Towards a design theoretic characterisation of software development process models

Conference or Workshop Item

Towards a Design-Theoretic Characterisation of Software Development Process Models

Jon G. Hall*, Lucia Rapanotti*
*Department of Computing and Communications,
The Open University, UK
Email: Jon.Hall@open.ac.uk, Lucia.Rapanotti@open.ac.uk

***Abstract*—*Context*: Effective assessment, comparison, selection and adaptation of software development processes remain an acute problem in Software Engineering practice. The quest for a unified theory which might serve this purpose is ongoing.**

***Objective*: To take a first step towards such a theory, with focus on characterising and comparing features of software development process models.**

***Method*: We consider a theory of design as problem solving and investigate how it can be applied to characterise and explicate specific process features in well known process models from the literature. The intention is to characterise emerging trade-offs between resource expenditure and risk mitigation, which result from the interplay between process efforts into problem and solution exploration vs stakeholder validation. The analysis, at this point performed, is purely qualitative, and the treatment of resource expenditure and risk quite abstract.**

***Results*: We provide an initial characterisation and comparison of features found in a wide range of process models from the literature, within a design theoretic framework using a single building block – the Problem Oriented Engineering (POE) Process Pattern – that allows the characterisation of information flow, the relationship between actors, resource usage and developmental risk.**

***Conclusions*: The initial characterisation identifies repeated structure in diverse processes, which allows basic process comparison across models. The interpretations are modular, allowing the possibility of relationships between different process models to be explored. As such, the theory allows for a unified means to characterise and compare systematically key features of different process models. In being of an exploratory nature, the work has a number of limitations, which should be addressed by further research**

I. Introduction

In Design research, design is considered as the fundamental process through which real-world problems are addressed via the invention, assembly and adaptation of technologies, and which involves much planning and decision making, as well as the concrete realisation of artefacts. From this perspective, software engineering/development processes can be regarded as problem solving processes, which begs the question of whether this perspective may lead to useful theories to help practitioners characterise, assess, select and adapt software development processes to their specific organisational, cultural and product development needs. As a starting point, here we focus on one such theory which has emerged in the context of our design theoretic framework for engineering as problem solving, Problem Oriented Engineering (shortly POE, [12]), which over the past decade has been applied in a wide range of real-world engineering contexts, from safety to mission critical systems development.

The intention of this paper is to take a first step concerning the characterisation and comparison of software processes. This is achieved by considering a wide range of software development process models from the literature and use the theory to characterise, explicate and compare key features of such models. The process features we consider are those identified by the theory.

The work reported is of an exploratory nature,

therefore many limitations will be noted, which will be the subject of further research.

POE has developed through our exploration of the engineering of real-world systems. Its application to characterise and compare software development process models in this paper is entirely novel.

This paper has the following structure. In Section II we provide some background on software process selection and tailoring, while Section III briefly recalls relevant POE theoretical concepts. Sections IV and V discuss the application. We end with related work in Section VI and conclusions in Section VII.

II. Background

Software Engineering has a long-standing debate on which software development processes should be adopted to develop quality software systems on time and to budget. An extensive catalogue of models exists, from the early product-manufacturing-inspired Waterfall model [28], its variants, e.g. the V-Model [22], various flavours of iterative and incremental processes, e.g., the RUP [19], and the most recent family of Agile processes [1]. Each model has its supporters and detractors; each has been shown effective in particular contexts and wanting in others. Simply, with the diversification of the software market and the vast ranges of complexity and volatility of the software engineering/development task there can be no one-size-fits-all process model. Practically, this means that organisations and practitioners adopt and adapt models as their needs change (see, for instance, [21], [8]).

That this is most often done in an *ad-hoc* fashion raises the following questions: how can a planned process be assessed for fit? Which evidence is needed to inform decisions about adaptation? Even given ten years' effort (see, e.g., [20], [23] for surveys) many knowledge gaps remain [26], [27] as well as there being a distinct lack of understanding of success criteria for process tailoring and of pragmatic process application and/or adaptation in specific contexts.

Our conjecture is that such questions can be answered only if direct comparisons between processes are possible. In this paper, we therefore provide an initial mapping of various process models into our general theory so as to provide a basis for their comparison.

III. A design-theoretic problem solving view of software development

POE takes a design-theoretic problem solving view of software engineering. Paraphrasing G.F.C Rogers' definition of engineering [25] we consider software engineering to be:

> the practice of organising the design and construction of any *software artefact*[1] which transforms the physical world around us to meet some recognised need

Let Env be Rogers' physical world, Need the recognised need (AKA requirement) of G – the problem holder – and Soln the software artefact[2].

Define a software (engineering) problem to be the proposition:

$$\mathsf{P}: \quad \mathsf{Env}(\mathsf{Soln}) \ \textbf{meets}_{\mathsf{G}} \ \mathsf{Need}$$

the truth value of which indicates that, when Soln is installed in the environment Env, their combination meets G's Need.

Establishing the truth value of the proposition is done within the propositional calculus, so that design can be presented as the familiar propositional proof tree[3], albeit augmented with steps that correspond to software-engineering-like process. [13], for instance, defines 'solution exploration' phase architectural expansion steps that allow a software architecture to structure code; other 'problem exploration' steps constitute 'sense making' for environment and need. Phases can be associated with problem and solution validation checkpoints by appropriate stakeholders.

If the validation associated with a phase fails, then the resource for that phase is lost. As such, development risk is addressable under POE ([17]). The relationship between resource consuming problem and solution exploration phases and risk mitigation through validation is shown in Figure 1,

[1] Rogers uses *artifice*, less suggestive of a (physical) object.

[2] POE does not constrain the description language for these problem elements.

[3] Which we call Natural Design; c.f. Gentzen's Natural Deduction ([9]). Like Natural Deduction, Natural Design is amenable to automation, [10].

and is known as the POE Process Pattern (shortly PPP, [12]). Note: the PPP is a pattern and not a model; it is suggestive of the ordering of phases and validation. Although the PPP has a formal interpretation within POE, we do not describe that here.

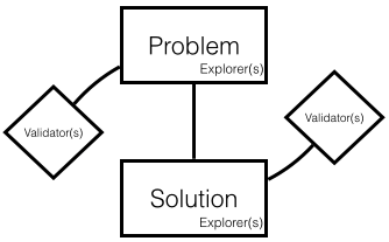

Fig. 1. The POE Process Pattern (PPP): the observed structure of problem solving in design; both Problem and Solution Exploration can be seen as problem solving activities in themselves; Validation, if done, is a separate activity involving one or more Validators.

Consider the following software development problem:

Example: Michael, an experienced developer, wants a piece of software for his smart phone which alerts him to take an umbrella when the forecast predicts rain. Following Rogers, Michael's physical world includes a smart phone, the weather, weather station, and an umbrella; the weather station monitors the weather through sensors and issues weather forecasts that are available via the Internet on his phone. Michael (as problem explorer) describes his need as:

$$\mathsf{Need} = \text{`when the weather forecast predicts rain, I am alerted to take an umbrella'}.$$

As a proposition Michael's problem is:

$$\mathsf{P}_{\mathsf{Michael}} : \langle \mathsf{Smart\_phone}, \mathsf{Weather}, \mathsf{Weather\_station}, \mathsf{Umbrella} \rangle (\mathsf{Soln}) \ \mathbf{meets}_{\mathsf{Michael}} \ \mathsf{Need}$$

where Soln is to be found. ■

As described above, problem exploration led to a (in the general case, partial) description of Env and Need. From this understanding of the problem, Michael can begin to construct his Soln:

Example: Given the above analysis of his problem, Michael begins the solution exploration (or coding) phase. Michael's first decision is that his platform of choice for the Soln is `If This Then That` – IFTTT – an internet software-as-a-service tool for creating simple internet enabled trigger-action programs ([29]; so-called *recipes*, [14]). His recipe is shown in Figure 2, with some output. ■

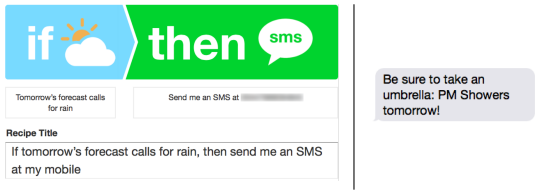

Fig. 2. (Left) the IFTTT recipe to remind Michael to take an umbrella if tomorrow's forecast is for rain; and (right) an SMS received from the service.

We can fit Michael's process to the PPP as follows: Michael has but a single instance of problem and solution exploration with no (external) validation: Michael is his own 'customer'. Thus, a single application of the PPP without validation leads to the simple 'bang-bang' process representation of Figure 4(left). We note that the graphical notation is suggestive rather than formal.

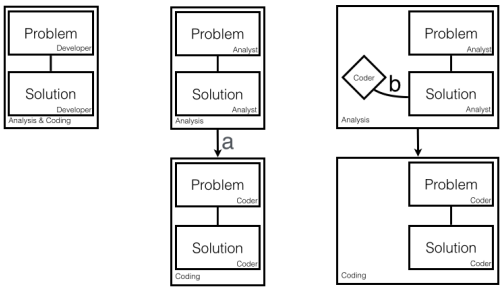

Fig. 3. Modelling Analysis and Coding as (left) a simple 'bang-bang' problem solving activity; (centre) so that a 'bang-bang' Analysis delivers its output to 'bang-bang' Coding; and (right) in which the coder as the recipient of the Analysis output is seen as a validator of, and perhaps even a collaborator in, that analysis

IV. Royce's Software Development Processes

In his treatise on large-scale software development processes Royce ([28]) describes a number

of process models, ranging from the simple to the complex, four of which are show in Figure 4. In this section, we will use Royce's models as part of the exemplar basis for software development processes characterisation in POE; later we model more modern processes.

A. Royce's 'essential' development steps

Describing Figure 4.1, Royce says:

> There are two essential steps common to all computer program developments, regardless of size or complexity. There is first an analysis step, followed second by a coding step [...]. This sort of very simple implementation concept is in fact all that is required if the effort is sufficiently small and if the final product is to be operated by those who built it.

This is Royce's simplest model, shown in Figure 4(1), appropriate in cases such as the simple example above. Our first and simplest model of Royce's software processes as problem solving processes, then, consists of a problem exploration phase – Analysis – and a solution exploration phase – Coding – together as a single 'bang-bang' application of the process pattern, as was shown in Figure 3(left); use is internal, so there is no need for validation in this simple case.

In simple, single person development, the results of analysis need not formally be recorded. In multi-developer situations, however, there are very good reasons to separate analysis from coding, documenting both. To represent this, our process model changes: instead of a single PPP instance, we use two, one each for the analysis and coding phases; see Figure 3(centre). To do so, however, we must identify the problems that are solved during analysis and coding; the analysis problem is

> to find a description of environment and need that is suitable as the basis for communication with the coding phase

whereas that for the coding phase is

> to understand the analysis description, and to produce from it a software artefact that satisfies it.

In contrast to the original 'bang-bang' model, there are *two* solution artefacts of differing natures: the first is documentation of the environment and need, the second, code. We note that, as the analysis phase no longer has a pure software solution, our starting point of software problem is too restrictive, other problems types are needed. We do not discuss this further here, other than to say that the solution to a problem can be based on any technology, not just software. What this means for POE are discussed in [12].

Returning to the multi-developer case, given that the coding phase will begin from the analyst's output, to lower development risk we could set the coder as validator for that output (see Figure 3(right)): as validator, the coder expresses their willingness to accept (or not) the Analysis output (point 'b' in the figure) before it is passed to them as coder. This would encourage the analyst to consult the coder during the analysis phase so that their validation environment and needs can be better understood.

It is this most sophisticated pattern application that we use as the basis of our modelling of Royce's more complex processes.

B. Idealised 'waterfall' model

Although Royce doesn't use the term 'waterfall' in [28], by that term is commonly understood the idealised process that Royce documented for larger installations, reproduced in Figure 4(2). Royce argues that while the previous model includes essential steps as representing:

> the kind of development effort for which most customers are happy to pay, since both steps involve genuinely creative work which directly contributes to the usefulness of the final product.

further steps are required and justified as the basis for larger software system development, despite not always being welcomed by developer or customer — they only contribute to the product indirectly, but still need planning and staffing. In this expanded model, each phase runs to completion before the next starts, with no thought for iteration. Given the discussion of the last section, the reader will readily see how Royce's cascade of steps is simply many sequential instances of the PPP.

C. Introducing iteration

As a stepping stone in our development, Figure 4 is useful. However, as Royce point out, this process model is unrealistic.

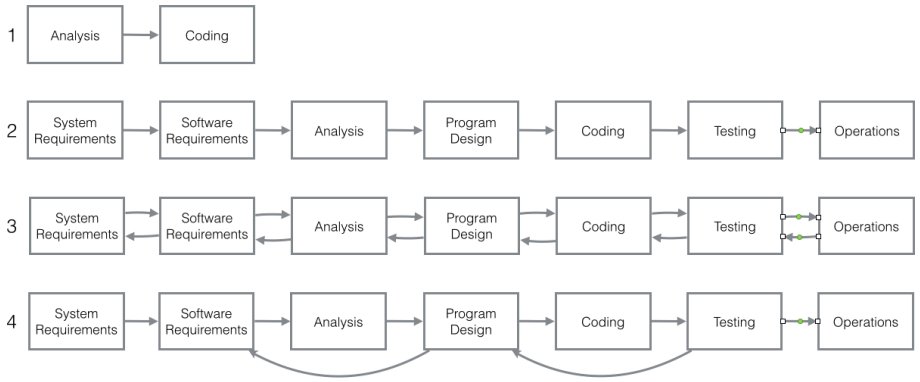

Fig. 4. Royce's first four software development process models, adapted from [28]: 1) 'Implementation steps to deliver a small computer program for internal operations'; 2) 'Implementation steps to develop a large computer program for delivery to a customer'; 3) 'Hopefully, the iterative interaction between the various phases is confined to successive steps'; 4) 'Unfortunately, for the process illustrated, the design iterations are never confined to the successive steps'

Figure 4(3) corresponds to a process in which iteration is possible between successive phases while, in Figure 4(4), iteration is present only between Testing and Coding and Coding and Software Requirements. Royce argues from experience that the latter is, in fact, the more realistic model and it is the one he goes on to develop further. For completeness, however, we provide in Figure 5 interpretations of both: in (a) iteration is achieved by setting the current phase development stakeholder as validator of the preceding phase; in (b), by setting the Program Designer as validator of the Software Requirements phase and the Tester as a validator of the Program Design phase.

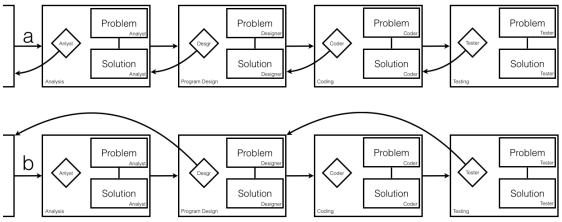

Fig. 5. Modelling the additional linkage between the Testing and Program Design phases of Royce's increasingly sophisticated models in Figure 4.3 & 4.4. In (b), the Tester uses the tests to decide not only on the Coding step but on the Program Design step too.

D. Process model with risk elimination

Royce goes on to argue that additional features must be added to Figure 4(4) to eliminate most development risk, leading to his most complex model, shown in Figure 6. These additional features concern effort required to produce non-software artefacts and to conduct customer reviews, discussed next.

1) Review points and Customer involvement: Review points (decorated circles in Figure 6) are points in the process in which the customer is involved formally, in depth and as a way of establishing commitment at points before delivery. Royce identifies three such review points in his process model:

- Preliminary Software Review (PSR), in which a preliminary program design is reviewed before the (essential) Analysis step;
- Critical Software Review (CSR), in which the full program design is reviewed before the (essential) Coding step;
- Final Software Acceptance Review (FSAR), in which the outcome of the testing phase is validated before Operations.

At this point we note that to gain the customer's formal commitment through the PSR means that some thought must be put into the way that the developmental team will interact with the customer

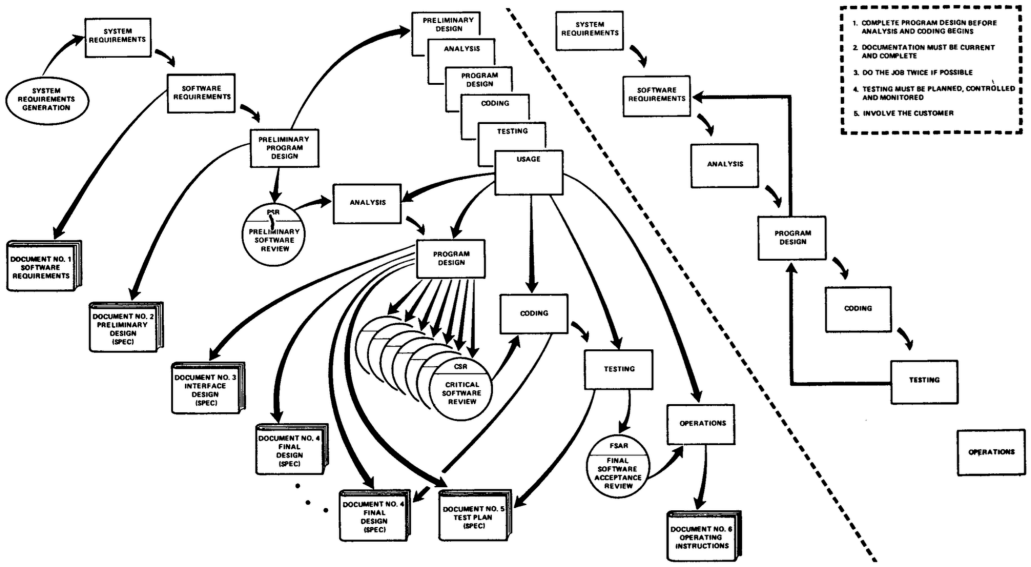

Fig. 6. Royce's summary of his risk-managed software development process (reproduced from [28, Figure 10, page 338])

through the process preparing the preliminary program design. This is likely to involve working with the customer during the Preliminary Program Design activity to ensure that the sign-off activity will, indeed, result in customer commitment. With reference to our process pattern, this identifies the customer as a validator for the output of Preliminary Program Design (corresponding to 'a' annotation in Figure 7). In reality, and although it might nor be desirable from a developmental perspective, it is likely that the customer will have views on whether the problem to which the preliminary program design is the solution has been formulated properly (annotation 'b' in the figure). As the software requirements were a component of this problem, the customer may, in actuality, decide that the preliminary program design does not gain their commitment because the software requirements were in error (annotation 'c' in the figure)[4] causing the process to iterate back to the software requirements phase.

[4]Of course, the possibility exists that the software requirements were in error because the system requirements were in error; however, this simple extension steps outside of Royce's model.

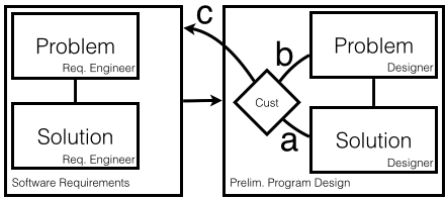

Fig. 7. The customer's role in Preliminary Program Design as validator may require further (a) solution or (b) problem exploration of the preliminary program design, or even (c) reconsideration of the Software Requirements.

When offering the customer a formal opportunity for validation the developers must accept that the customer can withhold it initially, until – perhaps substantial – rework discharges their validation needs. Such is the nature of development risk. We have seen that the first opportunity for customer validation in Royce's model brings into question any and all work carried out before the validation point. Our discussion so far then suggests that, as risk is limited to the accumulation of expended resources between validation points, customer involvement should be frequent and, perhaps, even more frequent than that suggested by Royce. Indeed, this is one

of the tenets of Agile development (as we also discuss in Section V-A). The downside of customer involvement is the difficulty and cost of frequent engagement and, thus, there is a trade-off to be made between frequency and cost.

For the other areas in which formal customer validation is suggested by Royce (CSR and FSAR), a similar analysis applies, albeit with perhaps the commitment of more resources and more depth. The ramifications of customer involvement – that it may cause the reconsideration of previous steps – are apparent. The ramifications for the model are clear.

2) Subsidiary effort and Documentation: In the process model of Figure 6, an undecorated ellipse is used to represent subsidiary effort, that is effort expended during a development step, but producing non-software artefacts, such as the system overview which is used for communication of ([28])

> an elemental understanding of the system [to be built]

throughout the developmental process, from

> at least one person [that has] a deep understanding which comes partially from having had to write the overview document.

Among all non-software artefacts, documentation is singled out in Royce's model, indicated by book-like icons. According to Royce [28, page 332]:

> Management of software is simply impossible without a very high degree of documentation

and gives a number of reasons he believes this to be the case:

- documentation is the vehicle by which a designer communicates with other designers, management and the customer;
- documentation *is* the spec and the design in the early stages of development;
- the monetary value of documentation is felt during testing, operations and redesign.

For processes in which it has such a prominent role, it follows that documentation should be fit-for-purpose. It is reasonable, therefore, to consider Royce's documentation as an artefact requiring validation by those downstream stakeholders that would use it. As such, an explicit problem solving process is needed for documentation, for which validating stakeholders need identifying: for instance, its downstream stakeholders may wish to serve as validators (in a similar fashion as discussed for the Customer as validator in the previous section). Of course, it may be that operations and redesign (or even testing) personnel are not be available as validators, in which case some other scheme for producing fit-for-purpose documentation might be chosen, such as over-engineering [18].

V. Post-Royce software development processes

Much has changed since Royce defined his initial models. In this section, we show how agile processes can be represented within our theory.

A. Agile processes: SCRUM

SCRUM is one of the best known agile processes. It also one which comes with a fairly clear software development process model (see Figure 8).

One interesting characteristic of this model is that it includes explicit consideration of planning activities alongside more traditional software development activities, as well as singling out specific stakeholder roles. This is in contrast to Royce's models (and those derived from them), where planning and project managing is usually done via a separate parallel process. Our interpretation of sprint planning, execution and review, is illustrated in Figure 9, and discussed in the sequel.

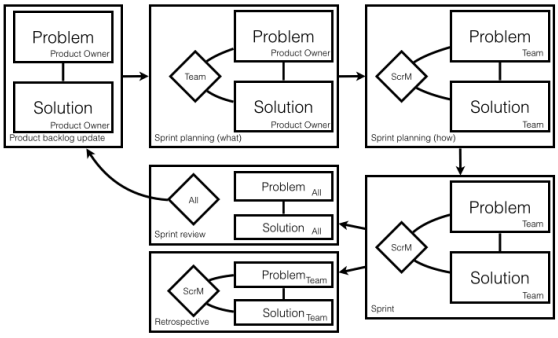

Fig. 9. Planning, executing and reviewing sprint as a composite problem solving process

In SCRUM, development is carried out in time-boxed iterative and incremental cycles, called *sprints*. Once a sprint's target is set (i.e., which

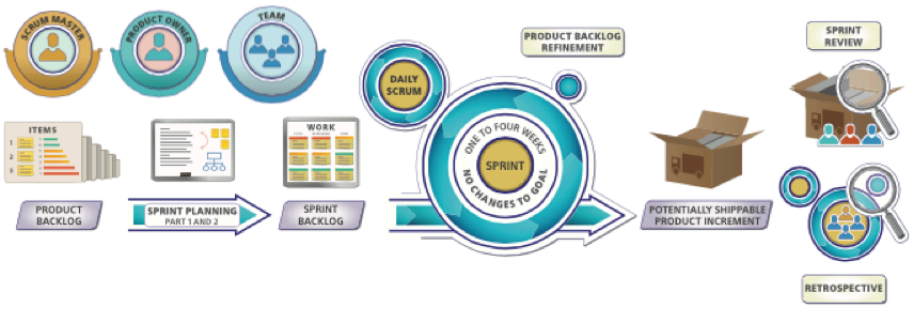

Fig. 8. The SCRUM process model. Adapted from [7].

feature or requirement to address), it remains fixed throughout the sprint, and the output is a 'potentially shippable product increment.' Responsibility for development is with a cross-functional co-located (Development) Team, coached and enabled by a ScrumMaster — two stakeholder roles identified in the SCRUM model.

A possible interpretation of a sprint takes us back to the first model we considered in Section IV-A: the Team is jointly responsible for analysing the (fixed) requirements and come up with a software solution; descriptions are easily exchanged among the co-located team members[5], so that there is no need for explicit documentation to be created. We could argue that the enabling role of the ScrumMaster is to mitigate risk within the sprint, as the ScrumMaster checks that activities within the sprint are carried out in the true spirit of SCRUM and provides advice and support to ensure that is the case.

Sprint planning is based on a 'Definition of Done', which is agreed between the Team and the Product Owner, another SCRUM-defined role, responsible for identifying and prioritising product features for development in sprints. Two Sprint Planning Meetings are used for the Product Owner to discuss with the Team *what* needs to be implemented, and for the Team to decide *how* to implement it.

[5]A 15-minute Daily Scrum is a stand-up meeting in which the Team exchanges information at the start of the working day. There is also a shared Sprint Backlog accessed and updated by the Team.

As a sprint is time-boxed (usually between two and four weeks) and work stops when time expires, it is easy to estimate accurately each sprint's resource expenditure, and development risk is confined to the loss of that expenditure. The duration of the Sprint Planning Meetings is also fixed, so, again resource expenditure is easy to predict.

The evaluation of the output of a sprint takes place after its completion via a Sprint Review, a meeting where Team, ScrumMaster, Product Owner and other invited stakeholders (e.g., customer, users, executives) participate. It is intended as an 'inspect and adapt' meeting for the product: inspecting what has been developed as well as how external factors or expectations may have changed, and adapt as necessary to plan the next product increment. Alongside Sprint Review, there is also a Sprint Retrospective, in which inspection and adaptation applies to the development process and environment: this is where the Team review their own working practices to decide what needs preserving or changing. This is often facilitated by a ScrumMaster for another Team. As everything else in SCRUM, these reviews are also time-boxed.

The responsibility for overall product planning lies with the Product Owner, who needs to maximise return on investment by identifying and prioritising product features for development in sprints. The Product Backlog, a list of prioritised features, is used for this purpose: features from the Product Backlog are input to Sprint Planning Meetings, and the list is updated following Sprint Reviews.

Note: Scrum does not prescribe how the Product

Owner should go about selecting and prioritising features: like Royce's model, how the initial requirements are set and validated with customers, users, and other stakeholders, lies outside of the scope of Scrum. There is, however, an expectation that the Product Owner be sufficiently senior and experienced to perform this function: risk is mitigated by appointing an appropriately experienced person.

B. Scalable Agile processes: DAD SCRUM

While Agile approaches are welcome as people-centric process models, they are also criticised, at least in their vanilla form, for being difficult to scale up to large scale development projects. As observed by Ambler [2]:

> Although agile teams have pretty much figured out how to effectively address functional requirements, most are still struggling with NFRs and constraints. [...] Agile requirements management strategies [...] assume that requirements are self-contained and can be addressed in a finite period of time, an assumption that doesn't always hold true for NFRs and constraints.

With this problem in mind, Ambler defines his Disciplined Agile Delivery (DAD) framework [3], as a way to combine architectural strategies with Agile process models. The DAD SCRUM model introduces the role of Architecture Owner to take ownership of the incremental development of the system architecture in a way which mirrors the way the Product Owner takes ownership of product features for development. At the start of the project, alongside the Product Owner working towards the initial Product Backlog, the Architecture Owner works towards an initial architectural vision, which is then shared with the Team and becomes the subject of scrutiny both in sprint planning and review meetings.

To model Ambler's architectural extensions, we look again at the assumptions underpinning Royce's process development. Royce's process is claimed to manage risk [28, page 335] under the assumption that:

> At any point in the design process after the requirements analysis is completed there exists a firm and closeup, moving baseline to which to return in the event of unforeseen design difficulties. What we have is an effective fallback position that tends to maximize the extent of early work that is salvageable and preserved. [28, page 328]

i.e., risk management is predicated on the completeness of requirements analysis.

In volatile contexts, the completeness of the requirements analysis cannot be guaranteed, whence Royce's claim does not necessarily hold. Indeed, when requirements and/or environment are in a constant state of flux, its clear that a rigorous application of Royce's process would never end – each change of requirements would necessitate, essentially, the whole process to be restarted.

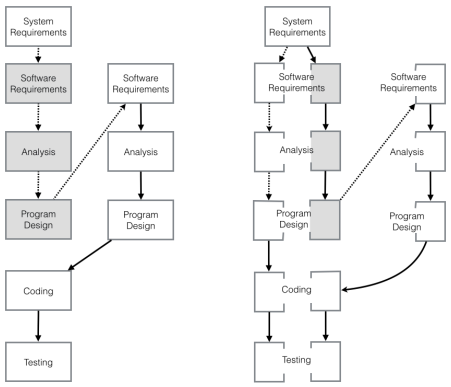

Fig. 10. In which (left) monolithic software requirements are (right) bundled into two allowing independent development paths for each bundle. Grey shading indicates backtracked development.

Arguably, we might identify the deficiencies of Royce's process in volatile contexts as its dependence on monolithic software requirements: Figure 10(left) illustrates that any change of requirements implies a complete redevelopment under Royce's model. As suggested in Figure 10(right) however, if software requirements could be packaged into two (or more) independent pieces then the cost of failure, and so developmental risk, could be reduced; even in the non-terminating situation, something – the left-hand side of Figure 10(right) – could be delivered.

The splitting of requirements in this way is not trivial. In [13], we define the problem transformation SEPARABLE PROBLEM that decomposes a problem's requirements into n subproblems if it can

be shown that those sub-problem do not interfere[6]. This is a severe constraint, however, and does not generally hold.

In [11] (expanded in [12]), however, we develop a characteristic relationship between functional and non-functional requirements, architectures and design rationale which is the basis of a much more general decomposition operation. In essence, the basis of the characterisation is the observation that architectures address quality (or non-functional) requirements whereas components address functional requirements. Briefly, if we assume that we have isolated required qualities Q from required function F and if $A_Q(S_1, ..., S_m)$[7] is an architecture that discharges Q then the solution of

$$\mathsf{Env}(\mathsf{Soln}) \ \ \textbf{meets}_{\mathsf{G}} \ \ F \text{ subject to } Q$$

reduces to the solution of

$$\mathsf{Env}(A_Q(S_1, ..., S_m)) \ \ \textbf{meets}_{\mathsf{G}} \ \ F$$

In practice, if the functional requirements can be expressed as a conjunction, $F = F_1 \wedge \ldots \wedge F_n$, then we can build a matrix relationship between the S_i and the F_j by which to identify their respective contributions.

The importance of this characterisation is that it allows us to progress from a process based on a single monolithic requirement to *independent modular sub-developments* corresponding to some decomposition thereof. We note that:

- as illustrated in Figure 11, the cost of the transformation is an Architectural Analysis phase that (i) develops an appropriate architecture and (ii) finds a suitable partitioning to enable the subsequent modular sub-developments;
- each of the independent modular sub-developments may still be susceptible to Royce's risk managed development model. Indeed, should they be sufficiently simple, then his simplest development model, that of Figure 4(1), might once again apply[8].

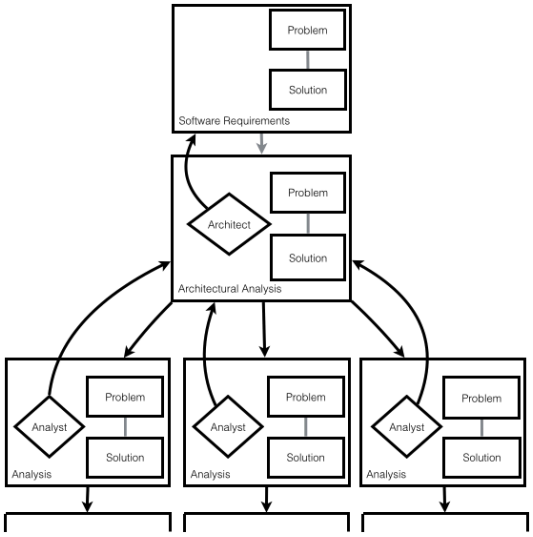

Fig. 11. In which an Architectural Analysis step allows the decomposition of complex requirements into modular sub-developments. With the correct choice of architecture, each module can be developed independently, such as occurs in agile processes. This links Royce's waterfall model with Ambler sophisticated scalable agile processes.

It is important to recall the difference between the way architectures are dealt with in plan-driven process models compared to the Agile version, in the context of DAD SCRUM, for instance: only an initial vision is set at inception time (i.e., before all sprints start) which is the subject of adaptation as a result of work within sprints, with the Architecture Owner working collaboratively with the Team to develop knowledge and a shared understanding. So the architecture emerges over time, starting from the Architecture Owner's initial vision, through increments which reflect the growing knowledge of both Architecture Owner and Team.

VI. Related work

Some of the ideas in our paper resonate with Jeffrey's work from the mid-90s ([15]) in which he argues the necessity to account for aspects of

[6]The definition of interfere is technical and is omitted for brevity. Details in [13]

[7]We have simplified the form of A_Q for brevity; see [12] for full details.

[8]Even if their complexity remains high, it may be that further architectural analysis can be applied to further simplify the requirements.

the human role in projects 'even in situations that seem purely technical.' One aspect of his argument, which is particularly relevant, is the observation that, as project teams become large, diverse 'communities' are formed, each with their own 'internal logic', which cause them to 'see the overall system in terms of the part they are creating.' Jeffrey suggests as mitigation

> every time, at every level, when the output of a process is input to some other subcommunity, include a member of the 'customer' community on the team doing the task to be the provider of reality checks for the team.

This is consistent with what our theory suggests as choice and role of validators among related development activities.

A recent survey of process tailoring [16] has identified 49 criteria and 20 related measures which have been observed or reported in the practice of process tailoring. Although a useful starting point and a good indication to its relevance in current practice, this survey falls short of providing any insight as to the interplay between those criteria and measures, their applicability or effects. It may be an interesting exercise for future work to apply our theory to try and explicate such relationships.

The Incremental Commitment Model (ICM) [4], [5] is a fairly recent generic software development process model which includes explicit consideration of process adaptation: risk-driven decision points are included to help practitioners tailor the process to the needs of their own projects/organisations by applying guiding risk patterns provided. Both model and patterns are based on the authors' own experience and understanding of best features of other mainstream process models, such as Royce's and the Agile models we have discussed in this paper. It is not clear at this point in time the extent such a model has been adopted and/or evaluated in practice. Differently from our approach, this is a full process model developed on a purely experiential basis, rather than a theory-based process pattern which can be used as an instrument of analysis, as is the intent of our approach.

Other work (e.g., [24]) looks at how to extend process modelling languages to include explicit notation for variation points. Such work fits within the extensive literature on software process modelling, which according to [6] relates to two main categories: descriptive models, with their clear identification of activities, roles, responsibilities and input/output artefacts; and executable models, that allow simulation and enacting of the modelled process ([30]). Both categories facilitate human understanding and communication and support process management and improvement. Our proposal may contribute to providing descriptive models with associated heuristics: our theoretical interpretation of process elements allows us to reason about resource expenditure and risk mitigation [17].

VII. Conclusion

In this paper, we have interpreted a diverse collection of software development processes as problem solving processes within a design theoretic framework. The scope of the interpretation was broad, ranging from traditional plan-driven processes to the most modern Agile ones. To do so, we have used a single building block – the POE Process Pattern – to capture process elements such as sense-making, solution design, and validation. When expressed in terms of the pattern, information flow and the relationship between actors in the processes are clarified as is resource usage and developmental risk.

Based on a single building block, the regularity of representation allows relationships between different process models to be seen – even in this proof-of-concept work, we already see the relationships between elements, and the structure, of Royce's early process models and those of modern Agile approaches. From our interpretation we have been able to explain why, in volatile contexts, Royce's model fails, which corresponds to observations of practice. We also argue, from first principles, that Agile's scalability to large systems requires an Architectural Analysis phase to be included taking their expression beyond Royce's early process models; this also agrees with Ambler's thoughts on scaling Agile processes.

In future work, we will extend our initial qualitative analysis through the definition of process metrics with the intention of imbuing our interpre-

tations with a predictive capability. Producing fully detailed, predictive models of software processes is one such challenge which, if met, would allow parametrisation against real world instances leading to a falsifiable general theory of software engineering processes.

ACKNOWLEDGMENTS

We thank the reviewers for their very helpful comments.

REFERENCES

[1] Agility. Agile alliance. http://www.agilealliance.org/home, 2004. Last accessed November 2004.

[2] S. W. Ambler. Beyond functional requirements on agile projects. *Dr. Dobb\'s Journal*, 33(10):64–66, 2008.

[3] S. W. Ambler. Agile architecture: Strategies for scaling agile development. Webpage, 2012.

[4] B. Boehm. An initial process decision table and a process evolution process. In *Proceedings of the 2014 International Conference on Software and System Process*, pages 187–188. ACM, 2014.

[5] B. Boehm, J. Lane, and S. Koolmanojwong. A risk-driven process decision table to guide system development rigor. In *Proceedings of the 19th International Conference on Software Engineering, Singapore (July 2009)*, volume 162, 2009.

[6] G. Canfora, F. García, M. Piattini, F. Ruiz, and C. A. Visaggio. A family of experiments to validate metrics for software process models. *Journal of Systems and Software*, 77(2):113–129, 2005.

[7] P. Deemer, G. Benefield, C. Larman, and B. Vodde. A lightweight guide to the theory and practice of scrum (version 2.0). Technical report, Technical report, http://www. scrumprimer. org, 2012.

[8] B. Fitzgerald, K.-J. Stol, R. O'Sullivan, and D. O'Brien. Scaling agile methods to regulated environments: An industry case study. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 863–872. IEEE Press, 2013.

[9] G. K. E. Gentzen. Untersuchungen über das logische schließen. *Mathematische Zeitschrift*, 39:176–210, 1935.

[10] J. G. Hall and L. Rapanotti. Software engineering as the design theoretic transformation of software problems. *Innovations in Systems and Software Engineering*, 8(3):175–193, 2012. DOI 10.1007s11334-011-0171-2.

[11] J. G. Hall and L. Rapanotti. Beauty in software engineering. *IEEE Computer*, 46(2):85–87, February 2013.

[12] J. G. Hall and L. Rapanotti. A general theory of engineering: Thinking bigger than software. Technical Report 2015/01, Computing and Communications Department, The Open University, UK, 2015.

[13] J. G. Hall, L. Rapanotti, and M. Jackson. Problem Oriented Software Engineering: solving the package router control problem. *IEEE Trans. Software Eng.*, 34(2):226–241, March 2008 2008.

[14] IFTTT. ifttt.com, Last Accessed, 26 Feb, 2015.

[15] H. J. Jeffrey. Addressing the essential difficulties of software engineering. *Journal of Systems and Software*, 32(2):157–179, 1996.

[16] G. Kalus and M. Kuhrmann. Criteria for software process tailoring: a systematic review. In *Proceedings of the 2013 International Conference on Software and System Process*, pages 171–180. ACM, 2013.

[17] D. Kaminsky and J. G. Hall. Towards process design for efficient organisational problem solving. In *Proceedings of BUSTECH 2015*, 2015.

[18] J. Kerievsky. Stop over engineering. *Software Development, April*, 2002.

[19] P. Kruchten. *The Rational Unified Process: An Introduction (2nd Edition)*. Addison-Wesley Professional, 2 edition, Mar. 2000.

[20] T. Martínez-Ruiz, J. Münch, F. García, and M. Piattini. Requirements and constructors for tailoring software processes: a systematic literature review. *Software Quality Journal*, 20(1):229–260, 2012.

[21] M. Mc Hugh, O. Cawley, F. McCaffcry, I. Richardson, and X. Wang. An agile v-model for medical device software development to overcome the challenges with plan-driven software development lifecycles. In *Software Engineering in Health Care (SEHC), 2013 5th International Workshop on*, pages 12–19. IEEE, 2013.

[22] L. Osborne, J. Brummond, R. D. Hart, M. Zarean, and S. M. Conger. Clarus: Concept of operations. Technical report, 2005.

[23] O. Pedreira, M. Piattini, M. R. Luaces, and N. R. Brisaboa. A systematic review of software process tailoring. *ACM SIGSOFT Software Engineering Notes*, 32(3):1–6, 2007.

[24] R. M. Pillat, T. C. Oliveira, and F. L. Fonseca. Introducing software process tailoring to bpmn: Bpmnt. In *Software and System Process (ICSSP), 2012 International Conference on*, pages 58–62. IEEE, 2012.

[25] G. F. C. Rogers. *The Nature of Engineering: A Philosophy of Technology*. Palgrave Macmillan, 1983.

[26] G. Rong. Are we ready for software process selection, tailoring, and composition? In *Proceedings of the 2014 International Conference on Software and System Process*, pages 185–186. ACM, 2014.

[27] G. Rong, B. Boehm, M. Kuhrmann, E. Tian, S. Lian, and I. Richardson. Towards context-specific software process selection, tailoring, and composition. In *Proceedings of the 2014 International Conference on Software and System Process*, pages 183–184. ACM, 2014.

[28] W. Royce. Managing the development of large software systems. In *Proceedings of IEEE WESCON*, volume 26, pages 1 – 9, 1970.

[29] B. Ur, E. McManus, M. Pak Yong Ho, and M. L. Littman. Practical trigger-action programming in the smart home. In *Proceedings of the 32nd annual ACM conference on Human factors in computing systems*, pages 803–812. ACM, 2014.

[30] H. Zhang, B. Kitchenham, and D. Pfahl. Software process simulation modeling: an extended systematic review. In *New Modeling Concepts for Today's Software Processes*, pages 309–320. Springer, 2010.