Development of Packaged Software

IJsbrand Oudshoorn
Vrije Universiteit Amsterdam
student number: 1275208
e-mail: ipoudsho@cs.vu.nl
April 2004

Abstract

Packaged software is a significant part of the software market. The development of a software package is different from the development of tailor-made software. The differences are in the used Life Cycle Models, in the Requirements Engineering phase and in the Architecture. This article shows why some normally used methods don't fit in software package development and describes various solutions.

Introduction

Developing packaged software is different from developing tailor-made software. How to build tailor-made software is commonly taught. At the Vrije Universiteit students learn through the course Software Engineering the five phases (requirements engineering, design, implementation, testing and maintenance) and the standard life cycle models (waterfall, prototyping, incremental development, spiral model, etc.) [Vli00].
But there is more than tailor-made software. The the best known software products are not tailor-made. See for example Microsoft's products Windows and Office, these are standard products made for a large market.
This article will first describe what packaged software is and then discuss the differences in software engineering techniques. The discussed techniques are the Life Cycle Model, the Requirements Engineering phase and the Architecture.

What is Packaged Software

Packaged software is a software product that is released for and traded in a specific market. Other terms in use are product, standard, market-driven, shrink-wrapped and COTS[1] (Commercial Off-The-Shelf) software. The opposite of packaged software is tailor-made software, also called bespoke, custom-made or customer-specific software.

Packaged software is in that sense different from tailor-made software that it is developed for (in principle) thousands of customers instead of one single customer. [Car93] named five characteristics for software package development: 1) accommodate the "unknown customer", 2) address special difficulties in formalizing customer requirements, 3) support multiple platforms and multiple versions, 4) facilitate speed of development, adjust to the window of opportunity, and react to market pressures, 5) minimize incorporate risk. It's not in the scope of this article to go into the details of these characteristics but it should make clear what the important differences are.

Related Work

The most research and articles in the software engineering field are about the development of tailor-made software. These articles don't say it explicitly but the discussed methods don't always fit the world of the software package development.
Sjaak Brinkkemper[2] and Michiel van Genuchten[3] were guest editors for *Informatie*[4] and organised a special issue about product software. The articles were mainly business oriented. [Sol03] discusses briefly the role of the customer in the requirements engineering phase.
The requirements engineering is a field in which more research is done. The *International Conference on Product Focused Software Process Improvement, PROFES* and the *International Workshop on COTS and Product Software, RECOTS* are totally focused on the development of software products.

Differences in The Life Cycle Model

Continuous Development

Perhaps the most visible difference in the development process between building tailor-made software and building a software product is the maintenance

1 It says COTS software not COTS *based* software.

2 Vrije Universiteit Amsterdam
3 Technische Universiteit Eindhoven
4 Informatie is a magazine of the Dutch Ngi and the Belgian SAI, both national organisation for ICT-professionals

phase. At a certain time a tailor-made program is "finished", after that moment the maintenance phase starts. Packaged software is continuously under development and has in that sense no real maintenance phase. After finishing one release there will come another.
Because of the needs for continuous improving the product simple models like the waterfall and the prototyping model (see [Vli00] chapter 3 for the pros and cons of the different models) can not be used. Only the incremental development and the spiral model come near the needs of continuous development because of the possibility to go on for ever.

Example 1 - Microsoft's Life Cycle Model

Microsoft does not "adopt too many of the structured software-engineering practices commonly promoted by such organizations as the Software Engineering Institute and the International Standard Organization" [Cus97] and made a model of their own. Figure 1 contains a brief explanation of the used life cycle model "synch-and-stabilize".
The development phase named in this figure has a lot in common with the MoSCoW[5] approach. Development can be stopped after each closed sub-phase.

Planning Phase
- *create vision statement*, based on user input
- *define functionality*, based on vision, just enough to arrange teams
- *schedule and arrange teams*, per feature

Development Phase
- *implement basic functionality*, the most critical features, one third of the features
- *implement extra functionality*, a second third of the features
- *implement more functionality*, the least critical features, the last third of the features

in each of these steps the developers go through a complete cycle, thus including integration, testing and fixing

Stabilization Phase
- *internal testing*, in house testing
- *external testing*, the well known beta versions
- *release preparation*

Figure 1 – An overview of the synch-and-stabilize development approach as described in "How Microsoft builds software" [Cus97]

Example 2 - An alternative waterfall model

Repeating the waterfall model for every release gives a model for continuous development. A new requirements phase starts after finishing the last release. The release cycle time, the time between two releases, is in this case the sum of the times of the different phases. This is really not an efficient way of working, certainly not when release cycle time is important (as it is for many software package firms).
A more efficient way is shifting those waterfalls in each other and arrange specialised teams for each phase. These specialised teams can work in parallel as shown in figure 2. This alternative model also fits better to the reality where customers can formulate their requirements at any time.
Figure 2 uses the following phases:
Requirements Engineering – See the next chapter.
Release decision – A few days before the start of the next development cycle it's needed to make a decision of which features will be in the next release.
Development – In this phase the chosen features will be implemented and tested. This phase can be implemented as a cycle model with steps like choosing, implementing and testing a requirement. This can be done of course in parallel per feature.
Preparation for distribution – If the product is fully tested and declared ready for distribution it's time to build the setup and burn the cd-roms.
Sales & Distribution – The final goal.
For each bug report there is a decision to make: must it be solved in the next release (the release currently under development, like the critical bug in the figure) or is it not that important and will it be decided in the next release decision phase.
Working according this alternative life cycle model gives a release cycle time that's equal to the time of the development phase.

Conclusions

For developing a software product it is needed to use another life cycle model than for developing a customer-specific program. By using for instance the described alternative waterfall model it's possible to decrease the release cycle time.

Differences in the Requirements Engineering phase

The main difference for requirements gathering is the source. The source is clear when developing a customer-specific program, because there is only one customer. For packaged software it's not that

5 MoSCoW, Must have, Should have, Could have, Won't have

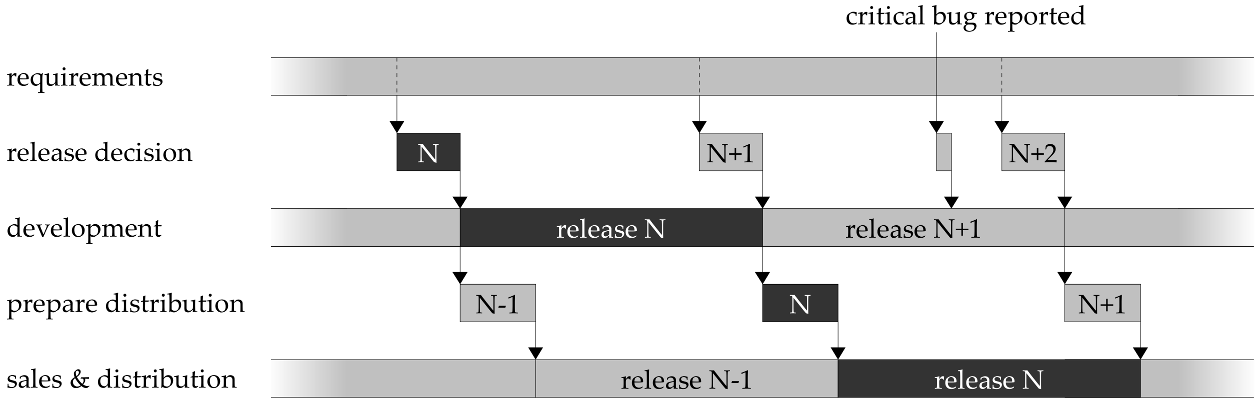

Figure 2 – A life cycle model for developing a software product

clear. To solve this problem [Sol03] suggests to interpret the marketing department as an internal customer. This is unfortunately not really a solution. Maybe it is for the development department, but not for the whole company. How do the marketeers get the requirements? It is just carrying over the problem. The solution must be found in different elicitation techniques.

Swedish research

To study the differences in elicitation techniques Swedish researchers has interviewed seven requirements engineers. Preliminary results are presented in [Dah03]. The report describes three characteristics based on current literature:

1) The primary goal is time-to-market. - This hardly depends on the competition in the market and is normally not true during the development of the first release.
2) Requirements are invented. - The requirements needed for the first release are often invented.
3) Requirements are rarely written. - The requirements are normally written down to serve as contract between the customer and the producer. There is no need for such a contract when developing a software product.

That requirements are rarely written in a formal Requirements Documents will be certainly true. But the constant flow of requirements needs to be registered and managed. Registering will be done by writing those down, on paper[6], in a document or in a database (so they are written).

Elicitation Techniques

Because there is no close relation between developers and customers (if there is already a customer) in software package development a lot of the in [Vli00] described elicitation techniques cannot be used as normal. Most customers of packaged software have no interest in spending a lot of time in helping the software firm. Joint Application Development (JAD), brainstorming and interviewing are time consuming techniques and can therefore only be used when there is a kind customer who can make time for this. These techniques are unfortunately not that powerful as they are in customer-specific development because it is always needed to keep the general use and purpose of the program in mind. Some features are enormously important for one customer but of no use for the rest. Or even worse, features can conflict with each other.

Here follows a list of well known elicitation techniques with comments of the usefulness in software package development.

Brainstorming with customers – Typical for tailor-made software.

Brainstorming with developers – Can be used for inventing requirements.

JAD – Too customer specific.

Form Analysis – Can be very useful if forms of different (potential) customers can be arranged.

Interviewing – Is customer specific. Can be used for specific questions about general tasks.

Questionnaires – Is a useful tool when there are enough customers.

Market Analysis – Looking at the products and/or folders of the competitors is useful for defining the base functionality. It's also useful for the marketeers to know what makes their product unique.

Two techniques I found useful are:

– reading the professional literature of the user domain; and
– interchanging people of the development and marketing staff (if possible).

Reading user's literature is a task for the requirements engineer and gives a view of coming events like new laws and rules. Interchanging people is only possible if there are employees with multiple disciplines.

6 eXtreme Programming Story Cards for example

Good practices

A useful list of good practices on requirements engineering focused on packaged software can be found in chapter 4 of [Saw99].

Conclusion

The main difference is the source of the requirements. For tailor-made software there is one customer. Software packages have (potential) thousands of customers. Most of the normally described elicitation techniques can not be used for gathering requirements for a software product. The most useful techniques are focused on the market in which the users operate.

Differences in the Architecture

There is not really a thick line between architectures that are good and that are bad for the development of packaged software. But there are a few aspects that are more important for developing a software package than for building a tailor-made program.

Future proof

The most important characteristic is that the architecture needs to be future proof. As [KLV02] defined: *The software architecture of deployed software is determined by those aspects that are the hardest to change.* And because software products are expected to have a long life time it's important to have a future proof architecture.
Future proof means in this sense that the architecture must be able to easily adapt to new requirements. Not only to adapt to new user requirements but also to the always changing environment like new releases of the operating systems and changes in the used database management system.

Independent

This characteristic is a kind of spin-off of the future proof aspect described above. The different components in the architecture need to be as independent as possible (weak coupling). This is for example useful when it's needed to change the used database management system or for porting the application to another operating system. It is almost needed when selling different versions of one product. For example a stand-alone version, a multi-user version for small businesses and a multi-user version for big enterprises need different database systems.
This last aspect is specific for packaged software. For tailor-made software it's known what the environment will be. For software packages the environment can be different for every customer.

Tailor-made	Packaged
Source	
clear	unclear
one customer	- marketing depart. - current users - consultants
Responsible / Financial Risk	
customer	software firm
Requirements Document	
used as contract	for internal use
is a text document	is a database
Elicitation Techniques	
JAD -	- market analysis
interviewing -	- questionnaires
brainstorming - (end-users)	- brainstorming (developers, marketeers, consultants)
task analysis -	

Figure 3 – An overview of the differences in Requirements Engineering

Example – The Product Line approach

The most useful architectural approach I saw so far is the product line. The Software Engineering Institute defines a product line as follows: *A product line is defined as a group of products sharing a common, managed set of features that satisfy specific needs of a selected market or mission*. This definition uses the terms "products" and "selected market" which already aims at the development of software packages. [SEI97] describes factors involved in product line practices and issues in the areas of architecture, organization and business.
The product line approach is a good way for defining an architecture for software products. The highly structured definitions and the weak coupling of different components make it possible to maintain (relatively) easy different versions of the same product.
In this approach it is important to have an architect that has a good overview over all the products in the family, that the architecture will be enforced, and that every developer is aware of the architecture and the meaning of the components in which he is working.

Conclusion

For the development of a software package it is needed to have a future proof architecture. The architecture needs to define clearly distinguishable components for achieving a future proof architecture. A usable approach for defining an architecture is the product line approach.

Evaluation

The specific problems for building packaged software instead of tailor made software were:

- *the continuous development*, products are always under development, in this endless cycles there exists no maintenance phase; and
- *the unknown customer*, thousands of customers instead of one.

The continuous development asks for a different life cycle model and architecture. The unknown user needs another way for gathering the requirements.

Life Cycle Model

The life cycle models used in general doesn't cover the development of software packages. The discussed alternative waterfall model is a real solution for continuous development.

Requirements Engineering

Gathering the requirements for a software product is more difficult than for a customer-specific program. This is because of the unclear source. The discussed solutions are just modified old techniques and not real solutions. A lot of research can be done in this field. But actually it is only a problem in the start-up phase of a product, when the first release is finished the users will come with questions and requirements. After the first release it is a matter of managing the requirements and keep an eye on the products of the competitors and the changes in the user's environment like new laws and rules.

Architecture

In architecture there is no real problem. The only important property of the architecture is that it needs to be future proof.

Conclusion

Developing a software product is different from building a custom-made system. The main differences are in the used life cycle model and the requirements engineering phase. By using an appropriate life cycle model it is even possible to decrease the release cycle time. In the architecture phase it is important to define a future proof architecture. Future proof in the means that it can easily be adapted to changes in user requirements and environment.

References

[Car93] Erran Carmel (1993). A discussion of special characteristics for software package development life cycle models. *ACM SIGSOFT Software Engineering Notes*, vol. 18, issue 2, pages 23-24. ACM Press, New York, USA.

[Cus97] Michael A. Cusumano and Richard W. Selby (1997). How Microsoft builds software. *Communications of the ACM*, vol. 40, issue 6, pages 53-61. ACM Press, New York, USA.

[Dah03] Åsa G. Dahlstedt, Lena Karlsson, Anne Persson, Johan Natt och Dag and Björn Regnell (2003). Market-Driven Requirements Engineering Processes for Software Products – a Report on Current Practices. *Proceedings of the International Workshop on COTS and Product Software (RECOTS'03).* Monterey Bay, California, USA.

[KLV02] Steven Klusener, Ralf Lämmel and Chris Verhoef (2002). *Architectural Modifications to Deployed Software*, revision February 2004, unpublished. Vrije Universiteit Amsterdam, The Netherlands.

[Saw99] Pete Sawyer, Ian Sommerville and Gerald Kotonya (1999). Improving Market-Driven RE Processes. *International Conference on Product Focused Software Process Improvement (PROFES'99)*, pages 222-236. Technical Research Centre of Finland (VTT), Oulu, Finland.

[SEI97] Len Bass, Paul Clements, Sholom Cohen, Linda Northrop and James Withey (1997). *Product Line Practice Workshop Report*, CMU/SEI-97-TR-003, June 1997. Software Engineering Institute, Pittsburg, USA.

[Sol03] Rini van Solingen (2003). De klant bepaalt. *Informatie*, jaargang 45, oktober 2003. NGI / Ten Hagen & Stam b.v., Den Haag, The Netherlands.

[Vli00] Hans van Vliet (2000). *Software Engineering Principles and Practice*, second edition. ISBN: 0-471-97508-7. John Wiley & Sons, Inc., New York, USA.