way to measure project size is team size. Around 55% of all software development projects have team sizes of one to ten persons. Only 10% of all projects have a team size greater than 50. But there are also other ways to define project size. It is possible to use the lines of code of the program or the quality and complexity of the final software product. To measure project size in quality and complexity it is important to know the kind of the final software product. According to McConnel [2] there are four kinds of programs which are different in complexity and have a varying demand for quality. The simplest software project is the development of a "program" that's used only by the original programmer or a few other persons. A more sophisticated program is a "software product". It is used by customers. It has to be tested, documented and maintained. It is usually three times more expensive to develop a software product than it is to develop a simple program. The next level is the development of a group of programs that work together. This is called a “software system”. Interfaces between the pieces of the system have to be developed and the pieces have to be integrated. It is usually three times more expensive to develop a software system than it is to develop a simple program. A “system product” has the polish of a product but is a software system. It is usually nine times as expensive to develop as a simple program.

2.2 Communication

As project size increases, communication becomes more difficult. Every member of the project team theoretically has a communication path to every other member. If there are three programmers there are three communication paths. If there are ten programmers there are already 45 communication paths [3].

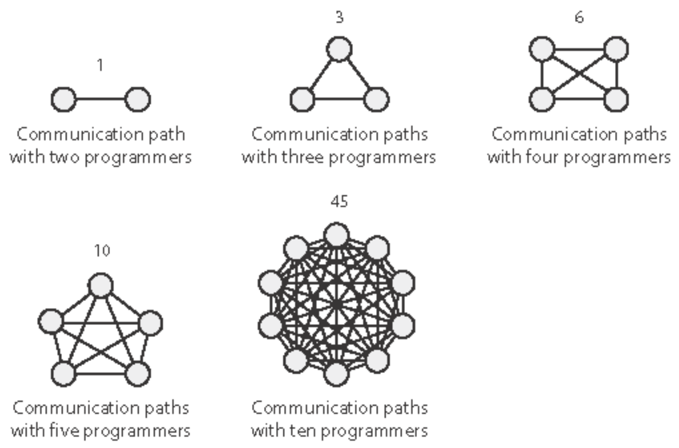
1
Communication path with two programmers
3
Communication paths with three programmers
6
Communication paths with four programmers
10
Communication paths with five programmers
45
Communication paths with ten programmers

Fig. 1. Number of communication paths increases faster than number of people on the team [4].

If the number of communication paths is high, the time spent for communication is also high and the opportunities for communication mistakes increases. Big projects have to use organizational techniques that streamline communication or limit it in a sensible way. The typical way to achieve this is to formalize it in documents [5].

2.3 Errors

On small projects 75% of all errors are construction errors. The biggest influence on program quality is often the skill of the programmer. At bigger projects a larger percentage of errors are caused by mistakes in requirements and design. The reason is that more requirement developments and architectural design work is required on large projects so errors in these areas are more common [6].

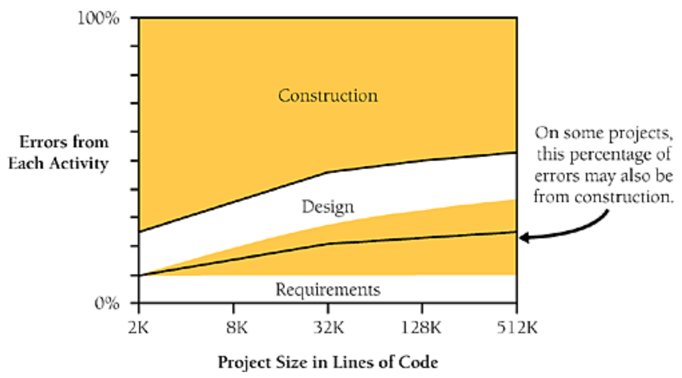

Fig. 2. With increasing program size the type of errors changes [7].

2.4 Productivity

On small projects productivity can be two to three times as high as productivity on large projects or five to ten times from the smallest to the largest. In projects with small size the skill of the individual programmer determines productivity. With increasing project size the team size and organization get more influence. But determining productivity is not easy. It depends on the kind of software, personnel quality, programming language, methodology, product complexity, programming environ-ment, tool support or how “lines of code” are counted [8].

2.5 Development Activities

Activities

As project size increases construction becomes a smaller part of the total effort. Small Projects are dominated by construction activities until a percentage of 65% of total development time. At medium sized projects this number falls to 50%. Larger projects need more architecture, integration and system testing [9]. Construction work like detailed design, coding, debugging and unit testing scales up proportionately. But other activities scale up faster. This includes: communication, planning, management, requirements development, system functional design, inter-face design and specification, architecture, integration, defect removal, system testing and document production. There are also activities which are important on small and big projects like disciplined coding practices, design and code inspections by other developers, good tool support and use of high-level languages [10].

Methodology

There are more and more large software systems in development and requirements change rapidly. Traditional software development is characterized by engineering and process improvement thinking. It focuses on plans and architectures. But this can lead to spending more time writing documents than producing software. Therefore so-called agile methods have been developed. These represent a different perspective and they are characterized by short cycle times, close customer involvement and an adaptive mind set. This generally means producing software with less documentation under conditions of rapid change to the satisfaction of clients. These two approaches can be combined into a hybrid approach to try to get the best out of both worlds [11].

Traditional plan-driven methods use a systematic engineering approach. It is characterized by defined requirements, standardized process management and thorough documentation. This includes detailed plans, activities, workflow, roles and responsbilities. Because of the comparability and repeatability of this standardization personnel can be moved quickly between products and loss of key personnel can be absorbed. But there are also problems. Innovation might be blocked, the focus can shift from the product to the process and a group of people is needed for managing and controlling. Examples for plan-driven approaches are Capability Maturity Model Software (SW-CMM), CMM Integration (CMMI) and Personal Software Process/Team Software Process (PSP/TSP) [12].

Agile methods see programming as a craft rather than an industrial process. The rapid change of the internet-based economy demands flexibility and speed from software developers. Fulfilling user expectations is more important than well-written code and documentation. Agile Methods usually have several iteration cycles and do not deliver the entire product at once. They rely on self-organizing teams that define processes, principles and work structures during the project. They need close relationship with the users/customers because they get feedback frequently and evolve the system based on that feedback step by step until the product is finished. Agile Methods need highly motivated, knowledgeable team members. They have to be willing to work closely with other programmers. Examples of agile methods are eXtreme Programming (XP), Adaptive software Development (ASD) and Scrum [13].

Traditional plan-driven methods usually are used on large complex systems who are often safety-critical and have to be highly reliable. Requirements have to be stable and the environment has to be predictable. Agile methods are working better with smaller teams, more volatile requirements and environments and close relationship with customers and users. To find the right balance between these two approaches the management has to consider the team members personalities and experience, the complexity of the project and other factors like budget, schedule and risk [14].

3 Related Work

This paper is mainly a summary of Chapter 27 from Steve McConnels “Code Com-plete, Second Edition”. Microsoft Press, 2004. Also Barry Boehms and Richard Turners “Balancing Agility and Discipline: A Guide for the Perplexed” has been used to describe the different methodologies for different program sizes in more detail.

4 Conclusion

Large projects will need to work harder than small projects to achieve the project goals. The more people you have to coordinate the more formal the documentation has to be. The plans, documentation and processes make better communication and coordination across large groups possible. The disadvantage is that it requires a big amount of work to manage it. So productivity will be lower on big projects because more work is needed for organizational activities. There will be more errors on big projects but with a greater amount of design and requirement development errors instead of construction errors. Using agile methodology at the beginning and scaling it up as project size increases usually works better than scaling formal methods down. For small projects with a less than forty team members it is usually the best way to work with agile processes. Bigger teams have to deal with the increasingly complex interactions of the project’s elements so they need more traditional plans, processes and documentation. It is important to balance agile and formal methods so they fit the project size.

References

1. Steve McConnel: Code Complete, Second Edition. Microsoft Press, 2004, p. 649
2. Steve McConnel: Code Complete, Second Edition. Microsoft Press, 2004, p. 656
3. Steve McConnel: Code Complete, Second Edition. Microsoft Press, 2004, p. 657
4. Steve McConnel: Code Complete, Second Edition. Microsoft Press, 2004, p. 650
5. Steve McConnel: Code Complete, Second Edition. Microsoft Press, 2004, p. 650
6. Steve McConnel: Code Complete, Second Edition. Microsoft Press, 2004, p. 652
7. Steve McConnel: Code Complete, Second Edition. Microsoft Press, 2004, p. 652
8. Steve McConnel: Code Complete, Second Edition. Microsoft Press, 2004, p. 653
9. Steve McConnel: Code Complete, Second Edition. Microsoft Press, 2004, p. 654
10. Steve McConnel: Code Complete, Second Edition. Microsoft Press, 2004, p. 654 - 655
11. Barry Boehm, Richard Turner: Balancing Agility and Discipline: A Guide for the Perplexed. Addison-Wesley, 2004, p. 2 - 5
12. Barry Boehm, Richard Turner: Balancing Agility and Discipline: A Guide for the Perplexed. Addison-Wesley, 2004, p. 9 - 16
13. Barry Boehm, Richard Turner: Balancing Agility and Discipline: A Guide for the Perplexed. Addison-Wesley, 2004, p. 16 - 22
14. Barry Boehm, Richard Turner: Balancing Agility and Discipline: A Guide for the Perplexed. Addison-Wesley, 2004, p. 22 - 24