**INTERNATIONAL CONFERENCE ON ENGINEERING DESIGN, ICED'07**
28 - 31 AUGUST 2007, CITE DES SCIENCES ET DE L'INDUSTRIE, PARIS, FRANCE

# AN ONTOLOGICAL MODEL OF EMERGENT DESIGN IN SOFTWARE ENGINEERING

**John S Gero[1] and Udo Kannengiesser[2]**
[1]Krasnow Institute for Advanced Study and Volgenau School of Information Technology and Engineering, George Mason University, USA
[2]NICTA, Australia

## ABSTRACT
This paper proposes an ontological account of a recent model of software engineering: emergent design. This model augments traditional software design approaches by accounting for incremental and unexpected modifications of the design state space. We show that the principles of emergent design are consistent with a model of designing as a reflective conversation. We use the situated function-behaviour-structure (FBS) framework to represent the activities driving emergent design. This is a basis for increasing understanding and acceptance of emergent design.

*Keywords: Emergent Design, Software Engineering, Ontology*

## 1 INTRODUCTION
Emergent design is a model of software design that includes the iterative development of software through gradual improvements and incremental additions of new functionality during the process of designing. This methodology is often seen as the opposite of "up-front design", or the "waterfall model", where designing is assumed to proceed in a fixed, single sequence of stages without iterations. The waterfall model has often been criticised as too inflexible with respect to dynamic changes in the requirements, and as too difficult to implement when feedback from downstream stages is needed to complete an upstream stage. The benefits of a more adaptive approach such as emergent design include increased ability to incorporate changes in the design requirements, easier elimination of coding errors and reduced development times. A number of methods have been developed to support emergent design, including extreme programming, test-driven development and refactoring. A number of tools, such as JUnit for Java development, are available that facilitate these methods.
Despite the advantages and increasing popularity of emergent design, the waterfall model still seems to be the predominant paradigm in software engineering [14]. One of the underlying reasons is the widely held view that emergent design de-emphasises "design" in favour of "coding", making this approach seem unprincipled and lacking top-down guidance. However, emergent design is increasingly seen as a complement rather than an alternative to top-down design approaches [4].
In this paper, we present an ontological model that assists in gaining a better understanding of emergent design by representing the activities driving the design. Specifically, we describe emergent design using Gero and Kannengiesser's [9] situated function-behaviour-structure (FBS) framework. This extends recent efforts in mapping software design onto Gero's [7] original FBS framework [13]. Specifically, our extension provides an ontology for connecting some of the principles of emergent design with Schön's [15] model of designing as a "reflective conversation". This can increase the acceptance of emergent design in research and practice.

## 2 EMERGENT DESIGN
"Emergent" is an adjective that has been used to characterise the process of designing in two ways. In one way, it refers to the gradual formation of a design solution as a global pattern resulting from a set of iterative design activities. Most models of emergent design in software engineering are based on this idea, as evident from synonymous terms currently employed for emergent design such as "continuous" or "evolutionary" design [18]. The technique of refactoring, for example, has been described as "test – small change – test – small change – ..." [5].

The second way of understanding the notion of "emergent" in connection with design is in the sense of new design properties "arising unexpectedly" during the design process, "calling for prompt action" (quotations taken from *Merriam-Webster* definitions of the word "emergent"). This way of understanding emergent design is less explored; perhaps because unexpected changes in requirements and designs have traditionally been regarded as undesirable.

Both aspects of the term "emergent" include the notion of local interactions that drive the gradual and/or unexpected formation of global patterns. While this notion has been well studied in the areas of complex systems and Artificial Life [11], less work has been done in mapping it to emergent design in software engineering.

One of the best known and most comprehensive accounts of emergent design is the Agile Manifesto [6]. It is the result of an attempt to build a common ground for the growing variety of approaches that support emergent design using "agile" methodologies and tools. The Agile Manifesto characterises agile software development as valuing

1. individuals and interactions over processes and tools,
2. working software over comprehensive documentation,
3. customer collaboration over contract negotiation, and
4. responding to change over following a plan. [6]

A more detailed description is provided through twelve principles [6]:

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.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference for the shorter timescale.
4. Business people and developers work together daily throughout the project.
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 with and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity – the art of maximising the amount of work not done – is essential (since it facilitates changing or adding to the software).
11. The best architectures, requirements and designs emerge from self-organising teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behaviour accordingly.

The Agile Manifesto can be viewed to capture the two aspects of the term "emergent", Table 1. Specifically, values 2 and 3, and principles 1, 3, 4, 6, 7, 8, 10 and 12, are understood to characterise emergent design in the sense of "continuous" design. Value 4 and principles 1, 2, 4, 6, 11 and 12 address emergent design in the sense of unexpected changes throughout the process of designing. However, the Agile Manifesto does not provide an ontological framework to represent the two aspects of emergent design more formally, and to understand their drivers in terms of the interactions of individual designers. These drivers are only superficially addressed in value 1 and principle 5. An ontological framework could also clarify the connection of "technical excellence and good design" (principle 9) with unexpected, bottom-up design changes. This is crucial for understanding and accepting emergent design as a complement rather than the opposite approach to top-down design.

## 3 SITUATEDNESS IN DESIGN

There is now a significant amount of research in various design disciplines that studies designing as an interaction between designers and their environment. The paradigm on which this research is based is called situatedness. It builds on the foundational concepts provided by Dewey [3] and Bartlett [1], and the more recent work by Clancey [2] on situated cognition.

*Table 1. Mappings (indicated using "X" symbols) between the Agile Manifesto and the*

*two aspects of emergent design*

| Values (V) and principles (P) of the Agile Manifesto | Aspect of continuity | Aspect of unexpectedness |
|---|---|---|
| V 1 | | |
| V 2 | X | |
| V 3 | X | |
| V 4 | | X |
| P 1 | X | X |
| P 2 | | X |
| P 3 | X | |
| P 4 | X | X |
| P 5 | | |
| P 6 | X | X |
| P 7 | X | |
| P 8 | X | |
| P 9 | | |
| P 10 | X | |
| P 11 | | X |
| P 12 | X | X |

Most influential for recognising designing as a situated activity has been Schön's [16] notion of designing as a "reflective conversation with the materials of a design situation", i.e. an interaction between designers and their intermediate design representations. Designers change their view of the current design as a result of them generating and interpreting representations of the design. Design concepts are the consequences, expected or unexpected, of the designer moving through the state space of possible designs. Every step of the designer through that space is seen as a "move experiment", which is then taken as the basis for both evaluating previous design concepts and generating new design concepts. Schön [16] describes the reflective process schematically as "seeing-moving-seeing".

Schön and Wiggins [17] illustrate this concept using the following excerpt from the design protocol of a school design task, performed by an architect:

> "I had six of these classroom units but they were too small to do much with. So I changed them to this more significant layout (the L-shapes). It relates grade one to two, three to four, and five to six grades, which is more what I wanted to do educationally anyway. What I have here is a space which is more of a home base. I'll have an outside/inside which can be used and an outside/outside which can be used – then that opens into your resource library/language thing."

We can apply Schön's schema of "seeing-moving-seeing": The first "seeing" allows the designer to observe and evaluate the current design, resulting in the recognition that the classroom units were "too small to do much with". The upper part of Figure 1, taken from [17], shows the designer's initial drawing. "Moving", i.e. the act of drawing a modified design representation (lower part of Figure 1), aims to solve this problem. The designer's repeated "seeing" of the results of this drawing finally includes a second judgment, that the initial problem has now been solved. It also includes the recognition of a set of unintended, desirable consequences of the "move", namely the spatial grouping of proximate grades, and the creation of a "home base" and of two kinds of spaces ("outside/inside" and "outside/outside").

Other empirical studies of designers support the view of designing as being driven by the designers' interactions. Suwa et al. [19] noted, from their protocol analyses of architects, a correlation of unexpected discoveries in sketches with the invention of new issues or requirements during the design process. They concluded that "sketches serve as a physical setting in which design thoughts are constructed on the fly in a situated way". Guindon's [10] protocol analyses of software engineers, designing control software for a lift, revealed that design strategies are largely opportunistic rather than top-down. She stated that designing is characterised by frequent discoveries of new requirements

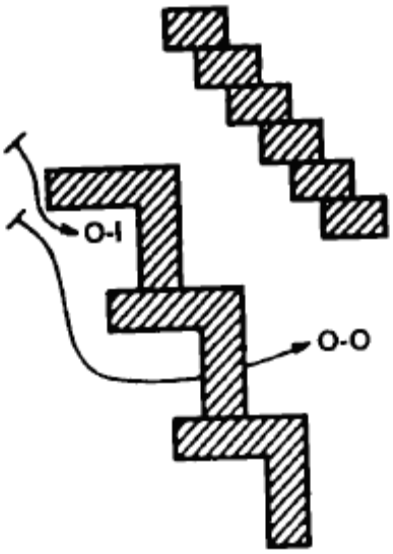


*Figure 1. Drawings of the layout of a classroom (taken from [17])*

interleaved with the development of new partial design solutions. As Guindon puts it, "designers try to make the most effective use of newly inferred requirements, or the sudden discovery of partial solutions, and modify their goals and plans accordingly". Woodcock and Bartlett [20] have explicitly applied Schön's model to software development. Their experimental studies show that software code is used in a similar way to sketches in architectural design, namely as the designer's partner for a reflective conversation. Nerur and Balijepally [15] have drawn conceptual parallels between agile software development and models of other design disciplines, based on the notion of reflection and related ideas.

# 4 AN ONTOLOGY OF SITUATED DESIGN

## 4.1 A Model of Situatedness

Gero and Kannengiesser [9] have modelled situatedness as the recursive interaction between three worlds: the external world, the interpreted world and the expected world, Figure 2(a).

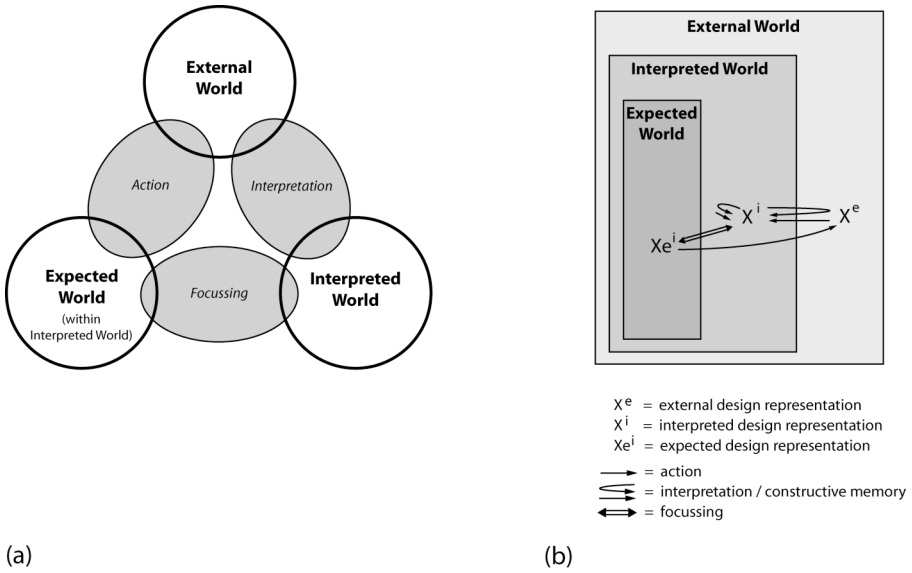


*Figure 2. Situated designing as the interaction of three worlds: (a) general model, (b) specialised model for design representations (after [9])*

- The *external world* is the world that is composed of representations outside the designer. This world is the designer's medium for communication, either with other designers or with the

designer themselves. The latter kind of communication corresponds to Schön's [16] "reflective conversation with the materials of a design situation". The "materials" are represented in the external world and typically include iconic and symbolic representations of the design object.

- The *interpreted world* is the world that is built up inside the designer in terms of sensory experiences, percepts and concepts. It is the internal representation of that part of the external world that the designer interacts with. The kinds of design objects represented in the interpreted world depend on the unique experience of the individual designer.
- The *expected world* is the world the imagined actions of the designer are expected to produce. It is the environment in which the effects of actions are predicted according to current goals and interpretations of the current state of the world. The expected world corresponds to the designer's current design state space, i.e. the state space of all potential design solutions currently considered by the designer. This world is located within the interpreted world, as all goals and expectations can be viewed as interpreted representations of potential future designs.

The three worlds are linked together by three classes of processes:

- *Interpretation* transforms variables which are sensed in the external world into the interpretations of sensory experiences, percepts and concepts that compose the interpreted world. This process includes Schön's [16] notion of "seeing".
- *Focussing* takes some aspects of the interpreted world and uses them as goals for the expected world that then become the basis for the suggestion of actions. These actions are expected to produce states in the external world that reach the goals. This process uses the results of qualitative judgments of the design. If the interpretation is judged to be useful or required for the current design task, focussing integrates that interpretation into the design state space.
- *Action* is an effect which brings about a change in the external world according to the goals in the expected world. This process corresponds to what Schön [16] calls "moving", including, for instance, the act of producing a design sketch or writing a piece of software code.

Figure 2(b) presents a specialised form of this view with the designer (as the internal world) located within the external world and placing general classes of design representations into the resultant nested model. The set of expected design representations ($Xe^i$) form the design state space. This state space can be modified during the process of designing by transferring new interpreted design representations ($X^i$) into the expected world and/or transferring some of the expected design representations ($Xe^i$) out of the expected world. This leads to changes in external design representations ($X^e$), which may then be used as a basis for re-interpretation, changing the interpreted world.

Novel interpreted design representations ($X^i$) may also be the result of constructive memory, which can be viewed as a process of interaction among design representations within the interpreted world rather than across the interpreted and the external world. Constructive memory is best exemplified by a paraphrase of Dewey by Clancey [2]: "Sequences of acts are composed such that subsequent experiences categorize and hence give meaning to what was experienced before". The implication of this is that memory is not laid down and fixed at the time of the original sensate experience but is a function of what comes later as well. Memories can therefore be viewed as being constructed in response to a specific demand, based on the original experience as well as the situation pertaining at the time of the demand for this memory. Therefore, everything that has happened since the original experience determines the result of memory construction. Each memory, after it has been constructed, is added to the existing knowledge (and becomes part of a new situation) and is now available to be used later, when new demands require the construction of further memories. These new memories can be viewed as new interpretations of the augmented knowledge.

In Figure 2(b), both interpretation and constructive memory are represented as "push-pull" processes. This emphasises the role of the designer's individual experience that constructs or "pulls" new design concepts to match first-person knowledge rather than just replicates or "pushes" what can be seen as third-person encodings [8].

### 4.2 The Situated FBS Framework

Design representations can be viewed as describing any of three aspects of a design: function (F), behaviour (B) and structure (S). These notions are the basic constituents of the function-behaviour-structure (FBS) ontology [7]. They are defined as follows:

- *Function* (F) of an object is defined as its teleology, i.e. "what the object is for". For example, some of the functions of a window include "to provide view", "to provide daylight" and "to

provide rain protection". Functions of a word-processing software include "to provide a display for text", "to store text" and "to allow manipulation of text". In the Rational Unified Process (RUP) terminology of software engineering, the notion of function corresponds to the stakeholders' needs, use case surveys and business cases [13].

- *Behaviour* (B) of an object is defined as the attributes that are derived or expected to be derived from its structure (S), i.e. "what the object does". Using the window example, behaviours include "thermal conduction" and "light transmission". In the word processor example, behaviours include "time required for saving a document" and "ability to select text fragments". In RUP, the (required or measured) behaviour of a piece of software corresponds to performances represented formally as specifications, use case models or test cases, or less formally as measurements, reports or test results [13].
- *Structure* (S) of an object is defined as its components and their relationships, i.e. "what the object consists of". Examples of structure properties of a window include "glazing length", "glazing height" and "type of glass". Structure of the word processor, like all other software artefacts [13], includes its software architecture and code.

Using the FBS ontology, we can specialise the model of situated designing shown in Figure 2(b) by replacing the variable X, which stands for design representations in general, with the more specific representations F, B and S. Gero and Kannengiesser's [9] situated FBS framework is based on this idea, Figure 3. In addition to using external, interpreted and expected F, B and S, this framework uses explicit representations of external requirements given to the designer by another person (usually the customer). Specifically, there are external requirements on function ($FR^e$), external requirements on behaviour ($BR^e$), and external requirements on structure ($SR^e$). The situated FBS framework also introduces the process of comparison between interpreted and expected behaviour, and a number of processes that transform interpreted structure into interpreted behaviour, interpreted behaviour into interpreted function, expected function into expected behaviour, and expected behaviour into expected structure.

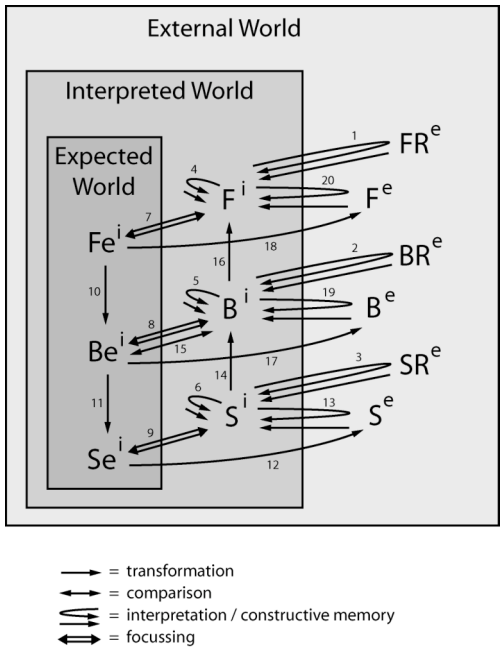


*Figure 3. The situated FBS framework (after [9])*

The 20 processes labelled in Figure 3 can be mapped onto eight fundamental design steps. Table 2 shows how Kruchten [13] has related some RUP activities to these steps, based on a previous, less detailed version of the FBS framework.

1. *Formulation*: consists of processes 1 – 10. It includes interpretation of external requirements, given to the designer by a customer, as function, behaviour and structure, via processes 1, 2 and 3. Requirements are also constructed as implicit requirements generated from within the designer, using constructive memory (processes 4, 5 and 6). Focussing transfers a subset of the (explicitly and implicitly) required function, behaviour and structure into the expected world

(processes 7, 8 and 9). Processes 1 – 9 represent activities that populate the interpreted and expected worlds with design concepts, providing the basis for subsequent transformations of these concepts. The transformation of expected function into expected behaviour, via process 10, is commonly understood to be the focus of requirements engineering.

2. *Synthesis*: consists of process 11 to generate a structure that is expected to meet the required behaviour, and the externalisation of that structure via process 12. The result of the synthesis of software is usually displayed on the computer screen or represented on paper as code, pseudo-code, UML diagrams or symbols of other languages.
3. *Analysis*: consists of interpretation of externalised structure (process 13) and the derivation of behaviour from that structure (process 14). In most cases, these activities are performed by the software designer's Integrated Development Environment (IDE).
4. *Evaluation*: consists of process 15 that includes a comparison of expected behaviour and behaviour derived through analysis.
5. *Documentation*: produces an external representation of the final design solution for purposes of communicating that solution in terms of structure (process 12), and, optionally, behaviour (process 17) and function (process 18).
6. *Reformulation type 1*: consists of focussing on different structures than previously expected (process 9). Precursors of this process are the interpretation of external structure (process 13), constructive memory of structure (process 6) or the interpretation of new requirements on structure (process 3).
7. *Reformulation type 2*: consists of focussing on different behaviours than previously expected (process 8). Precursors of this process are the derivation of behaviour from structure (process 14), the interpretation of external behaviour (process 19), constructive memory of behaviour (process 5) or the interpretation of new requirements on behaviour (process 2).
8. *Reformulation type 3*: consists of focussing on different functions than previously expected (process 8). Precursors of this process are the ascription of function to behaviour (process 16), the interpretation of external function (process 20), constructive memory of function (process 4) or the interpretation of new requirements on function (process 1).

*Table 2. The fundamental design steps in the FBS framework mapped onto RUP (after [13])*

| FBS design step | RUP activity |
|---|---|
| Formulation | Business modelling, Requirements definition |
| Synthesis | Analysis and design, Implementation |
| Analysis | Testing, Review activities |
| Evaluation | Assessment activities |
| Documentation | Implementation, Deployment |
| Reformulation type 1 | Refinement of design/code, Refactoring, Fixing defects in design/code |
| Reformulation type 2 | Scope management, Requirements change |
| Reformulation type 3 | Change in needs |

The "waterfall" model of software design can be mapped onto the first five fundamental design steps in the situated FBS framework. In this model, the numbering of the five design steps can be thought of as specifying a linear sequence of execution.

The three types of reformulation are an additional set of design steps that break the top-down and linear character of the "waterfall" model. They address the two aspects of emergent design: Modifications in the expected world after the initial formulation step correspond to the notion of unexpectedness. The three types of reformulation allow for unexpected changes both at the level of the design solution (i.e., structure) and at the level of the design requirements (i.e., behaviour and function). All three design steps capture the notion of iterative formation of design solutions, as incrementally modified design requirements lead to new cycles of synthesis, analysis and evaluation.

## 5 EMERGENT DESIGN IN THE SITUATED FBS FRAMEWORK

Using the situated FBS framework, we can describe the notion of reflective interaction in emergent design in more detail. Specifically, this interaction is located at the level of structure, and involves the following four processes:

1. Constructive memory: The software designer, after becoming aware of bugs, poor architectural or code structure or a functionality to be added, generates a new structure that potentially addresses these issues. Figure 4 highlights process 6 to represent this activity.

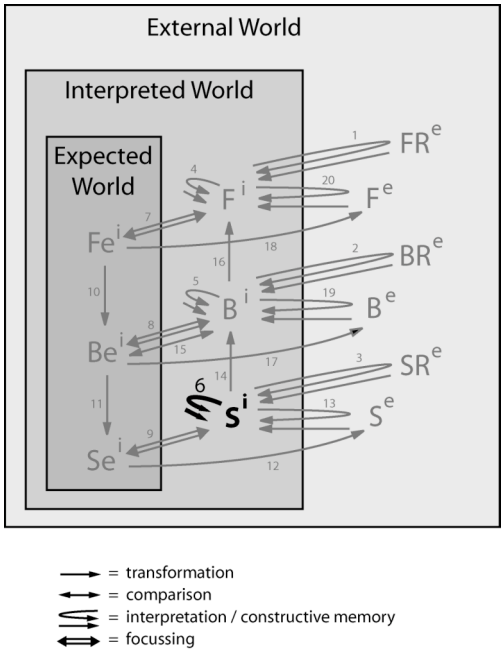


*Figure 4. Step 1 of reflective conversation with software structure: Constructive memory (process 6)*

2. Focussing: The software designer decides to take the new structure as a basis for the next design moves, including it in the current design state space. Figure 5 represents this activity as process 9.

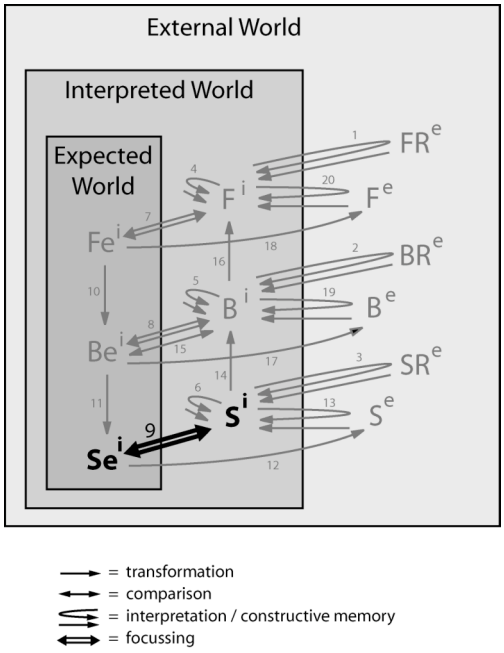


*Figure 5. Step 2 of reflective conversation with software structure: Focussing (process 9)*

3. Action: The software designer produces an externally visible account of the new structure, e.g.

by fixing bugs, refactoring or adding pieces of code. Figure 6 shows this activity as process 12.

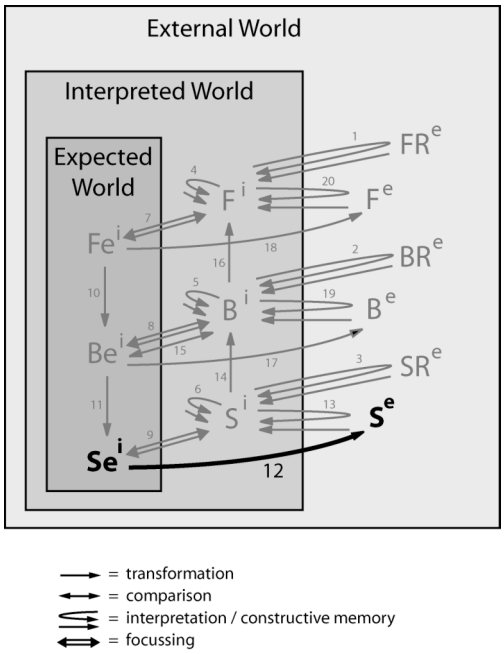


*Figure 6. Step 3 of reflective conversation with software structure: Action (process 12)*

4. Interpretation: The external representation of structure is interpreted by the software designer or compiled by an IDE. This is represented as process 13 in Figure 7.

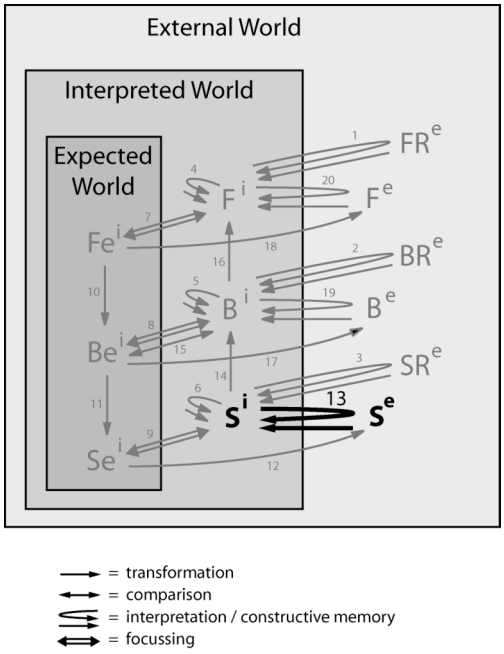


*Figure 7. Step 4 of reflective conversation with software structure: Interpretation (process 13)*

These four design steps compose what can be thought of as a cycle of reflective conversation. The result of this cycle, namely the new software structure created, can then be taken as a basis for discovering additional, unexpected consequences of the new structure, by deriving new (interpreted) behaviours ($B^i$). This can be carried out by an IDE that executes or debugs the compiled code (i.e., the interpreted structure ($S^i$)) for providing a representation of how the software behaves as a basis for testing. This is shown as process 14 in Figure 8.

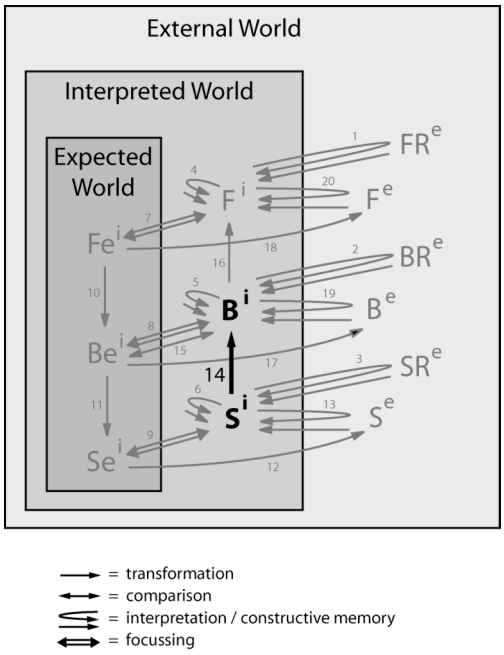


*Figure 8. Consequences of reflective conversation with software structure: Deriving new behaviour from structure (process 14)*

## 6 EXAMPLE

This Section illustrates the ontological steps of reflective conversation in software engineering. Figure 9 shows a sequence of class diagrams, adapted from [12]. For the present example, we take these diagrams as a software designer's intermediate (external) representations of parts of a new loan management system to be designed. Figure 9 can be seen as a description equivalent to Figure 1, replacing the sequence of architectural sketches with diagrams commonly used by software designers.

The designer, having drawn the diagram at the top of Figure 9, realises that the large number of constructors in the Loan class leads to reduced clarity of that class, and potential difficulties in maintaining the code. Constructive memory, Figure 4, is used to refactor this representation by replacing most of the constructors with creation methods. Focussing, Figure 5, is the process that transfers the result of this activity into the current design state space. Action, Figure 6, then produces an external representation of the refactored design, depicted in the middle of Figure 9. Interpretation, Figure 7, terminates the cycle of reflective conversation, providing the basis for the IDE testing the software, and for the designer recognising the improvements in terms of better software structure and more meaningful naming of the different kinds of Loan objects. At this stage in our example, no unexpected consequences of the modified design representation occur. Behaviour is derived from structure and evaluated only by the IDE, for purposes of testing if the change has produced any bugs.

Despite the design improvements achieved so far, the Loan class still looks quite "messy". Constructive memory, Figure 4, commences a new cycle of reflective conversation by using the "Extract class" refactoring strategy [5] to modify the current design. Focussing, Figure 5, and action, Figure 6, lead to a new external representation, shown at the bottom of Figure 9. Interpretation, Figure 7, is used as a basis for confirming the absence of bugs as well as the expected benefit of the design move: a clear separation of the creation methods from the "core methods" of the Loan class. The designer also becomes aware of an additional, unexpected consequence of the new design, namely of increased extensibility through encapsulating the creation methods in a separate class. This is a new behaviour derived from the modified structure, Figure 8, which can be introduced in the design state space (via reformulation type 2) and used to drive further reflective conversations.

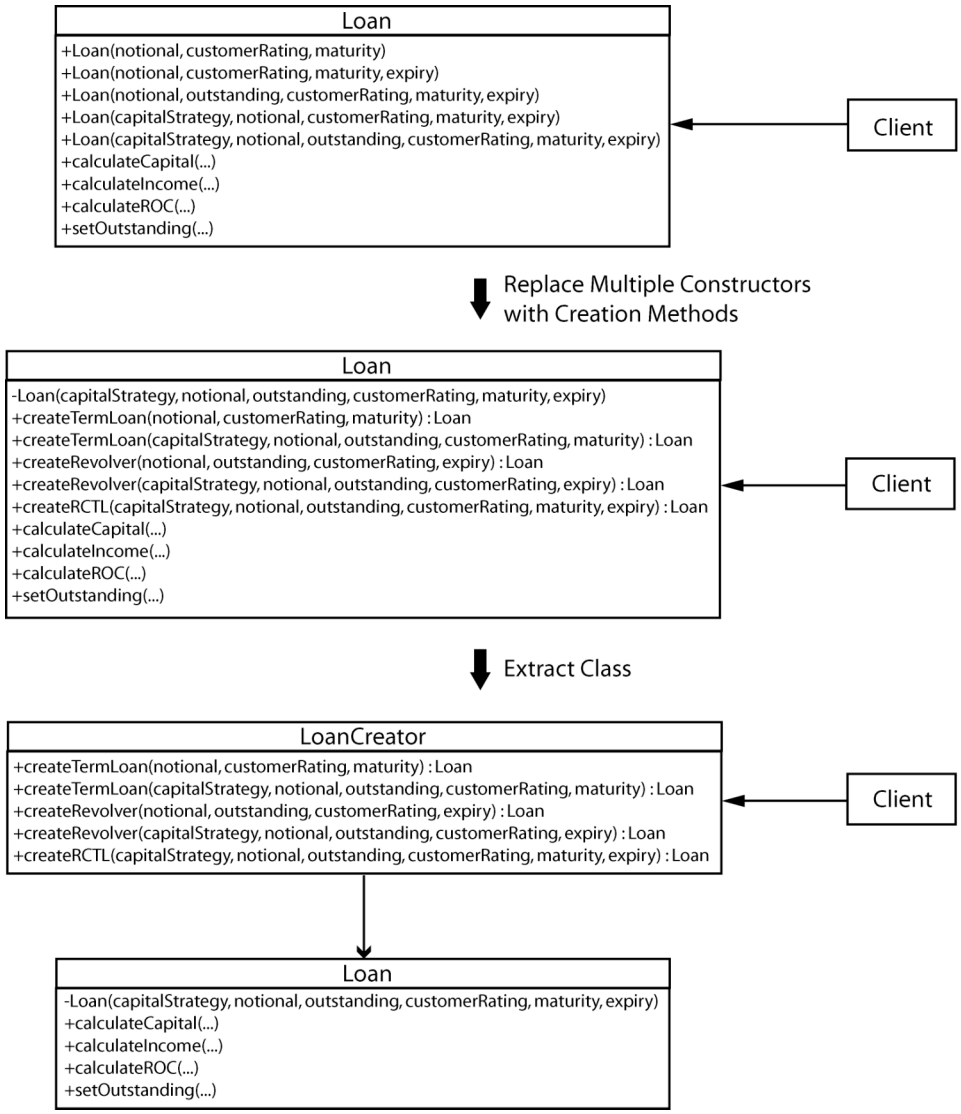


*Figure 9. Example of the external representation of a software design that evolves through refactoring (adapted from [12])*

## 7 CONCLUSION

We have presented an ontological framework that describes emergent design in software engineering. It captures two aspects of the notion of "emergent"; the gradual formation of design solutions and unexpected changes in solutions and requirements. Previous models of emergent design such as the Agile Manifesto address these two aspects in a less formal way. Their level of detail is not sufficient to describe what drives emergent design. Our framework uses Schön's model of designing as a reflective conversation to provide such a description. It is particularly useful for exploring the aspect of a designer's unexpected insights during the design. For example, as shown in Section 6, the framework accounts for the characteristic of refactoring as enabling "higher levels of understanding" of the design through "wiping the dirt off a window so you can see beyond" [5]. Our model can establish the grounds for further, empirical investigations of software design, and the emergent design paradigm in particular.

By connecting this paradigm to the traditional top-down approach, our model can show that emergent design supplements "waterfall" design principles. All design concepts, no matter whether they are expected or unexpected, are represented uniformly – as function, behaviour or structure. This allows for rigorous comparison of emerging designs against initial specifications. Design concepts that arise unexpectedly as a result of the designer's reflective conversation, can be turned into new expected requirements that are used for further moves through the design state space. We see our model as a step towards better understanding and greater acceptance of emergent design.

## ACKNOWLEDGEMENTS

This research is funded by the Australian Research Council, grant number DP0559885 and was carried out at the Key Centre of Design Computing and Cognition, University of Sydney.

## REFERENCES

[1] Bartlett F.C. *Remembering: A Study in Experimental and Social Psychology*, 1932 reprinted in 1977 (Cambridge University Press, Cambridge).
[2] Clancey W.J. *Situated Cognition: On Human Knowledge and Computer Representations*, 1997 (Cambridge University Press, Cambridge).
[3] Dewey J. The reflex arc concept in psychology. *Psychological Review*, 1896 reprinted in 1981, 3, 357-370.
[4] Fowler M. Is design dead?. In G. Succi and M. Marchesi (eds) *Extreme Programming Examined*, 2001, pp. 3-17 (Addison-Wesley, Boston).
[5] Fowler M. *Refactoring: Improving the Design of Existing Code*, 2004 (Addison-Wesley, Boston).
[6] Fowler M. and Highsmith J. The agile manifesto. *Software Development*, 2001, 9(8), 28-32.
[7] Gero J.S. Design prototypes: A knowledge representation schema for design. *AI Magazine*, 1990, 11(4), 26-36.
[8] Gero J.S. and Fujii H. A computational framework for concept formation for a situated design agent. *Knowledge-Based Systems*, 2000, 13(6), 361-368.
[9] Gero J.S. and Kannengiesser U. The situated function-behaviour-structure framework. *Design Studies*, 2004, 25(4), 373-391.
[10] Guindon R. Designing the design process: Exploiting opportunistic thoughts. *Human-Computer Interaction*, 1990, 5, 305-344.
[11] Holland J.H. *Emergence: From Chaos to Order*, 1998 (Oxford University Press, Oxford).
[12] Kerievsky J. *Refactoring to Patterns*, 2004 (Addison-Wesley, Boston).
[13] Kruchten P. Casting software design in the function-behaviour-structure framework. *IEEE Software*, 2005, 22(2), 52-58.
[14] Laplante P.A. and Neill C.J. "The demise of the waterfall model is imminent" and other urban myths. *ACM Queue*, 2004, 1(10), 10-15.
[15] Nerur S. and Balijepally V. Theoretical reflections on agile development methodologies. *Communications of the ACM*, 2007, 50(3), 79-83.
[16] Schön D.A. Designing as reflective conversation with the materials of a design situation. *Knowledge-Based Systems*, 1992, 5(1), 3-14.
[17] Schön D.A. and Wiggins G. Kinds of seeing and their functions in designing. *Design Studies*, 1992, 13(2), 135-156.
[18] Shore J. Continuous design. *IEEE Software*, 2004, 21(1), 20-22.
[19] Suwa M., Gero J.S. and Purcell T. Unexpected discoveries and s-inventions of design requirements: A key to creative designs. In J.S. Gero and M.L. Maher (eds) *Computational Models of Creative Design IV,* 1999, pp. 297-320 (Key Centre of Design Computing and Cognition, University of Sydney, Sydney, Australia).
[20] Woodcock A. and Bartlett R. Software authoring as design conversation. In P. Romero, J. Good, E. Acosta Chaparro and S. Bryant (eds) *Workshop of the Psychology of Programming Interest Group*, 2005, pp. 203-214 (University of Sussex, Brighton).

Contact: John S. Gero
Krasnow Institute for Advanced Study
George Mason University
USA
e-mail: john@johngero.com