- あなたのモデリングの対象は何ですか? 言葉を換えると, 例えばクラス図を描いたときにクラス (あるいはそのインスタンス) に対応するものはどこに存在しますか?
  - プログラムの中
  - 現実世界の中
    - ビジネス
    - デバイス
    - 法律
    - 数学
    - .....
  - 自分の頭の中
  - みんなの頭の中
  - どこにもない
  - .....
- UML (あるいは他のモデリング技法/記法でも構いませんが) の場合, 「クラス図」という名前から分かるように (少なくとも最初は) モデリングの対象はプログラムでした.
  - プログラムの中にある「クラス」をプログラムというコンテキスト (まさにテキスト!) から離れて見たいというモチベーションがあったように思います.
  - ここで「コンテキストから離れる」とは
    - プログラムのような1次元のコンテキストではないコンテキスト (例えば図のような2次元のコンテキスト) から見る (表現次元の拡張)
    - プログラムが要求するような些末 (!) な言語的/実装的制約を持たないコンテキストから見る (高度を上げる → 抽象化)
    - プログラムのように複数の異なる領域 (ドメイン) に関する記述を混在させるのではなく, 領域を切り出して, それらに対する記述を組み合わせる (視点の多様化)
      - 例えばCで書くということは, Cが提供するレジスタ・マシンという視点からすべてを理解/記述しなければならない
    - .....
  - 表現次元の拡張 → 図表記, 表表記などノーテーション
  - 抽象化 → 特定のプラットホーム上で動くことの放棄
  - 視点の多様化 → クラス図以外の図
- クラス図, 状態遷移図, シーケンス図, コンポネント図, アクティビティ図
  - これらは (少なくとも) 最初は計算機あるいはプログラムあるいはソフトウェアの中にある何かを直接表そうとしていたと言っていいでしょう.
  - 多分, プログラムの中にそのモデルが直接表す対応要素を見つけ出すことができます.
    - もしかしたらアクティビティ図はちょっと違っているかもしれません.
    - そしてユースケース図はもっと違っています (後述).
  - とすると, プログラムとモデルの違いはどこにあるのでしょうか.
    - この問いに対する答えによって, ソフトウェアのモデリングは次のような流れを生み出します.
      - 答え: 本質的な違いはない
        - じゃあ, モデルも動くべきである → モデル駆動開発
      - 答え: モデルの対象はプログラムよりももっと大きい
        - 例えば要求モデリング
        - 例えばビジネス・モデリング
        - 例えば.....
        - ユースケース図はその方向へのほんの小さい一歩かもしれない

- 一方, プログラミング, ソフトウェア開発の世界に目を転ずると.
  - 機能がすべて (functionalism)
    - ソフトウェア要求は最終的には「機能一覧」
      - 非機能要求もあるけれど, それはあくまでも機能としてはこぼれ落ちてしまう何かをまとめたもの
    - 見積もりは「機能点」(function point)
    - プログラミングは「機能の実現」
    - テストの対象は「機能が実現されているか否か」
  - ここで言う機能とは, 「ある状況においてある入力を与えるとある出力が得られる」ことです.
    - ここで言う機能主義とは, 「ソフトウェアは機能の集合から成り立っている」という考え方です.
    - 最近ではfeatureなどということもあります (プロダクトライン, Feature-driven developmentなど) が, 今日お話しする内容からは本質的には同じと言っていいと思います.
  - そのもっと前はソフトウェアとは「計算」でした.
    - 各県庁所在地/政令指定都市ごとにあった「XX電子計算センタ」
    - 計算とは数値を対象とした機能です.
    - 計算 → 機能 → ...?
  - もちろん最終的に機能が重要なもののひとつであるのは確かですが, 実際のソフトウェア開発の現場で, 重要な作業は機能を確定し, 機能を実現すること (だけ) ですか?
- (ある種の) ソフトウェア開発において, 「機能」を直接の対象としない作業が必要です.
  - ソフトウェア開発を何らかの問題解決過程だと考えると, 「機能」は解決の記述です.
  - しかし多くの場合, 「問題」の記述が必要なのです.
    - 問題はかつては自明であるように見えたり, 解決が見つかってからソフトウェアに持ち込まれることが多かったかもしれません.
    - つまり多くの場合, ソフトウェアは解決 (機能) の実現だけをしていれば良かった (少なくともそう思われていた) のです.
    - しかし, 今や問題そのものを見つけること, 問題を解決することが (ある種の) ソフトウェアには求められています.
  - しかし今までのプログラムには (たいてい) 問題を記述する能力はありません.
  - ソフトウェア開発に関わるさまざまなドキュメント・フォーマットにしても似たり寄ったりです.
  - そこで最近はビジネス・モデリング, エンタプライズ・アーキテクチャなどが盛んになりつつあります. BABOK (Business Analysis Body of Knowledge) などというものさえあります.
    - これらは業務系システムの場合に限られますが, 組込系システムにおいても (すべてがネットワーク化されつつある現在では) 同様だと言えるでしょう.
    - ビジネス・モデリングは, いわゆる要求定義/要求開発などとは必ずしも同じでないことに注意してください. 要求定義/要求開発が解決策の案をソフトウェアに対する機能という形でユーザや顧客から引き出す/作り出す作業であるのに対して, ビジネス・モデリングは機能 (解決) を記述するものではないからです.
    - ただし, 問題の記述を誰がやるべき作業かは, ここではあえて問いません (後述?)
    - ビジネス・モデリングは要求定義/要求開発の一部分であるとする考え方もあるでしょうが, 私はここでは別の考え方を選ぼうと思います.
  - ここで「ある種の」といっているのは, 既に機能がおおかた確定しているソフトウェアもあるからです (例えばプロトコル・スタック). ただし「ある種の」が特別とは限らず, むしろ非常に多くの業務系/組込系のソフトウェアが相当すると思います.
- 機能を対象としない作業 = 問題を記述する作業をビジネス・モデリングを例にとって, 話を進めたいと思います.
  - あなたがビジネス・モデリングを行い, ビジネス・モデルが作られたとしましょう. このビジネス・モデルはその後のソフトウェア開発において, どのように使われるでしょう?

    - 例えばビジネス・モデリングにおいて, そもそも何がゴールかを表すために, バランスド・スコア・カード (BSC) とそのための戦略マップ (以後は併せてBSC戦略マップと言います) というものを比較的初期の段階で作ります.
    - BSCは組織の方向性を財務, 顧客, 内部プロセス, 学習という四つの視点からの (定量的な) 目標として表すものです. BSCのための戦略マップは組織のビジョンを戦略として具体的なBSCにマップするためのパタン, 変換表のようなものです.
    - BSC戦略マップはその後のビジネス・モデリングの過程で, ゴールを表すものとして参照されます. それは別に問題ありませんね.
    - その参照によって, BSC戦略マップの内容はさまざまな形に変換され, その他のビジネス・モデル (例えばユースケース・モデル, 分析モデルなど) の中に分散して蓄積されていきます.
    - 同様にビジネス・モデルは参照され, エンタプライズ・アーキテクチャや要求モデル, システム・モデルの中に「溶けて」行きます.
    - さて, では最初 (の頃) に作ったBSC戦略マップは, 最終プロダクトであるソフトウェアの中のどこに存在するでしょう?
    - たいていの場合, BSC戦略マップの内容はソフトウェアの中に薄く溶けてしまっているので, 誰もソフトウェアのどの部分がBSC戦略マップのどの部分に由来しているのか明示することはできません.
    - 一つの方法として, 追跡性 (traceability) を確保することです. 追跡性をちゃんと維持すれば, あるソフトウェア・コンポネントがBSC戦略マップのどこに由来しているのかを指摘することができるかもしれません. ただし追跡性の確保は (特にビジネス・モデルとシステム・モデルのような「裂け目」が途中にある場合には) 大変ではあります.
  - では例えば, BSC戦略マップに掲げたあるゴールが達成されているかどうかはどのように知ることができるでしょうか?
    - 一番簡単なのは役員が部長に「XXXというデータをまとめてレポートせよ」と言うことです. でもこれが面倒な作業で, 度重なればムダ (余計なコスト) が発生します. 間違う可能性も高くなります.BSC戦略マップを実現したシステムがそこにあるにもかかわらず, です. つまり, 追跡性が例えあったとしても, システムの外部に (例えばドキュメントなどの形で) あったのではほとんど役に立たないと言うことです.
    - じゃあそれを機能として実現しようと言うのが今までのソフトウェア開発/ソフトウェア調達のやり方でした. そのときにBSC戦略マップからソフトウェア・コンポネントやデータ定義への追跡性があれば, (システムに外部にあったとしても) 機能の実現やソフトウェアの改修には役に立つかもしれません.
  - では例えば, 経営環境の変化などによってBSC戦略マップの見直しが必要になったとしましょう. ソフトウェアはどうすればいいでしょう. 多くの場合, ソフトウェアの機能追加/変更が行われます. また前と同じこと, つまりビジョン → ゴール → ビジネス・モデル → 要求モデル → システム・モデル → ソフトウェアのような道程をたどることになります. 「ソフトウェア工学的にちゃんと作られていれば」多少は楽になるかもしれませんがね.
    - ここでは例としてBSC戦略マップを取り上げましたが, 当然BSC戦略マップだけの問題ではありません. あらゆる問題の記述について言えることです.
  - つまり, 問題なのは問題の記述と解決の記述の間につながりがないか, もしくは非常に弱い, あるいは暗黙のつながりしかないことなのです. 最近はビジネス・モデリングを行うなど, 問題の記述へとソフトウェアの位置づけは拡張されているものの, それだけでは不十分ではないでしょうか.
  - 注: 実は問題の記述でもない, 解決の記述でもない作業がソフトウェア開発には必要です. 例えば方法の記述 (プロセス・モデリング) です. 方法の記述がなければ, ソフトウェア (自体) が進化することができないか, 非常に遅かったり難しかったりするでしょう. が, それについては今回は触れません.
- では次に, 問題の記述と解決の記述の間に不連続性がある理由について考えてみようと思います.
  - 今までのソフトウェアでは解決とは機能であり, 最終的に機能を記述するのはプログラムでした. プログラムは何らかのハードウェアと協調して, 「動作」します. つまり実行可能です.

    - 実は最終的な機能の記述であるプログラムに行き着くまでには多くのプログラム以外の記述を必要とすると信じられていますが, それらプログラム以外の記述 (XXX文書の類ですね) は通常, 実行可能ではありません.
    - もっとも最近では少しずつ, プログラミングとモデリングの緩やかな融合が進むと同時に, 実行可能でない記述が実行可能な記述に置き換えられつつあると思います.
        - 例えば, 形式仕様記述, 実行可能仕様, テスト駆動開発, 振る舞い駆動開発, モデル駆動開発, さまざまなシミュレータやジェネレータなどです.
- 一方, 問題の記述には多くの場合, いわゆるモデルが用いられています. これらのモデルは動きません. 実行可能ではないのです. 今までのソフトウェアの考え方では動くことは機能を実現するためであり, 問題にしろ解決にしろ, 記述するためではないからです. つまり機能とは関係ない, 問題を動かすってどういうこと? なのです.
    - 確かに静的な問題, つまりある時点でパラメタがすべて決まってしまい, 問題自身も変化しないような問題ならば動く必要はないかもしれません. しかし問題が動くならば, 問題の記述も動きたいではありませんか.
- 解決の記述は動くのに, 問題の記述は動かない. それがソフトウェアの問題と解決の間にある不連続性ではないか. 解決の記述が動くように問題の記述も動くべきである. 両方が動くならば, つながって動くべきである. 解決の記述や問題の記述が動く, ということはすでに「機能を実現する」ために動くわけではない. 動くことが第一で, その一部に「機能」があるのではないか.
    - と言うところまで, 話が進んできました. ここで問題をより明確に示すことができるようにいくつかの新しい用語などを導入して, 話を整理してみようと思います.
- ここで「様相」という用語を導入したいと思います. 様相とは, 「動く, 問題と解決の記述」のことです. 問題と解決の記述は「モデル」と言い直してもいいかもしれません. ただし単なるモデルではなく, 動くモデルです.
    - 「様相」という用語は建築家 原広司の1986年の論文「機能から様相へ」から取っています. この論文は今は「空間 <機能から様相へ>」2007, 岩波現代文庫で読むことができます. ただし, 原の様相はmodalityである (様相論理の様相ですね) のに対して, ここでの様相は英語で言うならばmodalityよりもcontext, texture, textileに近いと考えています. またそれ以外にも建築における様相とソフトウェアにおける様相を単純に対応づけられるかどうかはまだ分かりません. ここでは (とりあえずは) 単に言葉を借りてきただけと理解してください.
    - 様相は布のようなものです. 手で触るとある質感をもたらします. 布は糸を縦横に絡ませることによって織られています. 糸もまた繊維の絡み合いです. 布のパッチワークもまた布です. 布は柔らかく, 平らにすればある形をしているかもしれませんが, いくらでも変形が可能です. 「様相」が堅苦しければ「テクスチャ」と呼ぶのもいいと思います.
    - 様相/テクスチャとは「動く, 問題と解決の記述」のことですから実は, ITだけが様相ではありません. 「ビジネス」も一つの様相です. あるビジネス問題 (目標) とその解決のために「動く」ものだからです. 組織に属する「ひと」も一つの様相です. 現在のある組織 (あえて企業ではなく組織と言いますが) の様相とは, ビジネスの様相, ひとの様相, ITの様相が重なり合って (絡み合って, つなげられて) 作られています. したがってこの3枚の布がうまく重なって始めて組織という布ができあがると言うことになります.
    - ソフトウェアがこれからやらなければならないのは, このテクスチャを織ると言うこと, ビジネス・テクスチャとひとテクスチャをうまく写し取って (transcription), なおかつそこにITでなければあり得ないような何かを織り込んで, ITテクスチャを織り上げるということです. これを「テクスタイリング」 (本当はそういう言葉はないのですが) と言うのはどうだろうかと思っています.
- ITテクスチャにおいて, 「問題の記述が動く」と言うこと, つまりITテクスタイリングとは何かをさっきの例に戻って, もう少し具体的に考えてみたいと思います.
    - 解決の記述 (機能) の実現には通常プログラムを書くわけですが, 残念ながらいわゆる従来のプログラムには問題を記述する能力はありません. したがって問題の記述は, 従来の意味でのプログラミングではなく, モデリングによって行うと言っていいでしょう. ただし最初に述べたようにシステムのモデリングではなく, 要求, さらには対象領域 (ビジネスなど) のモデリングです. そこには例えばBSC戦略マップも含まれるでしょう.

- しかし, これだけでは単なるビジネス・モデルです. テクスチャであるためには, それが解決となめらかにつながって動かなければなりません. 問題を記述したモデルが動くとはどういうことでしょうか?
- BSC戦略マップを例に取れば, BSC戦略マップを単なるドキュメントや絵として記述するだけではなく, 次のようなことができなければならないと思います.
  - いつもシステムの中から状況に応じて見ることができる
  - 必要ならば編集することができる
  - 変更に伴ってどのような影響が及ぶかをシミュレートできる
  - 必要なデータを必要な形で得ることができる
  - ある条件を満たしているかどうか確認/検証することができる
- 言い換えると, BSC戦略マップのモデリング環境が実システムにおいても動いていて, ユーザがモデリングでき, そのモデルが実システムの他の部分と連動していることです. これは技術的にそれほど難しいことではありません. もっともそのためにはもっとも誰かが試行錯誤しながら, そういうアーキテクチャを実現する必要があるでしょうが, いったんアーキテクチャが確立してしまえば, 誰でも真似できるようなものだと思います.
  - 実際にモデリングとプログラミングが融合しているというか, それに近いような環境はいくつか存在します. 古くはSmalltalkなどをそのようなものとしてとらえている人々も居たのではないかと思います. 個人的には最近はGrailsというWebアプリケーション開発環境に注目していて, このようなITテクスタイリングのための環境をこの上に作れないかと実験をしているところです.
  - 今はBSC戦略マップを例に挙げましたが, これはそれ以外の問題の記述についても同じです.
  - 例えば普通にエンタプライズ系のソフトウェアを作るときに, 組織モデル, 扱っている商品のオントロジ, ビジネス・プロセス・モデル, 商習慣, 組織内の用語/隠語辞書などが必要になってきますよね. これらを暗黙的な知識や仕様書などのドキュメントのレベルに留めずに, 最終製品であるソフトウェアに組み込んで行く, ということです.
- これによって何が嬉しいのでしょう. 例えば以下のような点を挙げることができます.
  - いわば開発環境と一体化しているようなものですから, システムの寿命が非常に長くなる
  - 同じ理由から, システムの機能追加/変更が非常に簡単になる
  - 問題と解決の間の相互フィードバックが非常に緊密になり, タイムラグが非常に短くなる
  - 2年ごとのシステム更新などではなく, (Googleのサーバ側アプリケーションがそうであるように) 「常に」更新される
  - 問題が常に明確化される
  - 解決は常にテストされる
- ここでは機能はもちろん (ユーザの目から見て結果的に) 存在しますが, ソフトウェアの中心ではありません. ソフトウェアの中心は様相/テクスチャです. 機能はテクスチャから必要に応じて結果的に生み出されるものです. これは比喩ですが, 布を時には風呂敷として一升瓶を包むのに使ったり, 別の時には汗をぬぐうのに使ったり, あるいは切れた鼻緒をすげ替えるのに裂いて使ったりするのに似ています. これを私は (いささか挑発的に) 「機能はただである」と主張しています. 様相/テクスチャとそれを支えるアーキテクチャがちゃんと実現されていれば, 機能は必要に応じてただ同然に作り出すことができるからです.

- テクスタイリングは従来のソフトウェア受託開発, システム・インテグレーションなどとは異なるビジネス・スタイルです. ソフトウェアの機能を提供することでお客様からお金をいただくのではなく, 組織テクスチャという, 組織固有のコンテンツを編集するお手伝いをすることでお金をいただくわけです.
  - つまり, 例えば1年掛けてソフトウェアを開発して, その期間×人数に応じたお金をいただいて, 作ったものは納品して (今の日本の多くの事例ではその力関係上著作権も何もかも差し上げて) おしまい, ではありません.
    - いや, 実際には「保守」名目で毎年何パーセントかのお鳥目をいただいていると思いますが:-)
  - コンテンツ (組織のテクスチャ) は常に変わり続けますから, それを入力して, 編集して, 利用するお手伝いを (できれば) その組織が続く限りして, その組織の価値向上に寄与した分のいくらかをちょうだいするというビジネス・スタイルです.

        - つまりモデリング環境をユーザに提供してしまうわけです. ユーザがモデリングできなければ, 我々がそれをお手伝いすることができます.
        - リアルタイムにモデルとビジネスが同期し続けることになります.
            - むしろモデルを変えることでビジネスが変わる仕組みを提供すると言ってもいいかもしれません.
    - そしてテクスタイリングは, コンサルティングとも異なっています. コンサルティングには動くテクスチャという確実な何かがありません. もちろん優秀なコンサルタントは居てうまくいくこともあるでしょう. テクスタイリングはITテクスチャをビジネス・テクスチャとひとテクスチャに滑らかにつなげることであって, ビジネス・テクスチャやひとテクスチャを織るのは組織なのです.
        - もちろん, そこにさらにコンサルティングの存在する余地はあると思いますが.
- 様相という側面からは, ソフトウェアは「実行可能な知識」であると考えることができます.
    - 知識にはさまざまな属性, 質 (知質) があります.
        - 伝達可能性
        - 証明可能性
        - 反駁可能性
        - 変換可能性
        - 応用可能性
        - 追跡可能性
        - モジュール性
        - 自己完結性
        - 汎用性
        - 実行可能性
        - .....
    - Smalltalkは知識ブラウザ/シミュレータと考えることもできます.
    - 今の時点で成功しているとは言い難いですが, オントロジはその方向の一つの例と言えるかも知れません.
        - オントロジは「こと知」より「もの知」に偏っていますが.
    - 我々の仕事は「機能」を実現することから, 顧客の「知識」をコンテンツ化し, 実行可能にすること (= 様相, 実行可能なWikipedia?) になりつつあるのではないでしょうか.

ソフトウェアの機能から様相へ

masaki@metabolics.co.jp

* あなたの作ったモデルは何を対象にしていますか?

* あなたの作ったモデルは最終製品のどこに存在していますか?

3

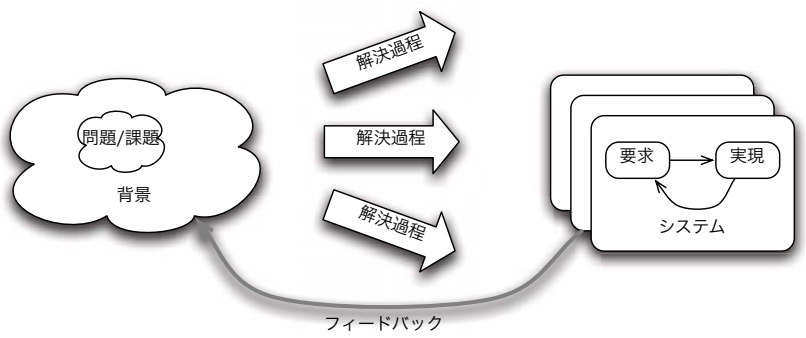
* 初期開発から (例えば) 2年経ち, 改修が必要になったとき, どのようにモデリングを行いますか?

4

* UMLモデリングは本質的に機能のモデリングである

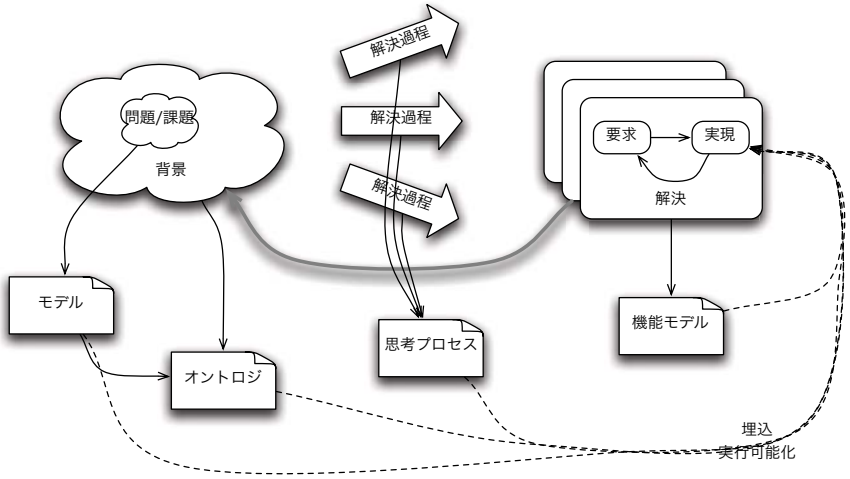
5

問題解決過程



6

問題の埋込/実行可能化

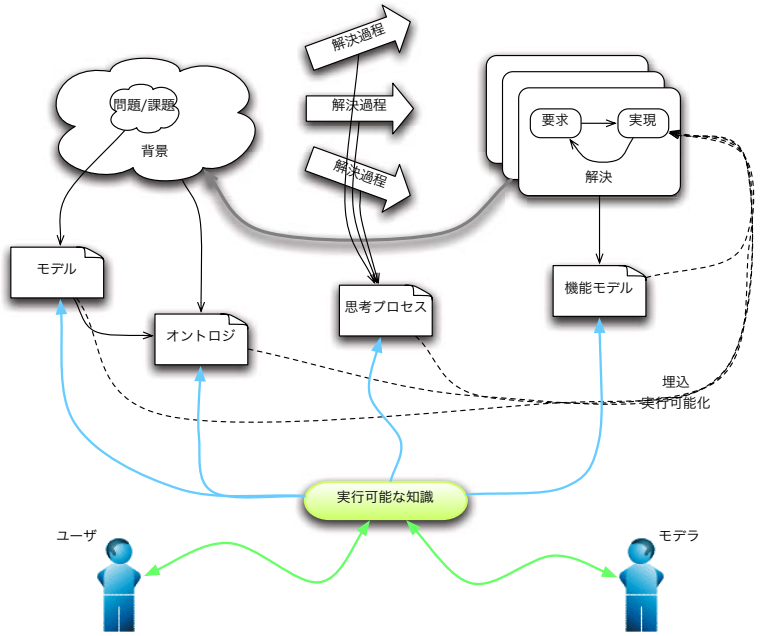


7

様相

8

実行可能な知識



9

* テクスタイリングは様相を紡ぎ続けることである

10

* 認知パターン - オブジェクト技術のための問題解決フレームワーク, C. ガードナーほか, 2001, ピアソン・エデュケーション
* 企業情報システムの一般モデル - UMLによるビジネス分析と情報システムの設計, C. マーシャル, 2001, ピアソン・エデュケーション
* Organizing Business Knowledge, T. W. Malone, et. al., ed., 2003, MIT Press
* 物のかたちをした知識 - 実験機器の哲学, D. ベアード, 2005, 青土社
* ビジネスパターンによるモデル駆動設計, P. Hrubyほか, 2007, 日経BP
* 空間 <機能から様相へ>, 原広司, 2007, 岩波現代文庫, 岩波書店
* アルゴリズム的思考とは何か, 松川昌平, 2007, 10+1, No.48, pp.155, INAX出版