Petri Kettunen and Maarit Laanti. 2006. How to steer an embedded software project: tactics for selecting agile software process models. International Journal of Agile Manufacturing, volume 9, number 1, pages 59-77.

© 2006 International Society for Agile Manufacturing (ISAM)

Reprinted by permission of International Society for Agile Manufacturing.

IJAM International Journal of Agile Manufacturing

How to Steer an Embedded Software Project: Tactics for Selecting Agile Software Process Models

Petri Kettunen, Maarit Laanti
Nokia Corporation, P.O. Box 301, 00045 NOKIA GROUP, FINLAND
E-mail: petri.kettunen@nokia.com, maarit.laanti@nokia.com

Abstract: Large, modern, new product developments (NPD) typically are characterized by many uncertainties and frequent changes. Often, the embedded software development projects working on such products face many problems compared to traditional, placid project environments. One of the major project management decisions is the selection of the project's software process model. An appropriate agile process model could help in coping with the challenges and even could prevent many potential project risks and problems. On the other hand, an unsuitable process choice often causes additional problems. This industrial paper investigates the agile software process model selection in the context of large, market-driven, embedded software product development for new telecommunications equipment. Based on a quasi-formal comparison of publicly known agile software process models, including XP, ASD, Scrum, FDD and RUP, we propose a process model selection frame, which the project manager can use as a systematic guide for (re)choosing the project's process model. A novel feature of this comparative selection model is that we make the comparison against typical software project problem issues. Some past real-life project case examples are examined against this model. The selection matrix expresses how different agile process models answer to different questions, and indeed there is not a single process model that can answer all questions. On the contrary, some of the seeds to the project problems are in the process models themselves, and no agile process model is a silver-bullet solution. Nevertheless, being conscious of these problems and pitfalls when steering a project enables the project manager to master the situation and to take advantage of agile process models.

Key Words: Software Project Management, Agile Software Process Models, Risk Management, Embedded Systems, New Product Development.

1. Introduction

Managing modern industrial product development projects successfully requires situation-aware control of possible and inevitable trouble, taking the anticipated and even unexpected situational conditions into account [20]. Often, the embedded software development projects working on such emerging products face many problems compared to traditional, placid project environments [27].

A powerful tool any project manager might have to cope with in such challenges is the command of initially choosing and – if necessary – later revising the software process model [7, 25]. Recently, a new software process (methodology) philosophy of agility has been advocated as a potential solution to such turbulent software project cases. Those agile software process models emphasize certain key practices of project mission elaboration, project initiation, short iterative (time-boxed) development cycles, constant feedback, and customer intimacy [17]. The underlying principles have been stated in the well-known "Agile Manifesto" [40].

Many different published agile software development methodologies are available: e.g., eXtreme Programming (XP) [4], Scrum [33], and Adaptive Software Development (ASD) [16] – just to name a few. The problem is now for the project manager to select an appropriate process model among the many alternatives. In this paper, we present a systematic approach regarding when it would be wise to use a certain agile software process model under certain project conditions, and why. Specifically, we are interested in investigating how different agile process models cope with different project problems. The purpose is to provide pragmatic aids for practicing project managers by combining and distilling knowledge from a variety of literature sources coupled with our practical experience.

In this study, we focus on one specific type of software project, namely, market-driven development of embedded software for telecommunications products (e.g., mobile phones, radio network elements). Even within this category there are many different project types, such as completely new product development, new features development for existing products and derivatives, and platform developments. Here, we limit ourselves to the first type, i.e., the software development for a whole new product. We have held that limitation to new product development, since we feel that there is much more freedom for the project manager to choose the initial software process used, than to make changes to one that already has been established and used for many past software releases. Neither of the two limitations, however, is exclusionary nor definitive as to the usage of our guidelines – the reader is encouraged to explore the suitability to her own application area.

The rest of the paper is organized as follows: Chapter 2 explores the background and related work, and sets the exact research question. Chapter 3 then describes our solution ideas, while Chapter 4 evaluates them. Finally, Chapter 5 offers concluding remarks and outlines further research ideas.

2. Many Agile Software Process Model Alternatives

2.1 Software Process Models and Project Problems

In this paper, we define "software process model" broadly so that it includes all the project life-cycle activities of project planning, tracking, and requirements management, as well as the actual software construction and release. Sometimes a more holistic concept of "methodology" is used in this context [9]. Software process, thus, is more than just a software development life cycle. "Agility" refers to the ability of the process to support responsive software product development, ultimately for the business success [17].

During the past few years, many agile software development process models have been proposed. Many research investigations and numerous software engineering guidebooks compare and contrast the different models. See, for example, [1, 3, 6-7, 9, 19, 25, 38]. Those different investigations use various different comparison viewpoints of the process models, such as

- universal prescription vs. situational adaptation [1];
- process definition flexibility (accommodating change) [21];
- primary objectives (e.g., rapid value vs. high assurance) [6];
- size, criticality, project priorities [9];
- cycles of operation (number and length), formalism ("ceremony") [25];
- relevance to software engineering (construction) vs. management [19];
- life-cycle coverage [3];
- people factors [38]; and
- multidimensional home ground profiles [7].

Note also that similar aspects of software process models have been investigated already earlier, although the term "agile process" was not yet coined explicitly at the time of the writing. See, for example, [28, 35].

In modern software product development environments, the basic premises and assumptions of the traditional process models have been stretched so much that many such classic models have become partially unsuitable. In addition, the growing understanding of innovation patterns and organizational learning has influenced software engineering management (knowledge management). Because of many unknowns and uncertainties coupled with ambitious time-to-market goals, basic serial document-driven development is often not feasible. Modern business pressures and technological advances require responsive, last-minute changes in the product contents. Agile software process models address such aspects in particular.

The current trend in software process model development advocates more adaptable and flexible ways of working, i.e., moving from rigid, all-defining, huge organizational processes toward sketched, tailorable, agile processes. The typical way of working is to give only a few of the most essential practices to the project – more like a process skeleton – in which the practices gradually can be added. The mental model here is merely to let the project determine the practices, when ready to take them into use, rather than following a well-set rigid model [14, 16/Ch. 8]. The underlying premise is that since uncertainty and frequent changes are inherent in current projects, it is typically not reasonable to lock the project's process in a prescriptive way.

Embedded systems have, in addition, certain intrinsic software project problems [37]. Software developers often must understand interdisciplinary product application domain knowledge. Systems engineering, then, is a key activity. Note that in complex product systems (e.g., mobile phones), there are often many profoundly different types of embedded software sub-systems ranging from real-time hardware drivers to sophisticated man-machine interfaces. The most recognized software models are pure models in the sense that the focus is only on software. Models used in embedded software development are often variations of these. Most existing process models, however, can be tuned, to some extent, to real-time embedded software projects by taking into account the systems engineering and hardware dependencies.

In industrial new product development environments, there are also many limiting business constraints to be taken into account [15, 27]. The embedded software project teams working in such environments often face many sources of turbulence, as illustrated in Fig. 1.

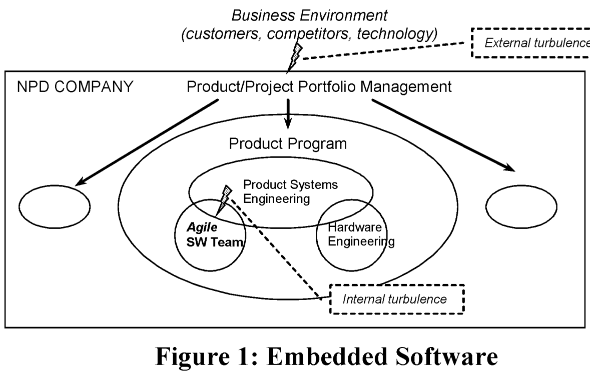


Figure 1: Embedded Software Project Team NPD Context.

The company, responding to emerging and fluctuating market needs, has to manage its product development portfolio accordingly. This may consecutively introduce various changes to the embedded software project teams (e.g., product features, release schedules, project resourcing). In addition, the other internal parts of the product development program (e.g., concurrent hardware engineering) may cause changes to the software part.

The question is now for a practicing software project manager to choose an appropriate agile process model for her particular project, taking into account the current and anticipated problems of the project. To the best of our knowledge, none of the published investigations cited above provides comprehensive guides for such purposes from that point of view. This is what we want to address.

2.2 Research Question

Based on the background in Ch. 2.1, we now offer the following specific question:

- *How do different agile process models respond to different project problems faced in turbulent environments (if at all)?*

The challenge is for the software project manager to find an appropriate agile process model among the many different alternatives, knowing how the selected model works under given project problem conditions [13]. Our aim here is to offer pragmatic aids for doing this in a systematic way, preventing the basic problems of selecting a fundamentally wrong model ("Lifecycle Malpractice"), or even not choosing any definite process model at all [8, 28/Ch. 7]. By making conscious choices, the project manager also can avoid any inherent disadvantages of the process model. She thus can avoid typical project problems by selecting an appropriate agile software process model, based on the project situational factors, realizing how the process model prevents particular problems from happening, or helps in mitigating them.

The rest of this paper proposes answers to that question. The research method for the question is a quasi-formal comparison based on distilling features [36]. As stated in Ch. 1, our special focus is embedded software development for new telecommunications products. In addition, we concentrate on large-scale projects, requiring tens of man-years of work effort.

Our underlying premise for investigating this question is that the process model is a significant productivity and quality factor for large software development projects. We do not argue, however, that it is the most important success factor. Often, people factors tend to be the ultimate keys [38]. Nevertheless, an efficient project management and development process has been recognized to be one typically characteristic of successful projects [22].

We earlier investigated similar questions with a broader scope elsewhere [23]. Here, our focus is on agile process models.

3. Tactics for Selecting the Agile Software Process Model

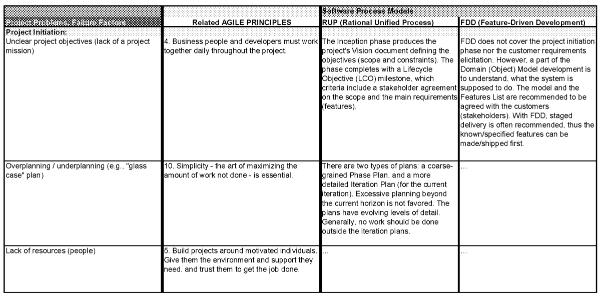
3.1 Software Process Model Comparison Matrix

Many agile process models are available, each having different characteristics and areas of suitability (home ground). The problem, then, is to find good matches with the actual project environment and current situational factors. No standardized solutions exist for this.

In order to help, we have composed an agile process model comparison matrix. Table 1 shows that structure. Table 2 is a sample excerpt of the actual matrix (top left-hand corner). See Appendix 1 for the complete matrix.

	Agile Software Process Model
Project Problem, Risk, Failure Factor	*How does this process model prevent that particular problem from happening, or help mitigate it (in the context of large embedded software projects)?*

Table 1: Software Process Model Comparison Matrix (Appendix 1) Structure.



		Software Process Models	
Project Problems, Failure Factors	Related AGILE PRINCIPLES	RUP (Rational Unified Process)	FDD (Feature-Driven Development)
Project Initiation:			
Unclear project objectives (lack of a project mission)	4. Business people and developers must work together daily throughout the project.	The Inception phase produces the project's Vision document defining the objectives (scope and constraints). The phase completes with a Lifecycle Objective (LCO) milestone, which criteria include a stakeholder agreement on the scope and the main requirements (features).	FDD does not cover the project initiation phase nor the customer requirements elicitation. However, a part of the Domain (Object) Model development is to understand, what the system is supposed to do. The model and the Features List are recommended to be agreed with the customers (stakeholders). With FDD, staged delivery is often recommended, thus the known/specified features can be made/shipped first.
Overplanning / underplanning (e.g., "glass case" plan)	10. Simplicity - the art of maximizing the amount of work not done - is essential.	There are two types of plans: a coarse-grained Phase Plan, and a more detailed Iteration Plan (for the current iteration). Excessive planning beyond the current horizon is not favored. The plans have evolving levels of detail. Generally, no work should be done outside the iteration plans.	...
Lack of resources (people)	5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.	...	...

Table 2: Software Process Model Comparison Matrix (Appendix 1) Example.

This matrix (Appendix 1) is basically a comparative analysis of different agile software process models. A notable feature of the matrix is that we have based the comparison on how well (if at all) each process model tackles typical problems of large embedded software projects. The reader is assumed to be familiar with the basics of the models in order to be able to understand the analysis points.

Note that the matrix (Appendix 1) is by no means an all-encompassing directory of agile software process models or potential project problems. The matrix has, in principle, been composed as follows.

We have selected the process model alternatives based on a literature survey (see Ch. 2.1), as well as on our own experience with large embedded software development projects. The idea is to cover the most well-known and widely used agile models. Currently, our matrix includes the following process models (columns): Rational Unified Process (RUP) [24], Feature-Driven Development (FDD) [30], Adaptive Software Development (ASD) [16], eXtreme Programming (XP) [4], and Scrum [33]. We also have included an antimodel titled "hacking" for contrasting purposes. RUP generally is not advocated as a pure agile process model, but since it is possible to use it in a lightweight way, we have included it as well [18]. Note that many other agile methodologies have been proposed. See, for example, [7, 17, 25].

No generally accepted scale of "agility" exists, although some informal ratings have been suggested [7, 10]. Therefore, the order of the process models (columns) in our matrix is not strictly defined, but we put RUP to the leftmost end, based on its traditional background, and "hacking" to the other end, following Boehm's spectrum [6].

For the comparison points, we have distilled distinct project problem areas and risk factors, based on well-known investigations (for example, [5, 22, 34]), coupled with our own large embedded software project experiences. Currently, our matrix includes some 50 problem items (rows). They incorporate, for example, the classic Boehm's risk list [5]. The rows are grouped according to the project life cycle: project initiation, execution, and completion (see the leftmost column of Table 2). The idea here is to cover a wide range of such essential project factors, which make a clear difference between the models in the context of large embedded software projects. Again, we acknowledge that other factors could have been included, and thus, that different orderings would be possible. Currently, the matrix includes mostly traditional project problem factors, but with agile process models, certain problem areas could be emphasized more.

Most (if not all) agile methodologies advocate at least some of the same common principles [40]. Therefore, we have included one column into our matrix reflecting how each project problem could be tackled in general (see column titled *Related AGILE PRINCIPLES* in Table 2). Note, however, that this reflection is just for reference purposes, since those agile principles have been formulated in quite a general way.

In addition, we have compiled an accompanying key point table of each agile process model's home ground, drawbacks, and typical pitfalls. Table 3 shows the outline (Appendix 2). Assuming that the reader is familiar with each process model in general, this summary serves as a quick reminder of notable remarks. Considering embedded systems, it summarizes the applicability of each process model for large embedded software projects. Notably current agile process models do not specifically address embedded software development [32].

	Agile Software Process Model
Home ground	*Most applicable project environment(s) – "sweet spot"*
Consequences, Side-effects, Drawbacks:	▨
Scope	*Coverage of the model (project life-cycle activities)*
Nature	*Methodological characteristics*
Advantages	*Key benefits*
Constraints, Disadvantages	*Limitations and disadvantages, prerequisites*
Cautions!	*Significant risks and pitfalls*
Notes	*Miscellaneous remarks*
EMBEDDED SYSTEMS	*Particular considerations for embedded software projects*

Table 3: Software Process Model Characteristics Matrix (Appendix 2) Structure.

3.2 Using the Comparison Matrix

A project manager can use the matrix (Appendix 1) described in Ch. 3.1 in the following two basic ways:

- *By columns:* Selecting the project's agile process model by comparing the basic alternatives according to the prevailing or anticipated project problem situation, i.e., by reflecting presented problem areas to her own known problem areas, and optimizing the best solutions, selecting a process model that supports it best.
- *By rows:* Evaluating how specific project problems can be tackled with different agile process model alternatives (if at all).

One even could give ratings of problems and the solutions each process model would provide – and calculate averages or weighted averages for each process model, and make analytical decisions on that basis.

In addition, certain color codes could be used in the matrix to highlight how well each process model tackles each problem. Such a colored cell map could provide a quick overview about the whole matrix.

4. Evaluation and Discussion

4.1 Validation

At the time of this writing, we are not ready to publish empirical case study data regarding the use of our agile process model selection matrix presented in Ch. 3 (Appendix 1). The following examples based on certain past real-life projects within Nokia, however, test some main points. Note that the examples have been sanitized for confidentiality reasons.

4.1.1 Example 1

PROBLEM: The project develops embedded software for a plug-in unit of a new product. Because of the time-to-market pressure, the unit hardware engineering and the software project are launched concurrently. The hardware design is based on new technology, and the hardware engineers cannot provide detailed frozen specifications of the hardware design to the software project until they have developed a series of prototype boards. The software project, therefore, must proceed, based on incomplete specifications and will be subject to many changes.

SUGGESTIONS: The synchronization with the hardware prototype development and the volatile hardware specifications are the keys here. ASD addresses such a situation by nature (see row titled *Incomplete Requirements / Specs (Poorly Defined Parts)*). Most of the agile models expect that the "customer" is able to define and clarify the software requirements. In this case, there could be an internal customer within the hardware team, and, for example, XP then could be applied. With short iterative cycles (e.g., Scrum), you should be able to accommodate the changes in a controlled manner (see row titled *Vague Milestones*). None of the agile process models, however, tackle especially well the problem with software project external dependencies (see row titled *Project External Dependencies Late and / or Imperfect (e.g., System Specs)*).

4.1.2 Example 2 [23]

PROBLEM: The product systems design is based on complex ASIC circuits and embedded software cooperation. The product development program initially is based on a waterfall model. At a late stage, however, when the first ASIC prototypes become available, a subtle ASIC design fault is discovered. Because of the tight product release schedule target, there is no time to redesign the ASIC. Instead, a non-trivial software workaround algorithm is specified, requiring considerable additional software design and testing efforts.

SUGGESTIONS: The initial choice of the waterfall model may have been wrong, if such a risk has been foreseeable from the beginning (see row titled *Underestimation of Project Size, Complexity*). None of the process models covered addresses such external dependency failures directly (see row titled *Project External Dependencies Late and / or Imperfect*), but such a change makes it difficult to continue with the waterfall model (see row titled *Project Redirected*). Agile, adaptive replanning is needed. One way of tackling this problem could be to have a new concurrent feature team to work on the additional functionality (FDD).

4.1.3 Example 3

PROBLEM: The project needed to develop a testing tool for an embedded product development with short time to market. The project had competent personnel, but it also was known that the project was unable to set up all the requirements correctly from the beginning. Moreover, there was some technical challenge due to the immature development environment. Because of all the uncertainty that the project had, it was clear that it would have been bound to fail if non-agile process models had been chosen for use.

SUGGESTIONS: The project originally had enjoyed using XP due to its good reputation and close time-to-market requirement (this was almost a Death March project). Pure XP, however, could not have been utilized fully because the developer's competences were not equal: Each developer had some special knowledge that was required for part of the final solution. Thus, the tasks were allocated to software developers based on features, as in FDD (c.f. row titled *Extreme Project*). Also, the XP practice of pair programming was abandoned partially because of the same reason: There was too little time to share the developers' special knowledge.

The project was experiencing some problems during the planning phase: First, requirements were changing a lot. This was tackled mainly by XP style having close contact with the customer (see row titled *Unstable (Volatile) Requirements, Continuous Requirement Changes*). Second, the project was missing some information from the product that the testing system was supposed to serve. Third, the project was forced to reuse some existing system solutions due to an extremely short project schedule.

The principal problems of this project case stemmed from the diversity of the domain knowledge and the technological environment. The main solution was to adopt FDD-style key practices. The project was quite successful, partially because the tasks were allocated correctly, based on competences and due to successful software reuse. In addition, the daily builds and their testing were automated successfully. The latter two problems mentioned above could not really have been avoided by any process model solutions alone.

4.2 Answering the Question

In Ch. 2.2, we offered a research question. We now evaluate our proposals presented in Ch. 3 against that question with respect to the literature reviews (Ch. 2.1).

How can the project manager steer the project through a problem-conscious selection of the project's software process model? What are the possible pitfalls of the selected agile process model? We have addressed this in the context of large embedded software projects by composing a software process selection matrix (Appendix 1).

Based on the limited set of retrospective project use cases examined in Ch. 4.1, we can conclude, that the process selection matrix works reasonably well on at least some typical embedded software project problem scenarios. It is certainly not a silver-bullet problem solver, however, and there are probably many situations in which the matrix cannot help so much. The usefulness depends much on the experience and assessment capabilities of the project manager.

How well do different agile process models work? Our comparison matrix suggests that there are certain project problem areas that none of the reviewed agile process models tackle especially well, such as the following (rows in Appendix 1):

- Lack of resources (people);
- New, immature software technology; and
- The project is large in size.

On the other hand, many typical problem areas tend to be addressed by every agile model, such as the following (rows in Appendix 1):

- Overplanning / underplanning;
- The march order: what should be done first and what next (phasing); and
- Unstable (volatile) requirements, continuous requirements changes.

The usefulness of each model thus depends on the actual project context and its prevailing problems. The key is to recognize the unique problems and goals of the particular project environment. Example 3 (Ch. 4.1.3) verifies the experience that many projects have reported: that, at least in larger projects, following the XP practices strictly is quite challenging. This leads to the thought that probably the problem space experienced with agile methods is somewhat different from the problem space described in literature based on the experience with traditional process models. When more experience is gathered utilizing agile methodologies, the agile-specific problems (and their possible solutions) also will become better known.

Our selection matrix does not provide new information about any agile process models nor project problem items, but the value of the matrix is in its systematic composition. The matrix contains distilled advice about the selected process models in a concise form. Notably, none of the reviewed investigations (Ch. 2.1) uses the viewpoint of the comparison based on project problem factors. Hull, *et al.,* compare a couple of generic process models with a rather similar approach on whether the process models support the avoidance of certain project failure factors or not [19]. Ould has used a similar viewpoint but with a much more limited scope [29/Ch. 4]. A recent work by Boehm and Turner includes an extensive comparison of many agile process models, based on a project's risk factors profile [7]. Larman categorizes some agile models based on their level of "ceremony", cyclic nature, and home ground on the "Cockburn scale" [25]. Typically, software process model comparisons are coarse-grained, indicating in general only the circumstances of when a certain model is suitable or not. We have elaborated this onto a more specific level.

Finally, embedded software development puts emphasis on certain process areas, as described in Ch. 2.1. The software process activities then must be focused upon accordingly [37]. We have highlighted this in the selection matrix by including a dedicated summary table for embedded software use (see Table 3). An even more thorough analysis, however, could be done. For example, Ronkainen and Abrahamsson have made a limited investigation in that direction [32]. Jaufman and Przewoznik compare a set of software process models, including many agile models with respect to their suitability for one specific branch of embedded systems (the automotive industry) [21]. Their particular concern is how well different process models accommodate change to new project conditions.

4.3 Application Possibilities

The main idea of using the selection matrix (Appendix 1) is, first, to select the agile process model by comparison, based on the problem issues (see Ch. 3.2). No reason exists, however, why the matrix could not be used in other ways, as well.

Another use of the matrix is to evaluate an ongoing project in case the process model already has been fixed (for external reasons). The project manager then can use the matrix to see how the process model behaves under certain problem conditions. In case there seem to be some weak points, she can start thinking about potential future mitigation strategies. Hence, the matrix helps in staying alert to those problems.

One also can use the matrix for training purposes. Although the matrix does not explain the basics of the agile process models, systematic reading of it may raise new thoughts about the project's potential risks and problems, or possibly useful new practices.

4.4 Limitations

Our agile process model selection matrix (Appendix 1) provides alternative ways (heuristics) to manage a large embedded software development project. It does not show any single best way of running a project – there is no one-size-fits-all methodology. Note that, typically, there is more than one way to tackle a certain problem. Also, some tradeoffs often exist.

All in all, this is about advanced software process competence (Level 3 competence, according to Turner and Boehm [38]). The degree to which the potential advantages of a certain process model actually can be realized depends much on the knowledge and skills of the project manager and the team.

5. Conclusions

Modern NPD project environments require flexible modes of operation. Hence, an important part of software project management is the selection of a suitable software process model. Agile process models provide many tools for coping with such turbulent project challenges, but it is often not straightforward to select an appropriate process model among the many alternatives. In this paper, we have developed some pragmatic aids for doing this in a systematic way.

We have made a comparative analysis of a range of agile software process models. Our specific viewpoint is to compare the models with respect to their characteristics under typical project problem conditions. The outcome of this comparison is not any particular process model recommendation, but the idea is that a project manager can use the comparison matrix (Appendix 1) to support her own selection of the particular process model. Each agile process model amplifies certain characteristics of the project. The key, then, is to match the current project situation with the process model alternatives.

Often, for a large, complex project, no single process model is the best. Instead, a hybrid model that blends and balances the practices of different models is often the choice [2, 6-7, 8/Ch. 4, 29, 35]. This depends on the varying characteristics of the different parts of the product and the project environment. Many industrial experience reports conclude that it is often less complicated to adopt certain key practices of one or more process models than to replace an existing one completely [10, 11, 15, 26, 27, 39].

The world of software engineering is in a state of continuous flux. As the products become more complex, the project complexity increases, making the projects subject to more difficult problems. Companies try to fight this complexity by hiring experienced managers (personal competence), as well as by building knowledge inside the organization (such as building detailed process models). In this paper, we have summarized some first-hand information to a structured form, giving the software fellows a fresh viewpoint of agile process models.

Our comparison matrix indicates that those agile process models tackle many potential project problems. Large variations exist, however, depending on the nature of the models. Some models (e.g., Scrum) address project management issues in particular, while other models (e.g., XP) focus more on the engineering area. None of the analyzed models are free from at least some limiting constraints.

Our special focus has been on large embedded software projects. None of analyzed agile process models is intended specifically for embedded software development, but most of them are applicable to some extent, and, for example, XP and Scrum have been applied to embedded and mission-critical software development [11]. The special concerns of large embedded software projects are not so much in software construction but in the related systems and hardware engineering issues.

This paper leaves room for further study:

(1) *Empirical validation:* At the time of the writing, we are not yet able to present current empirical validation data about our propositions. Such data could be collected by experimenting with the matrix (Appendix 1) in ongoing software projects.

(2) An improvement could be to add a different sort of keys to problem factors in the matrix (Appendix 1). In different situations, after all, different views might be useful. For example, Ambler has compared some development approaches with respect to their ability to support certain overall project requirements [2/Ch. 1]. Does the project need to prioritize—for example predictability, flexibility, or visibility?

(3) Changing the focus from large-scale embedded systems to some other, e.g., multisite or web site development project.

(4) How do other project management dimensions (people, technology) affect the process model selection?

Acknowledgments

The authors would like to thank Jani Kykyri (Yomi) for his input.

References

1. Abrahamsson, P., *et al.,* 2003, "New Directions on Agile Methods: A Comparative Analysis," *Proceedings of the 25th International Conference on Software Engineering (ICSE)*, 244-254.
2. Ambler, S., 1998, *Process Patterns – Building Large-Scale Systems Using Object Technology,* Cambridge, UK, Cambridge University Press.
3. Ambler, S. W., 2003, "The Right Tool for the Job," Software Development, December, 50-52.
4. Beck K. and M. Fowler, 2001, *Planning Extreme Programming,* Addison-Wesley / Pearson, Upper Saddle River, NJ, USA.
5. Boehm, B., 1991, "Software Risk Management: Principles and Practices," IEEE Software, **8**(1), 32-41.
6. Boehm, B., 2002, "Get Ready for Agile Methods, with Care," IEEE Computer, **35**(1), 64-69.
7. Boehm, B. and R. Turner, 2004, *Balancing Agility and Discipline – A Guide for the Perplexed,* USA: Addison-Wesley / Pearson Education, Boston, MA.
8. Brown, W. J., *et al.,* 2000, *AntiPatterns in Project Management,* John Wiley & Sons, New York, NY.
9. Cockburn, A., 2002, *Agile Software Development*, Addison-Wesley / Pearson, Boston, MA.
10. Dagnino, A., 2002, "An Evolutionary Lifecycle Model with Agile Practices for Software Development at

ABB," *Proceedings of the 8th International Conference on Engineering of Complex Computer Systems (ICECCS),* 215-223.

11. Drobka, J., Noftz, D. and R. Raghu, 2004, "Piloting XP on Four Mission-Critical Projects," IEEE Software, **21**(6), 70-75.
12. Fairley, R. E. and M. J. Willshire, 2003, "Why the Vasa Sank: 10 Problems and Some Antidotes for Software Projects," IEEE Software, **20**(2), 18-25.
13. Glass, R. L., 2004, "Matching Methodology to Problem Domain," CACM, **47**(5), 19-21.
14. Gnatz, M., *et al.,* 2003, "The Living Software Development Process," SQP, **5**(3), 4-16.
15. Greene, B., 2004, "Agile Methods Applied to Embedded Firmware Development," *Proceedings of the Agile Development Conference (ADG),* 71-77.
16. Highsmith, J. A., 2000, *Adaptive Software Development – A Collaborative Approach to Managing Complex Systems,* Dorset House Publishing, New York, NY.
17. Highsmith, J. A., 2002, *Agile Software Development Ecosystems,* Addison-Wesley / Pearson Education, Boston, MA.
18. Hirsch, M., 2002, "Making RUP Agile," *Proceedings of the Object-Oriented Programming Systems Languages and Applications (OOPSLA), Practitioners Reports.*
19. Hull, M. E. C., *et al.,* 2002, "Software Development Processes – An Assessment," *Information and Software Technology,* **44**(1), 1-12.
20. Iansiti, M., 1995, "Shooting the Rapids: Managing Product Development in Turbulent Environments," California Management Review, **38**(1), 37-58.
21. Jaufman, O. and S. Przewoznik, 2004, "Suitability of State-of-the-Art Methods for Interdisciplinary System Development in Automotive Industry," *Proceedings of the ACM Workshop on Interdisciplinary Software Engineering Research (WISER),* 78-82.
22. Jones, C., 1996, *Patterns of Software System Failure and Success,* Boston, MA, International Thompson Computer Press, USA.
23. Kettunen, P. and M. Laanti, 2004, "How to Steer an Embedded Software Project: Tactics for Selecting the Software Process Model," Information and Software Technology, **47**(9), 587-608.
24. Kruchten, P., 2000, *The Rational Unified Process: An Introduction,* Reading, MA, USA: Addison-Wesley, USA.
25. Larman C., 2004, *Agile and Iterative Development – A Manager's Guide,* Addison-Wesley / Pearson, Boston, MA.
26. Lindvall, M., *et al.,* 2004, "Agile Software Development in Large Organizations," IEEE Computer, **37**(12), 26-34.
27. Manhart, P. and K. Schneider, 2004, "Breaking the Ice for Agile Development of Embedded Software: An Industry Experience Report," *Proceedings of the 26th International Conference on Software Engineering (ICSE),* 378-386.
28. McConnell, S., 1996, *Rapid Development: Taming Wild Software Schedules,* Microsoft Press, Redmond, WA, USA.
29. Ould, M. A., 1999, *Managing Software Quality and Business Risk,* John Wiley & Sons, Chichester, UK.
30. Palmer, S. R. and J. M. Felsing, 2002, *A Practical Guide to Feature-Driven Development,* Prentice-Hall, Upper Saddle River, NJ.
31. Reifer, D., 2002, "Ten Deadly Risks in Internet and Intranet Software Development," IEEE Software*,* **19**(2), 12-14.
32. Ronkainen, J. and P. Abrahamsson, 2003, "Software Development Under Stringent Hardware Constraints: Do Agile Methods Have a Chance?" *Proceedings of the 4th International Conference on Extreme Programming and Agile Processes in Software Engineering,* 73-79.
33. Schwaber K. and M. Beedle, 2002, *Agile Software Development with Scrum,* Prentice-Hall, Upper Saddle River, NJ.
34. Smith, J., 2002, "The 40 Root Causes of Troubled IT Projects," *IEE* Engineering Management Journal, **12**(5), 238-242.
35. Sommerville, I., 1996, "Software Process Models," ACM Computing Surveys, **28**(1), 269-271.
36. Song, X. and L. J. Osterweil, 1992, "Toward Objective, Systematic Design Method Comparisons," *IEEE Software,* **9**(3), 43-53.
37. Taramaa, J., *et al*., 1998, "Product-based Software Process Improvement for Embedded Systems," *Proceedings of the 24th Euromicro Conference* (2), 905-912.
38. Turner, R. and B. Boehm, 2003, "People Factors in Software Management: Lessons From Comparing Agile and Plan-Driven Methods," CrossTalk, **16**(12), 4-8.
39. Vanhanen, J., Jartti, J. and T. Kähkönen, 2003, "Practical Experiences of Agility in the Telecom Industry," *Proceedings of the 4th International Confer-*

ence on Extreme Programming and Agile Processes in Software Engineering, 279-287.

40. http://www.agilemanifesto.org/, January, 2005.

Biographies

Petri Kettunen is an R&D software engineering specialist with Nokia Corporation, Finland. He received his M.Sc. in Computer Science at Helsinki University and his Lic.Sc. (Tech.) at Helsinki University of Technology. He has been involved with industrial embedded software development for more than 15 years in various positions. His current research interests include new product development project management methods, as well as embedded software engineering process models.

Maarit Laanti is a Senior Project Manager with Nokia Corporation, Finland. Her interest is in new product development, project management and leadership. She has been leading variously-sized software development projects for more than ten years in Nokia, including a two-year assignment in Dallas, Texas, USA. She holds an M.Sc. in Data Transfer and Computer Sciences from Helsinki University of Technology.

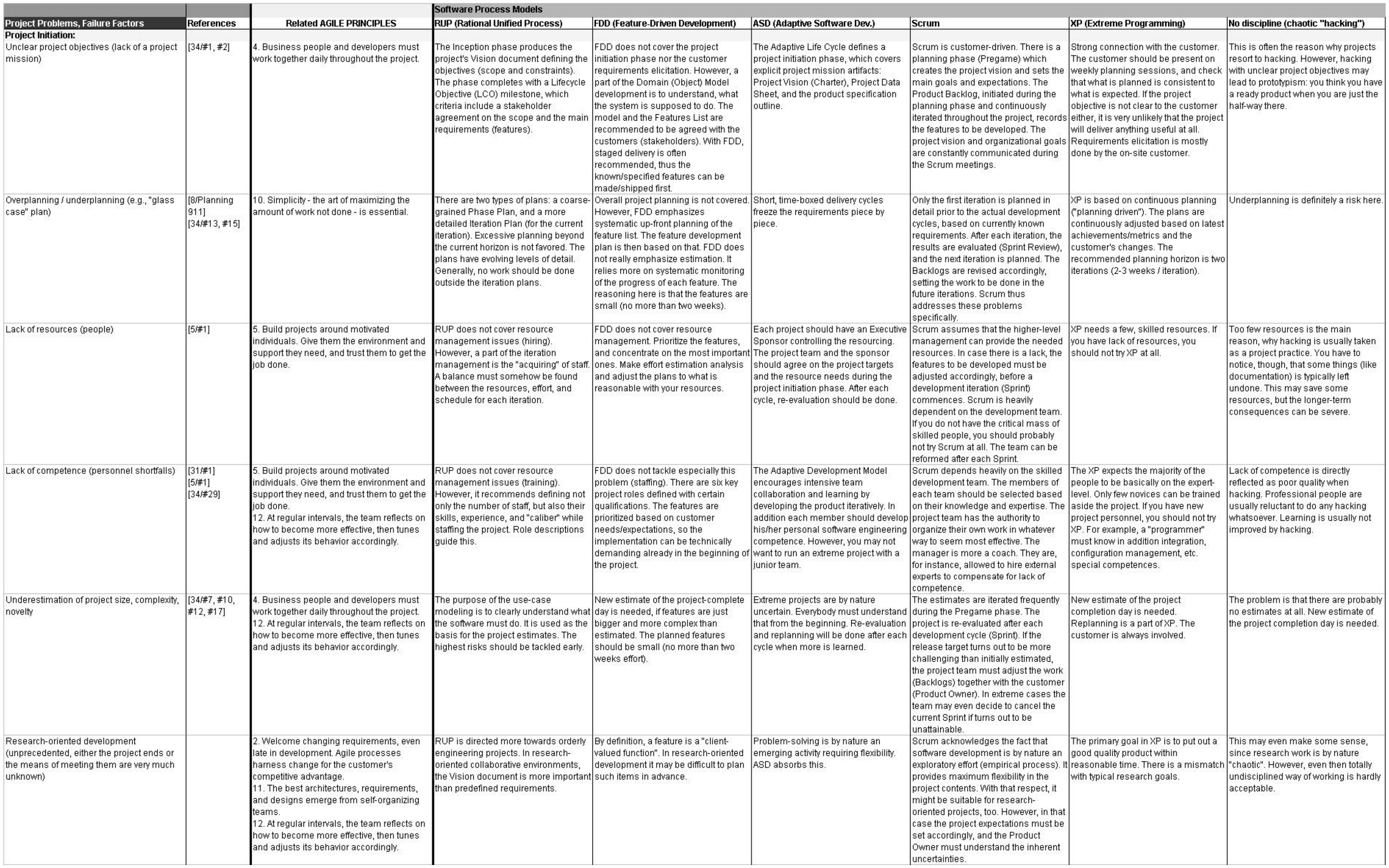
Appendices

1. Agile Software Process Comparison Matrix

NOTE: The column titled *References* shows the problem item numbers used in the respective publications, e.g., [34/#1] refers to the first item of the list in [34].

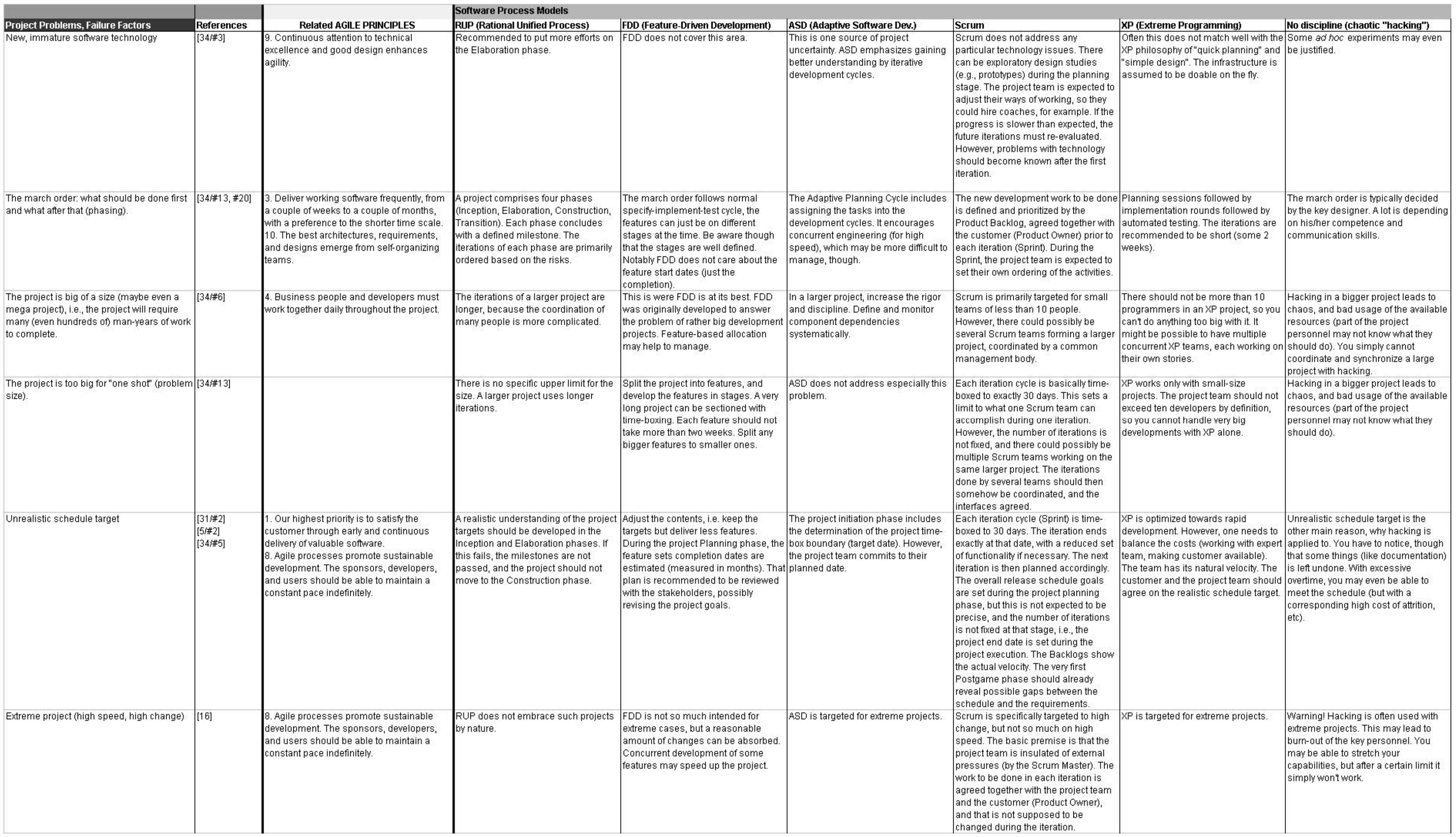
2. Agile Software Process Characteristics Matrix

Appendix 1: Agile Software Process Comparison Matrix



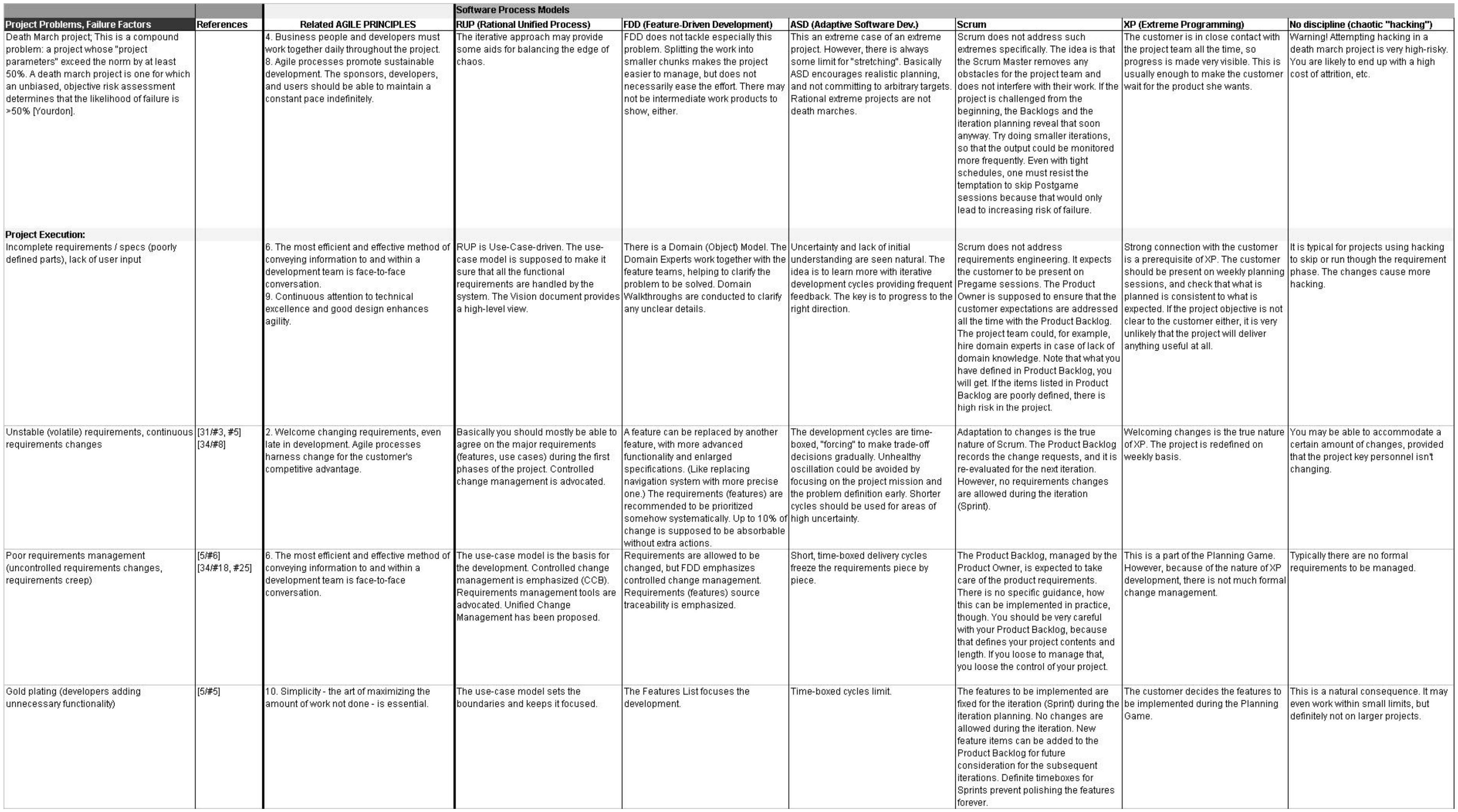
			Software Process Models					
Project Problems, Failure Factors	**References**	**Related AGILE PRINCIPLES**	**RUP (Rational Unified Process)**	**FDD (Feature-Driven Development)**	**ASD (Adaptive Software Dev.)**	**Scrum**	**XP (Extreme Programming)**	**No discipline (chaotic "hacking")**
Project Initiation:								
Unclear project objectives (lack of a project mission)	[34/#1, #2]	4. Business people and developers must work together daily throughout the project.	The Inception phase produces the project's Vision document defining the objectives (scope and constraints). The phase completes with a Lifecycle Objective (LCO) milestone, which criteria include a stakeholder agreement on the scope and the main requirements (features).	FDD does not cover the project initiation phase nor the customer requirements elicitation. However, a part of the Domain (Object) Model development is to understand, what the system is supposed to do. The model and the Features List are recommended to be agreed with the customers (stakeholders). With FDD, staged delivery is often recommended, thus the known/specified features can be made/shipped first.	The Adaptive Life Cycle defines a project initiation phase, which covers explicit project mission artifacts: Project Vision (Charter), Project Data Sheet, and the product specification outline.	Scrum is customer-driven. There is a planning phase (Pregame) which creates the project vision and sets the main goals and expectations. The Product Backlog, initiated during the planning phase and continuously iterated throughout the project, records the features to be developed. The project vision and organizational goals are constantly communicated during the Scrum meetings.	Strong connection with the customer. The customer should be present on weekly planning sessions, and check that what is planned is consistent to what is expected. If the project objective is not clear to the customer either, it is very unlikely that the project will deliver anything useful at all. Requirements elicitation is mostly done by the on-site customer.	This is often the reason why projects resort to hacking. However, hacking with unclear project objectives may lead to prototypism: you think you have a ready product when you are just the half-way there.
Overplanning / underplanning (e.g., "glass case" plan)	[8/Planning 911] [34/#13, #15]	10. Simplicity - the art of maximizing the amount of work not done - is essential.	There are two types of plans: a coarse-grained Phase Plan, and a more detailed Iteration Plan (for the current iteration). Excessive planning beyond the current horizon is not favored. The plans have evolving levels of detail. Generally, no work should be done outside the iteration plans.	Overall project planning is not covered. However, FDD emphasizes systematic up-front planning of the feature list. The feature development plan is then based on that. FDD does not really emphasize estimation. It relies more on systematic monitoring of the progress of each feature. The reasoning here is that the features are small (no more than two weeks).	Short, time-boxed delivery cycles freeze the requirements piece by piece.	Only the first iteration is planned in detail prior to the actual development cycles, based on currently known requirements. After each iteration, the results are evaluated (Sprint Review), and the next iteration is planned. The Backlogs are revised accordingly, setting the work to be done in the future iterations. Scrum thus addresses these problems specifically.	XP is based on continuous planning ("planning driven"). The plans are continuously adjusted based on latest achievements/metrics and the customer's changes. The recommended planning horizon is two iterations (2-3 weeks / iteration).	Underplanning is definitely a risk here.
Lack of resources (people)	[5/#1]	5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.	RUP does not cover resource management issues (hiring). However, a part of the iteration management is the "acquiring" of staff. A balance must somehow be found between the resources, effort, and schedule for each iteration.	FDD does not cover resource management. Prioritize the features, and concentrate on the most important ones. Make effort estimation analysis and adjust the plans to what is reasonable with your resources.	Each project should have an Executive Sponsor controlling the resourcing. The project team and the sponsor should agree on the project targets and the resource needs during the project initiation phase. After each cycle, re-evaluation should be done.	Scrum assumes that the higher-level management can provide the needed resources. In case there is a lack, the features to be developed must be adjusted accordingly, before a development iteration (Sprint) commences. Scrum is heavily dependent on the development team. If you do not have the critical mass of skilled people, you should probably not try Scrum at all. The team can be reformed after each Sprint.	XP needs a few, skilled resources. If you have lack of resources, you should not try XP at all.	Too few resources is the main reason, why hacking is usually taken as a project practice. You have to notice, though, that some things (like documentation) is typically left undone. This may save some resources, but the longer-term consequences can be severe.
Lack of competence (personnel shortfalls)	[31/#1] [5/#1] [34/#29]	5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done. 12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.	RUP does not cover resource management issues (training). However, it recommends defining not only the number of staff, but also their skills, experience, and "caliber" while staffing the project. Role descriptions guide this.	FDD does not tackle especially this problem (staffing). There are six key project roles defined with certain qualifications. The features are prioritized based on customer needs/expectations, so the implementation can be technically demanding already in the beginning of the project.	The Adaptive Development Model encourages intensive team collaboration and learning by developing the product iteratively. In addition each member should develop his/her personal software engineering competence. However, you may not want to run an extreme project with a junior team.	Scrum depends heavily on the skilled development team. The members of each team should be selected based on their knowledge and expertise. The project team has the authority to organize their own work in whatever way to seem most effective. The manager is more a coach. They are, for instance, allowed to hire external experts to compensate for lack of competence.	The XP expects the majority of the people to be basically on the expert-level. Only few novices can be trained aside the project. If you have new project personnel, you should not try XP. For example, a "programmer" must know in addition integration, configuration management, etc. special competences.	Lack of competence is directly reflected as poor quality when hacking. Professional people are usually reluctant to do any hacking whatsoever. Learning is usually not improved by hacking.
Underestimation of project size, complexity, novelty	[34/#7, #10, #12, #17]	4. Business people and developers must work together daily throughout the project. 12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.	The purpose of the use-case modeling is to clearly understand what the software must do. It is used as the basis for the project estimates. The highest risks should be tackled early.	New estimate of the project-complete day is needed, if features are just bigger and more complex than estimated. The planned features should be small (no more than two weeks effort).	Extreme projects are by nature uncertain. Everybody must understand that from the beginning. Re-evaluation and replanning will be done after each cycle when more is learned.	The estimates are iterated frequently during the Pregame phase. The project is re-evaluated after each development cycle (Sprint). If the release target turns out to be more challenging than initially estimated, the project team must adjust the work (Backlogs) together with the customer (Product Owner). In extreme cases the team may even decide to cancel the current Sprint if turns out to be unattainable.	New estimate of the project completion day is needed. Replanning is a part of XP. The customer is always involved.	The problem is that there are probably no estimates at all. New estimate of the project completion day is needed.
Research-oriented development (unprecedented, either the project ends or the means of meeting them are very much unknown)		2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage. 11. The best architectures, requirements, and designs emerge from self-organizing teams. 12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.	RUP is directed more towards orderly engineering projects. In research-oriented collaborative environments, the Vision document is more important than predefined requirements.	By definition, a feature is a "client-valued function". In research-oriented development it may be difficult to plan such items in advance.	Problem-solving is by nature an emerging activity requiring flexibility. ASD absorbs this.	Scrum acknowledges the fact that software development is by nature an exploratory effort (empirical process). It provides maximum flexibility in the project contents. With that respect, it might be suitable for research-oriented projects, too. However, in that case the project expectations must be set accordingly, and the Product Owner must understand the inherent uncertainties.	The primary goal in XP is to put out a good quality product within reasonable time. There is a mismatch with typical research goals.	This may even make some sense, since research work is by nature "chaotic". However, even then totally undisciplined way of working is hardly acceptable.

Appendix 1: Agile Software Process Comparison Matrix (cont.)



			Software Process Models					
Project Problems, Failure Factors	**References**	**Related AGILE PRINCIPLES**	**RUP (Rational Unified Process)**	**FDD (Feature-Driven Development)**	**ASD (Adaptive Software Dev.)**	**Scrum**	**XP (Extreme Programming)**	**No discipline (chaotic "hacking")**
New, immature software technology	[34/#3]	9. Continuous attention to technical excellence and good design enhances agility.	Recommended to put more efforts on the Elaboration phase.	FDD does not cover this area.	This is one source of project uncertainty. ASD emphasizes gaining better understanding by iterative development cycles.	Scrum does not address any particular technology issues. There can be exploratory design studies (e.g., prototypes) during the planning stage. The project team is expected to adjust their ways of working, so they could hire coaches, for example. If the progress is slower than expected, the future iterations must re-evaluated. However, problems with technology should become known after the first iteration.	Often this does not match well with the XP philosophy of "quick planning" and "simple design". The infrastructure is assumed to be doable on the fly.	Some *ad hoc* experiments may even be justified.
The march order: what should be done first and what after that (phasing).	[34/#13, #20]	3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter time scale. 10. The best architectures, requirements, and designs emerge from self-organizing teams.	A project comprises four phases (Inception, Elaboration, Construction, Transition). Each phase concludes with a defined milestone. The iterations of each phase are primarily ordered based on the risks.	The march order follows normal specify-implement-test cycle, the features can just be on different stages at the time. Be aware though that the stages are well defined. Notably FDD does not care about the feature start dates (just the completion).	The Adaptive Planning Cycle includes assigning the tasks into the development cycles. It encourages concurrent engineering (for high speed), which may be more difficult to manage, though.	The new development work to be done is defined and prioritized by the Product Backlog, agreed together with the customer (Product Owner) prior to each iteration (Sprint). During the Sprint, the project team is expected to set their own ordering of the activities.	Planning sessions followed by implementation rounds followed by automated testing. The iterations are recommended to be short (some 2 weeks).	The march order is typically decided by the key designer. A lot is depending on his/her competence and communication skills.
The project is big of a size (maybe even a mega project), i.e., the project will require many (even hundreds of) man-years of work to complete.	[34/#6]	4. Business people and developers must work together daily throughout the project.	The iterations of a larger project are longer, because the coordination of many people is more complicated.	This is were FDD is at its best. FDD was originally developed to answer the problem of rather big development projects. Feature-based allocation may help to manage.	In a larger project, increase the rigor and discipline. Define and monitor component dependencies systematically.	Scrum is primarily targeted for small teams of less than 10 people. However, there could possibly be several Scrum teams forming a larger project, coordinated by a common management body.	There should not be more than 10 programmers in an XP project, so you can't do anything too big with it. It might be possible to have multiple concurrent XP teams, each working on their own stories.	Hacking in a bigger project leads to chaos, and bad usage of the available resources (part of the project personnel may not know what they should do). You simply cannot coordinate and synchronize a large project with hacking.
The project is too big for "one shot" (problem size).	[34/#13]		There is no specific upper limit for the size. A larger project uses longer iterations.	Split the project into features, and develop the features in stages. A very long project can be sectioned with time-boxing. Each feature should not take more than two weeks. Split any bigger features to smaller ones.	ASD does not address especially this problem.	Each iteration cycle is basically time-boxed to exactly 30 days. This sets a limit to what one Scrum team can accomplish during one iteration. However, the number of iterations is not fixed, and there could possibly be multiple Scrum teams working on the same larger project. The iterations done by several teams should then somehow be coordinated, and the interfaces agreed.	XP works only with small-size projects. The project team should not exceed ten developers by definition, so you cannot handle very big developments with XP alone.	Hacking in a bigger project leads to chaos, and bad usage of the available resources (part of the project personnel may not know what they should do).
Unrealistic schedule target	[31/#2] [5/#2] [34/#5]	1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software. 8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.	A realistic understanding of the project targets should be developed in the Inception and Elaboration phases. If this fails, the milestones are not passed, and the project should not move to the Construction phase.	Adjust the contents, i.e. keep the targets but deliver less features. During the project Planning phase, the feature sets completion dates are estimated (measured in months). That plan is recommended to be reviewed with the stakeholders, possibly revising the project goals.	The project initiation phase includes the determination of the project time-box boundary (target date). However, the project team commits to their planned date.	Each iteration cycle (Sprint) is time-boxed to 30 days. The iteration ends exactly at that date, with a reduced set of functionality if necessary. The next iteration is then planned accordingly. The overall release schedule goals are set during the project planning phase, but this is not expected to be precise, and the number of iterations is not fixed at that stage, i.e., the project end date is set during the project execution. The Backlogs show the actual velocity. The very first Postgame phase should already reveal possible gaps between the schedule and the requirements.	XP is optimized towards rapid development. However, one needs to balance the costs (working with expert team, making customer available). The team has its natural velocity. The customer and the project team should agree on the realistic schedule target.	Unrealistic schedule target is the other main reason, why hacking is applied to. You have to notice, though that some things (like documentation) is left undone. With excessive overtime, you may even be able to meet the schedule (but with a corresponding high cost of attrition, etc).
Extreme project (high speed, high change)	[16]	8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.	RUP does not embrace such projects by nature.	FDD is not so much intended for extreme cases, but a reasonable amount of changes can be absorbed. Concurrent development of some features may speed up the project.	ASD is targeted for extreme projects.	Scrum is specifically targeted to high change, but not so much on high speed. The basic premise is that the project team is insulated of external pressures (by the Scrum Master). The work to be done in each iteration is agreed together with the project team and the customer (Product Owner), and that is not supposed to be changed during the iteration.	XP is targeted for extreme projects.	Warning! Hacking is often used with extreme projects. This may lead to burn-out of the key personnel. You may be able to stretch your capabilities, but after a certain limit it simply won't work.

Appendix 1: Agile Software Process Comparison Matrix (cont.)

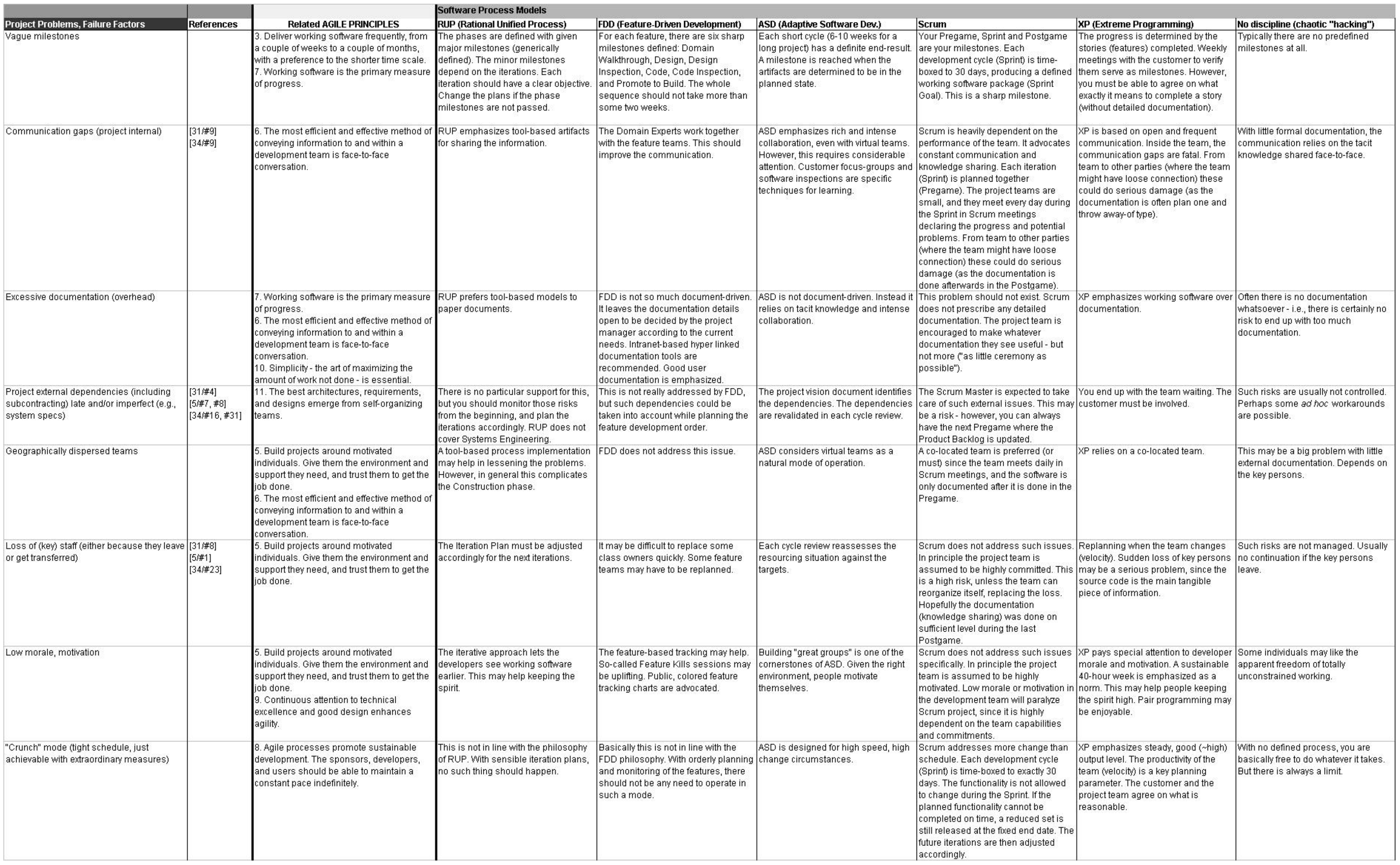


			Software Process Models					
Project Problems, Failure Factors	**References**	**Related AGILE PRINCIPLES**	**RUP (Rational Unified Process)**	**FDD (Feature-Driven Development)**	**ASD (Adaptive Software Dev.)**	**Scrum**	**XP (Extreme Programming)**	**No discipline (chaotic "hacking")**
Death March project; This is a compound problem: a project whose "project parameters" exceed the norm by at least 50%. A death march project is one for which an unbiased, objective risk assessment determines that the likelihood of failure is >50% [Yourdon].		4. Business people and developers must work together daily throughout the project. 8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.	The iterative approach may provide some aids for balancing the edge of chaos.	FDD does not tackle especially this problem. Splitting the work into smaller chunks makes the project easier to manage, but does not necessarily ease the effort. There may not be intermediate work products to show, either.	This an extreme case of an extreme project. However, there is always some limit for "stretching". Basically ASD encourages realistic planning, and not committing to arbitrary targets. Rational extreme projects are not death marches.	Scrum does not address such extremes specifically. The idea is that the Scrum Master removes any obstacles for the project team and does not interfere with their work. If the project is challenged from the beginning, the Backlogs and the iteration planning reveal that soon anyway. Try doing smaller iterations, so that the output could be monitored more frequently. Even with tight schedules, one must resist the temptation to skip Postgame sessions because that would only lead to increasing risk of failure.	The customer is in close contact with the project team all the time, so progress is made very visible. This is usually enough to make the customer wait for the product she wants.	Warning! Attempting hacking in a death march project is very high-risky. You are likely to end up with a high cost of attrition, etc.
Project Execution:								
Incomplete requirements / specs (poorly defined parts), lack of user input		6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation. 9. Continuous attention to technical excellence and good design enhances agility.	RUP is Use-Case-driven. The use-case model is supposed to make it sure that all the functional requirements are handled by the system. The Vision document provides a high-level view.	There is a Domain (Object) Model. The Domain Experts work together with the feature teams, helping to clarify the problem to be solved. Domain Walkthroughs are conducted to clarify any unclear details.	Uncertainty and lack of initial understanding are seen natural. The idea is to learn more with iterative development cycles providing frequent feedback. The key is to progress to the right direction.	Scrum does not address requirements engineering. It expects the customer to be present on Pregame sessions. The Product Owner is supposed to ensure that the customer expectations are addressed all the time with the Product Backlog. The project team could, for example, hire domain experts in case of lack of domain knowledge. Note that what you have defined in Product Backlog, you will get. If the items listed in Product Backlog are poorly defined, there is high risk in the project.	Strong connection with the customer is a prerequisite of XP. The customer should be present on weekly planning sessions, and check that what is planned is consistent to what is expected. If the project objective is not clear to the customer either, it is very unlikely that the project will deliver anything useful at all.	It is typical for projects using hacking to skip or run though the requirement phase. The changes cause more hacking.
Unstable (volatile) requirements, continuous requirements changes	[31/#3, #5] [34/#8]	2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.	Basically you should mostly be able to agree on the major requirements (features, use cases) during the first phases of the project. Controlled change management is advocated.	A feature can be replaced by another feature, with more advanced functionality and enlarged specifications. (Like replacing navigation system with more precise one.) The requirements (features) are recommended to be prioritized somehow systematically. Up to 10% of change is supposed to be absorbable without extra actions.	The development cycles are time-boxed, "forcing" to make trade-off decisions gradually. Unhealthy oscillation could be avoided by focusing on the project mission and the problem definition early. Shorter cycles should be used for areas of high uncertainty.	Adaptation to changes is the true nature of Scrum. The Product Backlog records the change requests, and it is re-evaluated for the next iteration. However, no requirements changes are allowed during the iteration (Sprint).	Welcoming changes is the true nature of XP. The project is redefined on weekly basis.	You may be able to accommodate a certain amount of changes, provided that the project key personnel isn't changing.
Poor requirements management (uncontrolled requirements changes, requirements creep)	[5/#6] [34/#18, #25]	6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.	The use-case model is the basis for the development. Controlled change management is emphasized (CCB). Requirements management tools are advocated. Unified Change Management has been proposed.	Requirements are allowed to be changed, but FDD emphasizes controlled change management. Requirements (features) source traceability is emphasized.	Short, time-boxed delivery cycles freeze the requirements piece by piece.	The Product Backlog, managed by the Product Owner, is expected to take care of the product requirements. There is no specific guidance, how this can be implemented in practice, though. You should be very careful with your Product Backlog, because that defines your project contents and length. If you loose to manage that, you loose the control of your project.	This is a part of the Planning Game. However, because of the nature of XP development, there is not much formal change management.	Typically there are no formal requirements to be managed.
Gold plating (developers adding unnecessary functionality)	[5/#5]	10. Simplicity - the art of maximizing the amount of work not done - is essential.	The use-case model sets the boundaries and keeps it focused.	The Features List focuses the development.	Time-boxed cycles limit.	The features to be implemented are fixed for the iteration (Sprint) during the iteration planning. No changes are allowed during the iteration. New feature items can be added to the Product Backlog for future consideration for the subsequent iterations. Definite timeboxes for Sprints prevent polishing the features forever.	The customer decides the features to be implemented during the Planning Game.	This is a natural consequence. It may even work within small limits, but definitely not on larger projects.

Appendix 1: Agile Software Process Comparison Matrix (cont.)

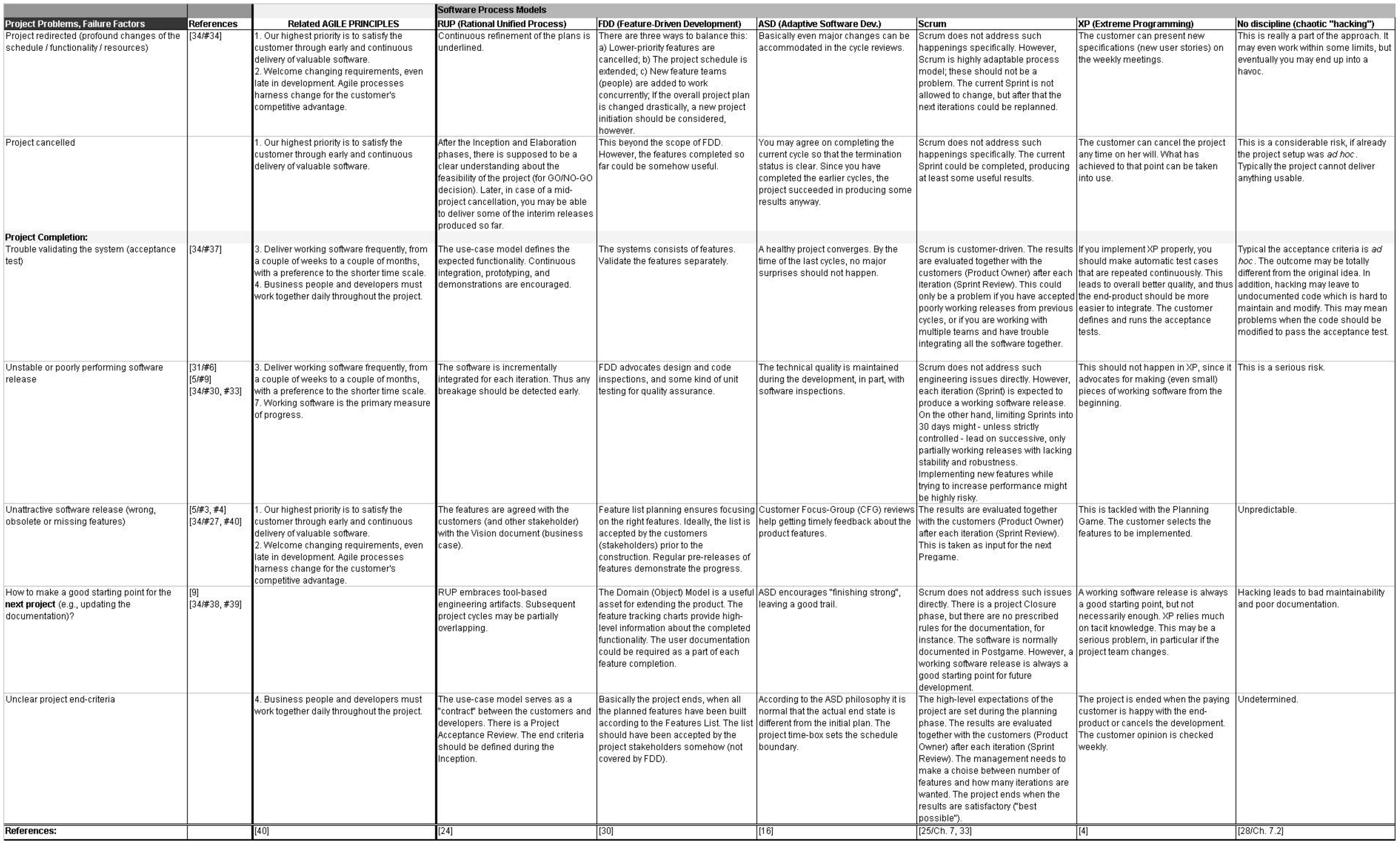
			Software Process Models					
Project Problems, Failure Factors	**References**	**Related AGILE PRINCIPLES**	**RUP (Rational Unified Process)**	**FDD (Feature-Driven Development)**	**ASD (Adaptive Software Dev.)**	**Scrum**	**XP (Extreme Programming)**	**No discipline (chaotic "hacking")**
Constantly changing schedule target		3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter time scale.	The iteration plan defines the start and end dates, and the delivery date. You should not change the current iteration much. However, the next one could be replanned.	Features are recommended to be very small (no more than two weeks of effort). Feature sets could occasionally be reassigned between the teams (but not too often).	The project is time-boxed. The cycle dates are not changed. If the original schedule turns out to be wrong, it can be renegotiated in cycle replanning.	Scrum is rather balancing with the features than the time. Each iteration cycle (Sprint) is time-boxed to exactly 30 days. The number of iterations is not fixed at the project outset, though. The project completion date is thus set during the project. Adding more iteration cycles makes also a risk, though.	The iterations are time-boxed. The releases are small. New releases can be scheduled, if the customer wishes that.	The schedule depends very much on the key persons. They may or may not be able to make it, but it is hard to tell that in advance (poor predictability).
Poor software architecture design quality	[34/#30]	9. Continuous attention to technical excellence and good design enhances agility.	RUP is an architecture-centric process emphasizing, evolutionary, component-based architecture work with visual modeling (UML).	FDD does not tackle especially this problem. It may be very hard to make any corrections to the architecture in the middle of the project, when half of your features are already ready.	ASD does not cover architecture design details.	The system architecture is expected to be defined during the planning phase. The software architecture emerges and evolves during the iterations. However, Scrum does not address how the architecture is actually designed. The project team is expected to solve the problems on their own. Small changes to architecture can be implemented easily.	Small changes to architecture can be implemented easily. However, XP does not offer much support for system architecture design (just "metaphors" and "simple design").	This is definitely a risk, since typically there is no systematic architecture design at all. "Quick-and-dirty" solutions are typical.
Wrong architecture solution selected in the first phase (inadequate systems engineering)	[5/#10] [34/#19]	11. The best architectures, requirements, and designs emerge from self-organizing teams. 9. Continuous attention to technical excellence and good design enhances agility.	The architecture choices are based on architecturally significant use cases. An (evolutionary) architectural prototype is recommended. Thus, no totally wrong solutions should result (architecture first).	FDD does not tackle especially this problem. It may be very hard to make any corrections to the architecture in the middle of the project, when half of your features are already ready.	The problem definition done during the project initiation guides the architecture selections. Iterative development cycles support learning more about the architectural choices.	The system architecture is expected to be defined during the planning phase. The project team is expected to solve the technical problems on their own. If you make wrong initial choises, you might loose the work done during the first iterations.	XP does not tackle especially this problem. It might be hard to convince the customer to buy the development cost for the better architecture (when the customer is actually expecting progress in form on some new features). Refactoring could help to some extent, but fundamentally wrong solutions cannot be salved.	This is an obvious risk for any longer-term development.
Inappropriate design methods	[31/#7] [34/#21]	9. Continuous attention to technical excellence and good design enhances agility. 12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.	RUP advocates certain design methods which are supposed to be generally applicable (such as Use Cases, UML, components).	Rework features to some later release with better tools.	ASD does not cover design details.	Scrum does not address such engineering issues. Changing the tools should be possible during the Pregame.	Replan and re-schedule your project. If the customer accepts this, can be done. In general, do not try to use totally new design methods with XP.	We can try to change them on the fly.
Unsuitable or low-quality tools		12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly. 9. Continuous attention to technical excellence and good design enhances agility.	RUP is very much tool-oriented. There is a wide set of commercially available tools.	FDD does cover any tool issues.	ASD does not cover any tools details.	Scrum does not address such engineering issues. Changing the tools should be possible during the Pregame.	XP does not cover any tool details. But there would not be any sense of buying the best experts on the field and equip them with poor tools. In general, do not try to use totally new tools with XP.	We can try to change them on the fly.
Integration difficulties	[34/#28, #32]	3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter time scale. 9. Continuous attention to technical excellence and good design enhances agility. 11. The best architectures, requirements, and designs emerge from self-organizing teams.	RUP encourages almost continuous test and integration (executable releases for each iteration). Any breakage should thus become visible early. Early architectural risk reduction is emphasized.	FDD does not define integration in any exact way. However, the Chief Programmers are responsible for testing their features. FDD used with staged delivery makes the integration steps smaller, and thus easier. A regular build schedule is recommended (supported by solid configuration management).	There is no particular emphasis on integration, but each cycle should end with valid executable results.	Each iteration (Sprint) produces a working (integrated and tested) software package. Daily builds are recommended. Potential integration difficulties are thus revealed early. The whole software team can participate to the integration.	You have the whole software team to back-up the integration. However, this requires that everybody knows how to do the integration. Note also that it may be difficult to manage the integration of a large complex system without rigorous up-front planning.	Hacking is likely to lead to undocumented code and unspecified interfaces, which make the integration step extremely difficult.
Low visibility to progress	[34/#22]	7. Working software is the primary measure of progress. 10. Simplicity - the art of maximizing the amount of work not done - is essential.	Progress is measured in terms of use cases (features) completed, test cases passed, performance requirements satisfied, and risks eliminated. Regular, demonstration-based assessment is emphasized. Iteration Assessments are conducted after each iteration (e.g., revalidating the requirements).	FDD provides good visibility to progress, because delivery of each feature can be monitored. The progress reporting is recommend to be done based on feature completeness. If the project takes longer than some 3 months, formal monthly progress reviews are recommended.	ASD does not improve the traditional project visibility since it relies on intense collaboration (tacit knowledge). The documents evolve during the whole development. Only the results matter.	This should not be a problem at all with Scrum. The project team reports the progress in daily Scrum meetings, estimating the remaining effort. The Sprint Backlog is updated accordingly. This constant monitoring (Sprint Backlog Graph) ensures high visibility. The Product Backlog is always visible, showing the current priorities of the work to be done. The working features after each Sprint demonstrate the true progress for the Release Backlog.	This should not be a problem at all with XP. The customer sees the progress weekly.	You may be able to show some progress by demonstrating the software. However, typically the quality tends to be unpredictable. The progress is often variable due to unplanned design.

Appendix 1: Agile Software Process Comparison Matrix (cont.)



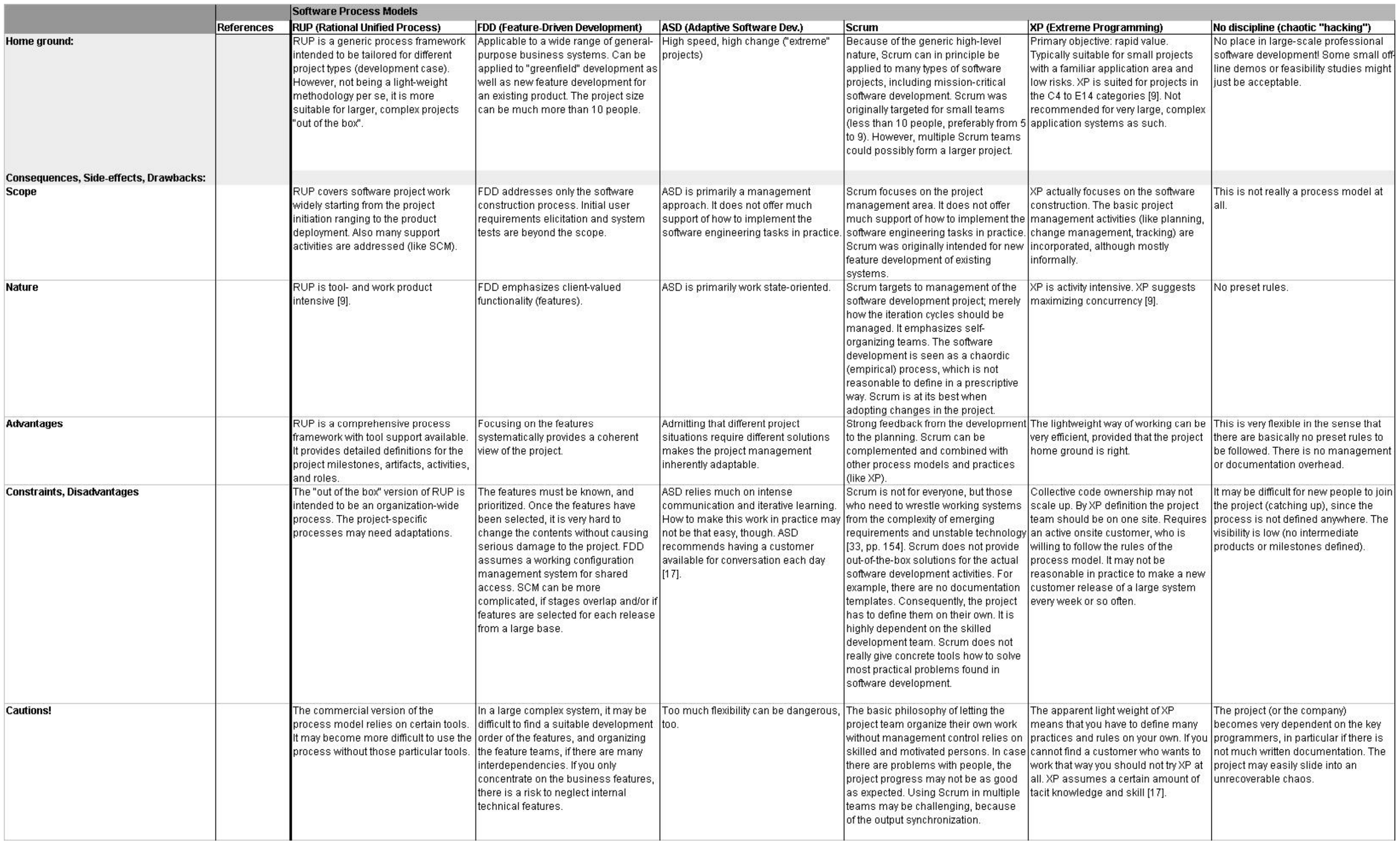
			Software Process Models					
Project Problems, Failure Factors	**References**	**Related AGILE PRINCIPLES**	**RUP (Rational Unified Process)**	**FDD (Feature-Driven Development)**	**ASD (Adaptive Software Dev.)**	**Scrum**	**XP (Extreme Programming)**	**No discipline (chaotic "hacking")**
Vague milestones		3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter time scale. 7. Working software is the primary measure of progress.	The phases are defined with given major milestones (generically defined). The minor milestones depend on the iterations. Each iteration should have a clear objective. Change the plans if the phase milestones are not passed.	For each feature, there are six sharp milestones defined: Domain Walkthrough, Design, Design Inspection, Code, Code Inspection, and Promote to Build. The whole sequence should not take more than some two weeks.	Each short cycle (6-10 weeks for a long project) has a definite end-result. A milestone is reached when the artifacts are determined to be in the planned state.	Your Pregame, Sprint and Postgame are your milestones. Each development cycle (Sprint) is time-boxed to 30 days, producing a defined working software package (Sprint Goal). This is a sharp milestone.	The progress is determined by the stories (features) completed. Weekly meetings with the customer to verify them serve as milestones. However, you must be able to agree on what exactly it means to complete a story (without detailed documentation).	Typically there are no predefined milestones at all.
Communication gaps (project internal)	[31/#9] [34/#9]	6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.	RUP emphasizes tool-based artifacts for sharing the information.	The Domain Experts work together with the feature teams. This should improve the communication.	ASD emphasizes rich and intense collaboration, even with virtual teams. However, this requires considerable attention. Customer focus-groups and software inspections are specific techniques for learning.	Scrum is heavily dependent on the performance of the team. It advocates constant communication and knowledge sharing. Each iteration (Sprint) is planned together (Pregame). The project teams are small, and they meet every day during the Sprint in Scrum meetings declaring the progress and potential problems. From team to other parties (where the team might have loose connection) these could do serious damage (as the documentation is done afterwards in the Postgame).	XP is based on open and frequent communication. Inside the team, the communication gaps are fatal. From team to other parties (where the team might have loose connection) these could do serious damage (as the documentation is often plan one and throw away-of type).	With little formal documentation, the communication relies on the tacit knowledge shared face-to-face.
Excessive documentation (overhead)		7. Working software is the primary measure of progress. 6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation. 10. Simplicity - the art of maximizing the amount of work not done - is essential.	RUP prefers tool-based models to paper documents.	FDD is not so much document-driven. It leaves the documentation details open to be decided by the project manager according to the current needs. Intranet-based hyper linked documentation tools are recommended. Good user documentation is emphasized.	ASD is not document-driven. Instead it relies on tacit knowledge and intense collaboration.	This problem should not exist. Scrum does not prescribe any detailed documentation. The project team is encouraged to make whatever documentation they see useful - but not more ("as little ceremony as possible").	XP emphasizes working software over documentation.	Often there is no documentation whatsoever - i.e., there is certainly no risk to end up with too much documentation.
Project external dependencies (including subcontracting) late and/or imperfect (e.g., system specs)	[31/#4] [5/#7, #8] [34/#16, #31]	11. The best architectures, requirements, and designs emerge from self-organizing teams.	There is no particular support for this, but you should monitor those risks from the beginning, and plan the iterations accordingly. RUP does not cover Systems Engineering.	This is not really addressed by FDD, but such dependencies could be taken into account while planning the feature development order.	The project vision document identifies the dependencies. The dependencies are revalidated in each cycle review.	The Scrum Master is expected to take care of such external issues. This may be a risk - however, you can always have the next Pregame where the Product Backlog is updated.	You end up with the team waiting. The customer must be involved.	Such risks are usually not controlled. Perhaps some *ad hoc* workarounds are possible.
Geographically dispersed teams		5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done. 6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.	A tool-based process implementation may help in lessening the problems. However, in general this complicates the Construction phase.	FDD does not address this issue.	ASD considers virtual teams as a natural mode of operation.	A co-located team is preferred (or must) since the team meets daily in Scrum meetings, and the software is only documented after it is done in the Pregame.	XP relies on a co-located team.	This may be a big problem with little external documentation. Depends on the key persons.
Loss of (key) staff (either because they leave or get transferred)	[31/#8] [5/#1] [34/#23]	5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.	The Iteration Plan must be adjusted accordingly for the next iterations.	It may be difficult to replace some class owners quickly. Some feature teams may have to be replanned.	Each cycle review reassesses the resourcing situation against the targets.	Scrum does not address such issues. In principle the project team is assumed to be highly committed. This is a high risk, unless the team can reorganize itself, replacing the loss. Hopefully the documentation (knowledge sharing) was done on sufficient level during the last Postgame.	Replanning when the team changes (velocity). Sudden loss of key persons may be a serious problem, since the source code is the main tangible piece of information.	Such risks are not managed. Usually no continuation if the key persons leave.
Low morale, motivation		5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done. 9. Continuous attention to technical excellence and good design enhances agility.	The iterative approach lets the developers see working software earlier. This may help keeping the spirit.	The feature-based tracking may help. So-called Feature Kills sessions may be uplifting. Public, colored feature tracking charts are advocated.	Building "great groups" is one of the cornerstones of ASD. Given the right environment, people motivate themselves.	Scrum does not address such issues specifically. In principle the project team is assumed to be highly motivated. Low morale or motivation in the development team will paralyze Scrum project, since it is highly dependent on the team capabilities and commitments.	XP pays special attention to developer morale and motivation. A sustainable 40-hour week is emphasized as a norm. This may help people keeping the spirit high. Pair programming may be enjoyable.	Some individuals may like the apparent freedom of totally unconstrained working.
"Crunch" mode (tight schedule, just achievable with extraordinary measures)		8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.	This is not in line with the philosophy of RUP. With sensible iteration plans, no such thing should happen.	Basically this is not in line with the FDD philosophy. With orderly planning and monitoring of the features, there should not be any need to operate in such a mode.	ASD is designed for high speed, high change circumstances.	Scrum addresses more change than schedule. Each development cycle (Sprint) is time-boxed to exactly 30 days. The functionality is not allowed to change during the Sprint. If the planned functionality cannot be completed on time, a reduced set is still released at the fixed end date. The future iterations are then adjusted accordingly.	XP emphasizes steady, good (~high) output level. The productivity of the team (velocity) is a key planning parameter. The customer and the project team agree on what is reasonable.	With no defined process, you are basically free to do whatever it takes. But there is always a limit.

Appendix 1: Agile Software Process Comparison Matrix (cont.)



			Software Process Models					
Project Problems, Failure Factors	**References**	**Related AGILE PRINCIPLES**	**RUP (Rational Unified Process)**	**FDD (Feature-Driven Development)**	**ASD (Adaptive Software Dev.)**	**Scrum**	**XP (Extreme Programming)**	**No discipline (chaotic "hacking")**
Project redirected (profound changes of the schedule / functionality / resources)	[34/#34]	1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software. 2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.	Continuous refinement of the plans is underlined.	There are three ways to balance this: a) Lower-priority features are cancelled; b) The project schedule is extended; c) New feature teams (people) are added to work concurrently; If the overall project plan is changed drastically, a new project initiation should be considered, however.	Basically even major changes can be accommodated in the cycle reviews.	Scrum does not address such happenings specifically. However, Scrum is highly adaptable process model; these should not be a problem. The current Sprint is not allowed to change, but after that the next iterations could be replanned.	The customer can present new specifications (new user stories) on the weekly meetings.	This is really a part of the approach. It may even work within some limits, but eventually you may end up into a havoc.
Project cancelled		1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.	After the Inception and Elaboration phases, there is supposed to be a clear understanding about the feasibility of the project (for GO/NO-GO decision). Later, in case of a mid-project cancellation, you may be able to deliver some of the interim releases produced so far.	This beyond the scope of FDD. However, the features completed so far could be somehow useful.	You may agree on completing the current cycle so that the termination status is clear. Since you have completed the earlier cycles, the project succeeded in producing some results anyway.	Scrum does not address such happenings specifically. The current Sprint could be completed, producing at least some useful results.	The customer can cancel the project any time on her will. What has achieved to that point can be taken into use.	This is a considerable risk, if already the project setup was *ad hoc*. Typically the project cannot deliver anything usable.
Project Completion:								
Trouble validating the system (acceptance test)	[34/#37]	3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter time scale. 4. Business people and developers must work together daily throughout the project.	The use-case model defines the expected functionality. Continuous integration, prototyping, and demonstrations are encouraged.	The systems consists of features. Validate the features separately.	A healthy project converges. By the time of the last cycles, no major surprises should not happen.	Scrum is customer-driven. The results are evaluated together with the customers (Product Owner) after each iteration (Sprint Review). This could only be a problem if you have accepted poorly working releases from previous cycles, or if you are working with multiple teams and have trouble integrating all the software together.	If you implement XP properly, you should make automatic test cases that are repeated continuously. This leads to overall better quality, and thus the end-product should be more easier to integrate. The customer defines and runs the acceptance tests.	Typical the acceptance criteria is *ad hoc*. The outcome may be totally different from the original idea. In addition, hacking may leave to undocumented code which is hard to maintain and modify. This may mean problems when the code should be modified to pass the acceptance test.
Unstable or poorly performing software release	[31/#6] [5/#9] [34/#30, #33]	3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter time scale. 7. Working software is the primary measure of progress.	The software is incrementally integrated for each iteration. Thus any breakage should be detected early.	FDD advocates design and code inspections, and some kind of unit testing for quality assurance.	The technical quality is maintained during the development, in part, with software inspections.	Scrum does not address such engineering issues directly. However, each iteration (Sprint) is expected to produce a working software release. On the other hand, limiting Sprints into 30 days might - unless strictly controlled - lead on successive, only partially working releases with lacking stability and robustness. Implementing new features while trying to increase performance might be highly risky.	This should not happen in XP, since it advocates for making (even small) pieces of working software from the beginning.	This is a serious risk.
Unattractive software release (wrong, obsolete or missing features)	[5/#3, #4] [34/#27, #40]	1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software. 2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.	The features are agreed with the customers (and other stakeholder) with the Vision document (business case).	Feature list planning ensures focusing on the right features. Ideally, the list is accepted by the customers (stakeholders) prior to the construction. Regular pre-releases of features demonstrate the progress.	Customer Focus-Group (CFG) reviews help getting timely feedback about the product features.	The results are evaluated together with the customers (Product Owner) after each iteration (Sprint Review). This is taken as input for the next Pregame.	This is tackled with the Planning Game. The customer selects the features to be implemented.	Unpredictable.
How to make a good starting point for the **next project** (e.g., updating the documentation)?	[9] [34/#38, #39]		RUP embraces tool-based engineering artifacts. Subsequent project cycles may be partially overlapping.	The Domain (Object) Model is a useful asset for extending the product. The feature tracking charts provide high-level information about the completed functionality. The user documentation could be required as a part of each feature completion.	ASD encourages "finishing strong", leaving a good trail.	Scrum does not address such issues directly. There is a project Closure phase, but there are no prescribed rules for the documentation, for instance. The software is normally documented in Postgame. However, a working software release is always a good starting point for future development.	A working software release is always a good starting point, but not necessarily enough. XP relies much on tacit knowledge. This may be a serious problem, in particular if the project team changes.	Hacking leads to bad maintainability and poor documentation.
Unclear project end-criteria		4. Business people and developers must work together daily throughout the project.	The use-case model serves as a "contract" between the customers and developers. There is a Project Acceptance Review. The end criteria should be defined during the Inception.	Basically the project ends, when all the planned features have been built according to the Features List. The list should have been accepted by the project stakeholders somehow (not covered by FDD).	According to the ASD philosophy it is normal that the actual end state is different from the initial plan. The project time-box sets the schedule boundary.	The high-level expectations of the project are set during the planning phase. The results are evaluated together with the customers (Product Owner) after each iteration (Sprint Review). The management needs to make a choise between number of features and how many iterations are wanted. The project ends when the results are satisfactory ("best possible").	The project is ended when the paying customer is happy with the end-product or cancels the development. The customer opinion is checked weekly.	Undetermined.
References:		[40]	[24]	[30]	[16]	[25/Ch. 7, 33]	[4]	[28/Ch. 7.2]

Appendix 2: Agile Software Process Characteristics Matrix



		Software Process Models					
	References	RUP (Rational Unified Process)	FDD (Feature-Driven Development)	ASD (Adaptive Software Dev.)	Scrum	XP (Extreme Programming)	No discipline (chaotic "hacking")
Home ground:		RUP is a generic process framework intended to be tailored for different project types (development case). However, not being a light-weight methodology per se, it is more suitable for larger, complex projects "out of the box".	Applicable to a wide range of general-purpose business systems. Can be applied to "greenfield" development as well as new feature development for an existing product. The project size can be much more than 10 people.	High speed, high change ("extreme" projects)	Because of the generic high-level nature, Scrum can in principle be applied to many types of software projects, including mission-critical software development. Scrum was originally targeted for small teams (less than 10 people, preferably from 5 to 9). However, multiple Scrum teams could possibly form a larger project.	Primary objective: rapid value. Typically suitable for small projects with a familiar application area and low risks. XP is suited for projects in the C4 to E14 categories [9]. Not recommended for very large, complex application systems as such.	No place in large-scale professional software development! Some small off-line demos or feasibility studies might just be acceptable.
Consequences, Side-effects, Drawbacks:							
Scope		RUP covers software project work widely starting from the project initiation ranging to the product deployment. Also many support activities are addressed (like SCM).	FDD addresses only the software construction process. Initial user requirements elicitation and system tests are beyond the scope.	ASD is primarily a management approach. It does not offer much support of how to implement the software engineering tasks in practice.	Scrum focuses on the project management area. It does not offer much support of how to implement the software engineering tasks in practice. Scrum was originally intended for new feature development of existing systems.	XP actually focuses on the software construction. The basic project management activities (like planning, change management, tracking) are incorporated, although mostly informally.	This is not really a process model at all.
Nature		RUP is tool- and work product intensive [9].	FDD emphasizes client-valued functionality (features).	ASD is primarily work state-oriented.	Scrum targets to management of the software development project; merely how the iteration cycles should be managed. It emphasizes self-organizing teams. The software development is seen as a chaordic (empirical) process, which is not reasonable to define in a prescriptive way. Scrum is at its best when adopting changes in the project.	XP is activity intensive. XP suggests maximizing concurrency [9].	No preset rules.
Advantages		RUP is a comprehensive process framework with tool support available. It provides detailed definitions for the project milestones, artifacts, activities, and roles.	Focusing on the features systematically provides a coherent view of the project.	Admitting that different project situations require different solutions makes the project management inherently adaptable.	Strong feedback from the development to the planning. Scrum can be complemented and combined with other process models and practices (like XP).	The lightweight way of working can be very efficient, provided that the project home ground is right.	This is very flexible in the sense that there are basically no preset rules to be followed. There is no management or documentation overhead.
Constraints, Disadvantages		The "out of the box" version of RUP is intended to be an organization-wide process. The project-specific processes may need adaptations.	The features must be known, and prioritized. Once the features have been selected, it is very hard to change the contents without causing serious damage to the project. FDD assumes a working configuration management system for shared access. SCM can be more complicated, if stages overlap and/or if features are selected for each release from a large base.	ASD relies much on intense communication and iterative learning. How to make this work in practice may not be that easy, though. ASD recommends having a customer available for conversation each day [17].	Scrum is not for everyone, but those who need to wrestle working systems from the complexity of emerging requirements and unstable technology [33, pp. 154]. Scrum does not provide out-of-the-box solutions for the actual software development activities. For example, there are no documentation templates. Consequently, the project has to define them on their own. It is highly dependent on the skilled development team. Scrum does not really give concrete tools how to solve most practical problems found in software development.	Collective code ownership may not scale up. By XP definition the project team should be on one site. Requires an active onsite customer, who is willing to follow the rules of the process model. It may not be reasonable in practice to make a new customer release of a large system every week or so often.	It may be difficult for new people to join the project (catching up), since the process is not defined anywhere. The visibility is low (no intermediate products or milestones defined).
Cautions!		The commercial version of the process model relies on certain tools. It may become more difficult to use the process without those particular tools.	In a large complex system, it may be difficult to find a suitable development order of the features, and organizing the feature teams, if there are many interdependencies. If you only concentrate on the business features, there is a risk to neglect internal technical features.	Too much flexibility can be dangerous, too.	The basic philosophy of letting the project team organize their own work without management control relies on skilled and motivated persons. In case there are problems with people, the project progress may not be as good as expected. Using Scrum in multiple teams may be challenging, because of the output synchronization.	The apparent light weight of XP means that you have to define many practices and rules on your own. If you cannot find a customer who wants to work that way you should not try XP at all. XP assumes a certain amount of tacit knowledge and skill [17].	The project (or the company) becomes very dependent on the key programmers, in particular if there is not much written documentation. The project may easily slide into an unrecoverable chaos.

Appendix 2: Agile Software Process Characteristics Matrix (cont.)

		Software Process Models					
	References	**RUP (Rational Unified Process)**	**FDD (Feature-Driven Development)**	**ASD (Adaptive Software Dev.)**	**Scrum**	**XP (Extreme Programming)**	**No discipline (chaotic "hacking")**
Consequences, Side-effects, Drawbacks:							
Notes		RUP is more like a heavyweight methodology. Some lighter adaptations have been proposed for smaller projects.	Staged delivery causes partially same problems as incremental development (overhead in testing and content management). FDD assumes that the overall value of the features is determined early in the project and that scheduling those features should be primarily a technical decision [17].	There is a philosophy of complex adaptive systems behind.	Since only the high-level management frame is preset, Scrum leaves much room for flexibility - but also puts a lot of responsibility - for adjusting the project work according to the circumstances. Sprint is basically a procedure for adopting to the changing environmental variables. Assumes that the higher-level management can provide enough resources and resolve any external obstacles promptly (even during the day). Product Backlog defines everything that is needed on the final product.	A user story = a feature.	You should not really consider this model as an alternative. Hacking is a process antipattern, sometimes mistakenly justified by iterative development [2].
EMBEDDED SYSTEMS	[32]	There are some real-time software design specialities.	May be suitable. Does not address embedded systems specifically. Planning the feature list with concurrent hardware development may be challenging.	May be suitable. Does not address embedded systems specifically.	There is no specific support for embedded software development. However, there are no particular impediments, either. May be suitable, especially if the hardware is already available. Scrum has been used in safety-critical software projects.	May be suitable, especially if the hardware is already available. Does not address embedded systems specifically.	Some software experiments with the target hardware may make sense.
References:		[24]	[30]	[16]	[25/Ch. 7, 33]	[4]	[28/Ch. 7.2]