衛星データ処理勉強会（第15回）

Railsの紹介とSIB2入力インタフェースプロトタイプ開発

宇宙航空研究開発機構　宇宙科学研究本部
システム開発部　探査機システム開発グループ
馬場　肇

概要

- Ruby on Rails (RoRまたは単にRailsと称される)は、Webアプリケーション開発のためのWebフレームワークの一つである。
- Railsの特徴として、DRYやCoCという基本思想、Scaffoldやコードスケルトン生成による省力化、ActiveRecordによる強力なデータモデリング手法、随所に用いられるDSL 風味な言語、Ajaxへの標準対応、これらによって実現されるメタプログラミングとアジャイルソフトウェア開発、などが挙げられる。これらは、科学本部で行なわれるインハウス型ソフトウェア開発の多くに親和的であって有効だと考えている。
- その実証実験として、衛星運用室で開発中の次世代衛星情報ベース(SIB2)の入力インタフェースをWebアプリケーションとしてプロトタイピングするソフトウェア開発プロジェクトを提案し、現在は立ち上げ段階である。
- 今回は、上述のようなRailsの概要とその特徴、そしてStrutsやTsunagi等のフレームワークとの比較を紹介する。加えて、計画中の開発プロジェクトの概要と今年度の獲得目標を紹介し、アジャイル開発がもたらす新しい可能性について議論する。

目次

- 前半部:
  - Railsの紹介
  - アジャイルソフトウェア開発の紹介
- 後半部:
  - SIB2入力I/Fプロトタイプの開発

前半部: Railsの紹介

Railsとは

- MVCフルスタックなWebアプリケーション開発フレームワーク
  - 37 Signals（という会社）の DHH（人名）によって開発された
- オブジェクト指向スクリプト言語のRubyを用いている

Rails の特長

Dave “達人プログラマー” Thomas による Rails が好きな理由ベスト 10（の超訳）

1. Web 開発をアジャイルにする
2. ちょっと小粋なエフェクトつきの Web ページが作れる
3. 「フレームワークのお守り」ではなく、アプリの作成に集中できる
4. 規模が大きくなってもメンテナンス可能でありつづける
5. 顧客に対して「できます」という答えがもっと言えるようになる
6. テスティングが組み込まれていて簡単に使える
7. 即時のフィードバック: コードを編集して、再読み込みすると、その変更がブラウザで確認できる
8. メタプログラミングのおかげで高いレベルでプログラミングできる
9. コード生成のおかげで手早く始められる
10. XML 不要!

- http://ll.jus.or.jp/2005/files/lldn2005-rails.pdf

Railsのキーワード

- DRY (Don't Repeat Yourself)： 重複は悪、同じことを違うところで書かない
- Convention Over Configuration： 細かく設定するより適切な規約で無設定に
- Agile： アジャイル開発しやすいように全てが設計
- DSL (Domain Specific Language)： ドメイン特化言語風味の構文
- Active Record： シンプルで便利な O/R マッピング
- (Functional/Unit) Testing Framework： テストがかなり書きやすくなる
- Code Generation ： 何も書かなくてもコードの雛型を生成
- Ajax： prototype.js を始めとして Ajax には妙に熱心
- Rake (Ruby Make)：これまた DSL 風味な Makefile が Ruby で書ける
- Rubyish：「The Ruby Way」「Ruby らしさ」にわりとこだわる
- fun!：「たのしさ」を重視
  - http://ll.jus.or.jp/2005/files/lldn2005-rails.pdf

RailsのMVCモデル

- モデル: ActiveRecord(専用の O/R マッパーライブラリ)
- コントローラ: ActionController(アクションをクラス単位ではなくメソッド単位で管理)
- ビュー: ActionView+ERb(ヘルパーライブラリをつけた埋め込み Ruby)

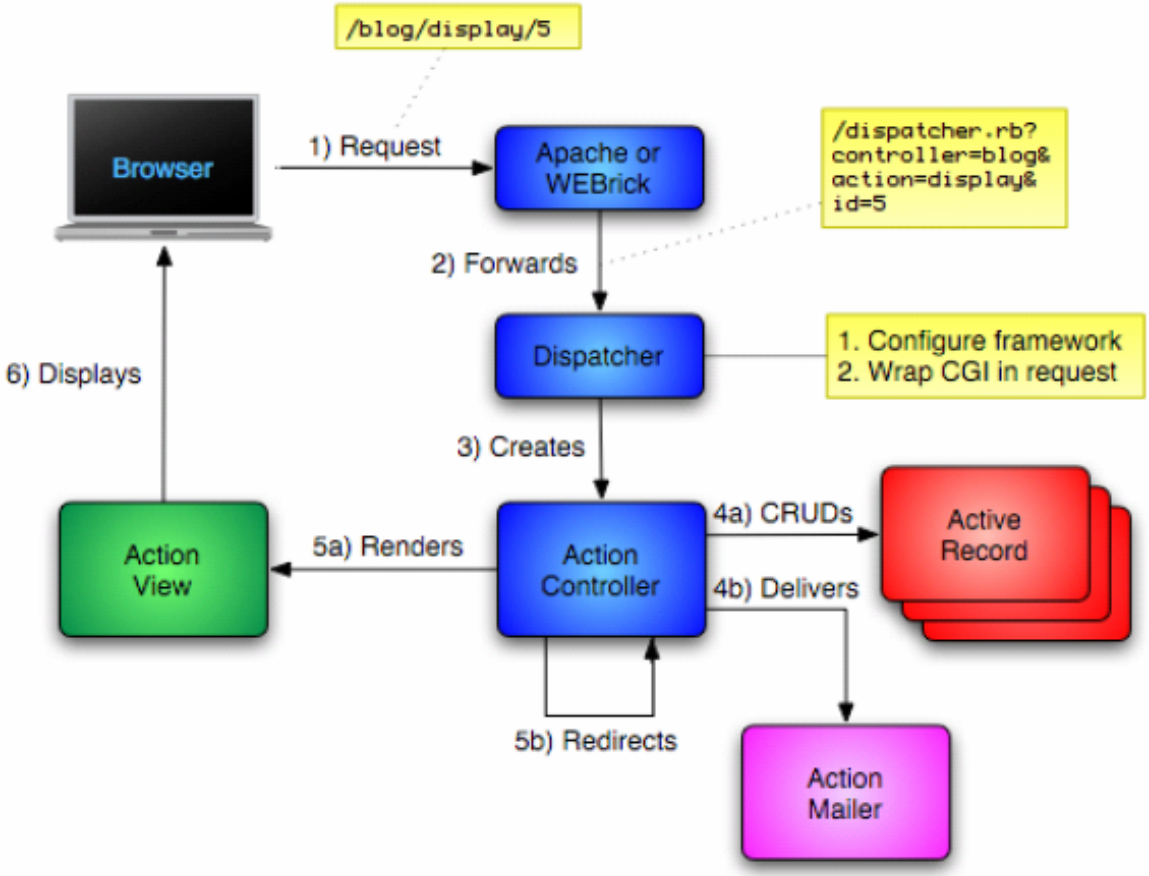


図 1:Rails のリクエスト処理の流れ

- http://ll.jus.or.jp/2005/files/lldn2005-rails.pdf

MVCフルスタック？

- Model – View – Controller の全てを備えているという意味
- Struts/Tsunagi は、DAO（Data Access Object）を使ってはいるが、Model表現のためのORM（Object-Relation Mapping tools）は使っていない
  - Strutsなら、Hibernate や TorqueといったORMを使うらしい

Strutsとの比較

	Rails	Struts
MVC準拠	準拠	準拠
ORMのハンドル	ActiveRecord	×
Ajaxへの対応	対応	未対応
設定ファイルの記述量	少ない	多い
規約の柔軟性・汎用性	カスタマイズで可能	設定の記述で対応
コードの記述量	Scaffoldで削減可能	やや冗長なコーディングも多い
統合開発環境	あり	あり
実績	リリース間もない	豊富な実績
日本語情報	少ない	豊富

- http://www.atmarkit.co.jp/fjava/special/javvsaror/ror02.html

Railsの思想: CoC と DRY

- CoC: Convention over Configuration
  - 「設定より規約」=積極的にデフォルト値を利用しよう
  - Railsが推奨する各種の命名規則に従う事で記述量が大幅に削減できる
- DRY: Don't Repeat Yourself
  - 「繰り返しは避けよ」=重複する部分は一か所にまとめよう

- どちらも記述するコード量を削る事が可能→余計なコードが減る→残ったコードがそれだけ本質的→そこでやっている処理が明確に
- 可読性も上がる→バグの介入も減る

ActiveRecord による強力なデータモデリング

- ORM(Object-Relation マッピングツール)
- オブジェクト指向言語におけるオブジェクトと、リレーショナルデータベースにおけるレコードとを対照させること

- ドメインロジックをモデルに持たせる特徴もある（検証機能）

その他の特長

- Scaffold やコードスケルトン生成による省力化
  - ほとんど変更する事なくスケルトンが動作する
  - より本質的な変更に集中できる
- DSL風味な文法
  - Ruby の文法にも大きく依存
- Ajaxに標準対応
  - Prototype.js, Script.aculo.us, RJS

Railsフレームワーク

- ActiveSupport
- ActiveRecord　RoRにおいてデータモデルを担うO/Rマッピングフレームワーク
- ActionPack　　MVCにおける、ビューとコントローラを担当
  - ActionController
  - ActionView
- ActionMailer
- ActionWebService
- Railities

サンプルアプリの作成

- 初期設定
- テーブル・足場を作成、CRUDアクセス
- コントローラとビューの定義
- Hello World!の作成
- テーブルカラムの追加
- Validation: 検証機能(モデル)
- Association: モデル間の関連
- コンソールアプリケーション
- 任意のビューを作成（AR.with_scope）

初期設定

```
# アプリケーションを新規作成
% rails sample

# データベースを設定して作成
% cd sample
% vi config/database.yml
% mysqladmin -uroot -p create sample_development
```

Railsアプリのディレクトリ構成

実際にコードを書く部分

<root directory>	
app/	アプリケーション用のコードを置く
|controllers	コントローラ用の.rb ファイルを置く　※自動 URL マッピング
|helpers	ビューヘルパー用の.rb ファイルを置く
|models	モデル用の.rb ファイルを置く
|views	ビュー用のテンプレート.rhtml を置く　※アクション名
components/	共通コンポーネントを置く
config/	Rails 環境用設定ファイルを置く
db/	データベーススキーマ、マイグレーションファイル用
doc/	ドキュメントを置く、また自動生成時の出力先
lib/	アプ当該リケーション用のライブラリを置く　※load path に含まれる
log/	ログ出力先
public/	Web 上で公開するファイルを置く　※Web サーバが利用できる
script/	サーバ起動スクリプト、各種自動生成などの支援スクリプトを置く
test/	ユニットテスト、機能テスト、fixture、テスト自動生成時の出力先
tmp/	pid, cache, session, socket などのテンポラリ情報が置かれる
vendor/	サードパーティライブラリ、プラグインなどを置く　※load path に含まれる

テーブル・足場を作成、CRUDアクセス

```
# メンバー(member)
#   属性: 名前(name)、肩書(title)、メール(email)
% ./script/generate model member
% vi db/migrate/001_create_members.rb
  t.column :name, :string
  t.column :title, :string
  t.column :email, :string
# データベーステーブルを作成
% rake db:migrate

# CRUD用の足場コードを作成
% ./script/generate scaffold member member

# サーバを起動してCRUDアクセス
% ./script/server
% firefox http://localhost:3000/member/
```

コントローラとビューの定義

```
% less app/controller/member_controller.rb
class MemberController < ApplicationController
  ...
  def show
    @member = Member.find(params[:id])
  end
  ...

% less app/view/member/show.rhtml
<% for column in Member.content_columns %>
<p>
  <b><%= column.human_name %>:</b> <%=h
   @member.send(column.name)
%>
</p>
<% end %>
<%= link_to 'Edit', :action => 'edit', :id => @member %> |
<%= link_to 'Back', :action => 'list' %>
```

Hello World!の作成

```
# コントローラの定義
% vi app/controller/member_controller.rb
  def hello
    @message = "Hello World!"
  end

# ビューの定義
% vi app/view/member/hello.rhtml
<pre>
  <%= @message %>
  現在時刻 <%= Time.now %>
  10分後 <%= 10.minutes.from_now %>
</pre>

# ブラウザでアクセス
% firefox http://localhost:3000/member/hello
```

テーブルカラムの追加

```
# 内線(telno)カラムを追加してみる
% ./script/generate migration add_telno_to_members
% vi db/migrate/002_add_telno_to_members.rb
  add_column :members, :telno, :integer

# データベーステーブルを作成
% rake db:migrate

# CRUD用の足場コードを再作成
% ./script/generate scaffold member member
```

Validation: 検証機能(モデル)

```
% vi app/models/member.rb
  # 名前が空のメンバーが追加されないようにする
  validates_presence_of :name
  # 同じメールアドレスのメンバーが追加されないようにする
  validates_uniqueness_of :email
  # メールアドレスのフォーマットをチェックする
  validates_format_of :email, :with => /¥w+@¥w+/
```

Association: モデル間の関連

```
## 組織(organization) 属性: 名前(name)
% ./script/generate model organization
% vi db/migrate/003_create_organizations.rb
  t.column :name, :string
# membersテーブルに organization_id カラムを追加
% ./script/generate migration add_organization_id_to_members
% vi db/migrate/004_add_organization_id_to_members.rb
  add_column :members, :organization_id, :integer
% rake db:migrate
# CRUD用の足場コードを作成
% ./script/generate scaffold organization organization

## モデル間の関係を記述
# メンバーは一つの組織にしか所属できない
% vi app/models/member.rb
  belongs_to :organization
# 組織にはメンバーが複数人所属している
% vi app/models/organization.rb
  has_many :members
```

コンソールアプリケーション（1）

```
% ./script/console
>> require 'pp'

# 件数と、全数検索
>> Member.count
>> pp Member.find(:all)

# データベース演算: 選択(selection)
>> pp Member.find(:all, :conditions => {:name => '馬
   場'})
>> pp Member.find_by_name('馬場')

# データベース演算: 射影(projection)
>> Member.find(:all, :select => "name,email")
>> Member.find(:all).map(&:name)
```

コンソールアプリケーション（2）

```
# メンバーを追加
>> yamamoto = Member.new
>> yamamoto.name = "山本"
>> yamamoto.title = "助教"
>> yamamoto.email = "yamamoto@localhost"
>> yamamoto.save

# メンバーを追加(yet another method)
>> murakami = Member.create(:name=>"村上
   ", :title=>"PD", :email=>"murakami@localhost")

# メンバーを追加(検証機能に引っかかって失敗)
>> nguser = Member.create(:email=>'abc')
>> nguser.save  #=> false
>> pp nguser.errors
```

コンソールアプリケーション（3）

```
# PLAINのメンバーを追加
>> plain = Organization.find_by_name("PLAINセンター")
>> ebisawa=Member.find_by_name('海老沢')
>> plain.members << ebisawa
>> plain.members << matuzaki
>> plain.members << yamamoto
>> plain.members.count

# XML表現
>> plain.members.to_xml

# データベース演算: 結合(JOIN)
>> Organization.find_by_name("PLAINセンター", :include
   => :members)
```

ビューを自由に定義（AR.with_scope）

```
# 登録メンバー全員の数
>> Member.count
# PLAINセンターに所属しているメンバー
>> Member.with_scope(:find => {:conditions
  => "organization_id = 1"}) do
>>   Member.count
>>   Member.find(:all)
>> end

## SQL の CREATE VIEW と違う点
1. ビューなのに書き込みできる（RDBMSによっては不可）
2. RDBMSに依存しない
3. 動的な条件によってビューを動的に作成できる
```

プロトタイピングの定義

- ここで、「プロトタイピング」という活動の定義を明確にしておきましょう。ソフトウェア開発に限らず、一般的なプロトタイピングには以下のような目的があるでしょう。
  - ユーザー要求を具現化し、フィードバックによってさらに明確化する
  - プロトタイプの設計結果から問題点を洗い出し、本番開発時の設計品質を
  - 高める
  - ユーザー要求や、机上で行った設計の実現可能性を検証する
- 本連載ではプロトタイピングを、「本番開発の前に、これらの目的を満たす試作品を創造する活動」と定義します。

- http://www.atmarkit.co.jp/farc/rensai2/proto01/proto01b.html より

- 　アジャイルプロトタイピングでは、顧客からのフィードバックを頻繁に受けるために、詳細な内容のデモを何度も行い、プロトタイプに対する試行錯誤的な変更を繰り返します。この際、プロトタイプに対して次にどのような変更を行うかは、顧客や開発者が持つ知識や経験を基に発想され、決定されます。つまり仕様策定や、設計に関する知見を獲得するためのアジャイルプロトタイピングは、本質的にクリエイティブな活動なのです。そして、顧客と開発者のクリエイティビティをどこまで引き出せるかが、アジャイルプロトタイピングを成功させるための鍵となります。

- http://www.atmarkit.co.jp/farc/rensai2/proto03/proto03a.html

- アジャイルプロトタイピングの最終成果物は、「過程から得られる知見」です。中間生産物として動くプロトタイプというものが存在します。このプロトタイプこそが顧客と開発する側である私たちが共有するシステムのイメージに当たります。アジャイルプロトタイピングとは、具現化されたシステムのイメージを基に「本当はどうしたいの？」を探す顧客との協調作業なのです。
- プロトタイプはプロトタイプ
- 繰り返しになってしまいますが、アジャイルプロトタイピングの最終成果物は「知見」です。中間生産物であるプロトタイプの完成度に必要以上にこだわることは時間のムダと考えます。どうしても「ここがオブジェクト指向っぽくない」などと細部に目が行ってしまいがちですが、最低でも以下の項目が喚起できればプロトタイプの役割としては十分だと考えられます。
  - 顧客の要件、要望
  - その実現可能性
  - データモデル
  - 画面構成
  - モジュール構成
  - 工数

- http://www.atmarkit.co.jp/im/carc/serial/proto05/proto05.html

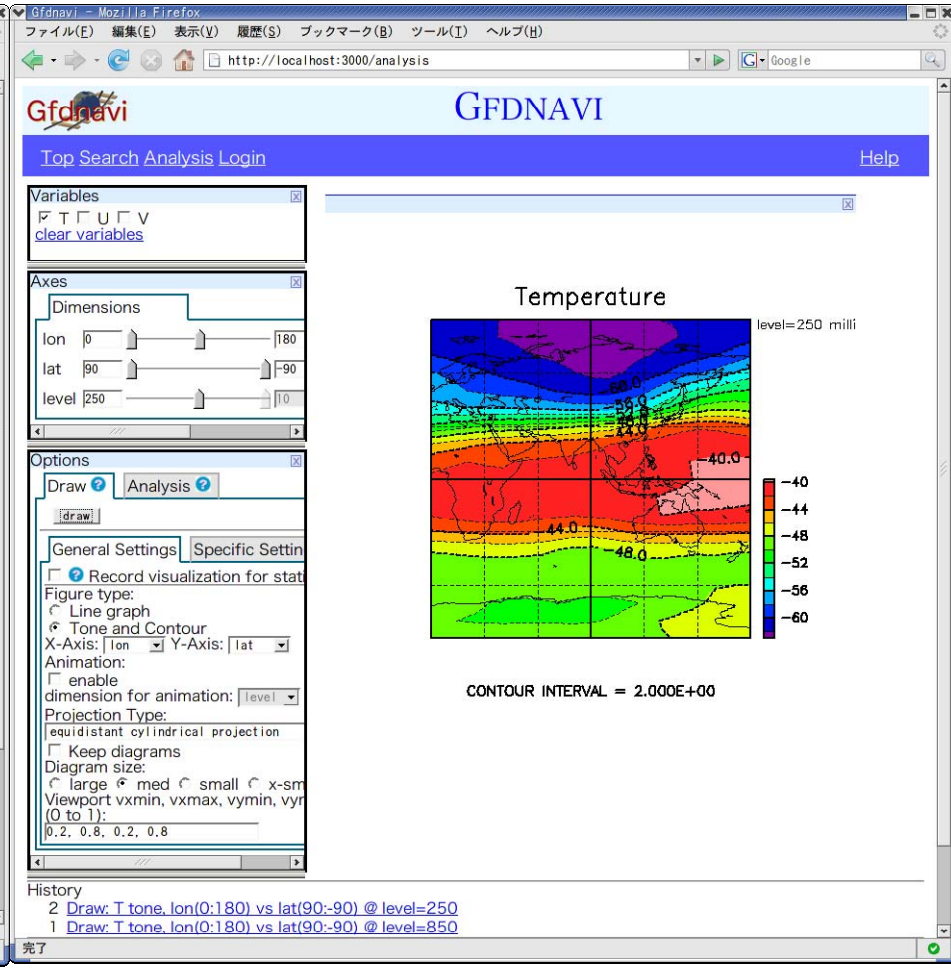
アジャイルソフトウェア開発

- 最終的に得られるものは「知見」
  - 正式版の開発（仕様書・設計書など）にフィードバックする事が目標
    - 最終プロダクトの作成はメーカーに出して問題ない
  - いたずらにUIなどに凝って時間を取られない事が大事

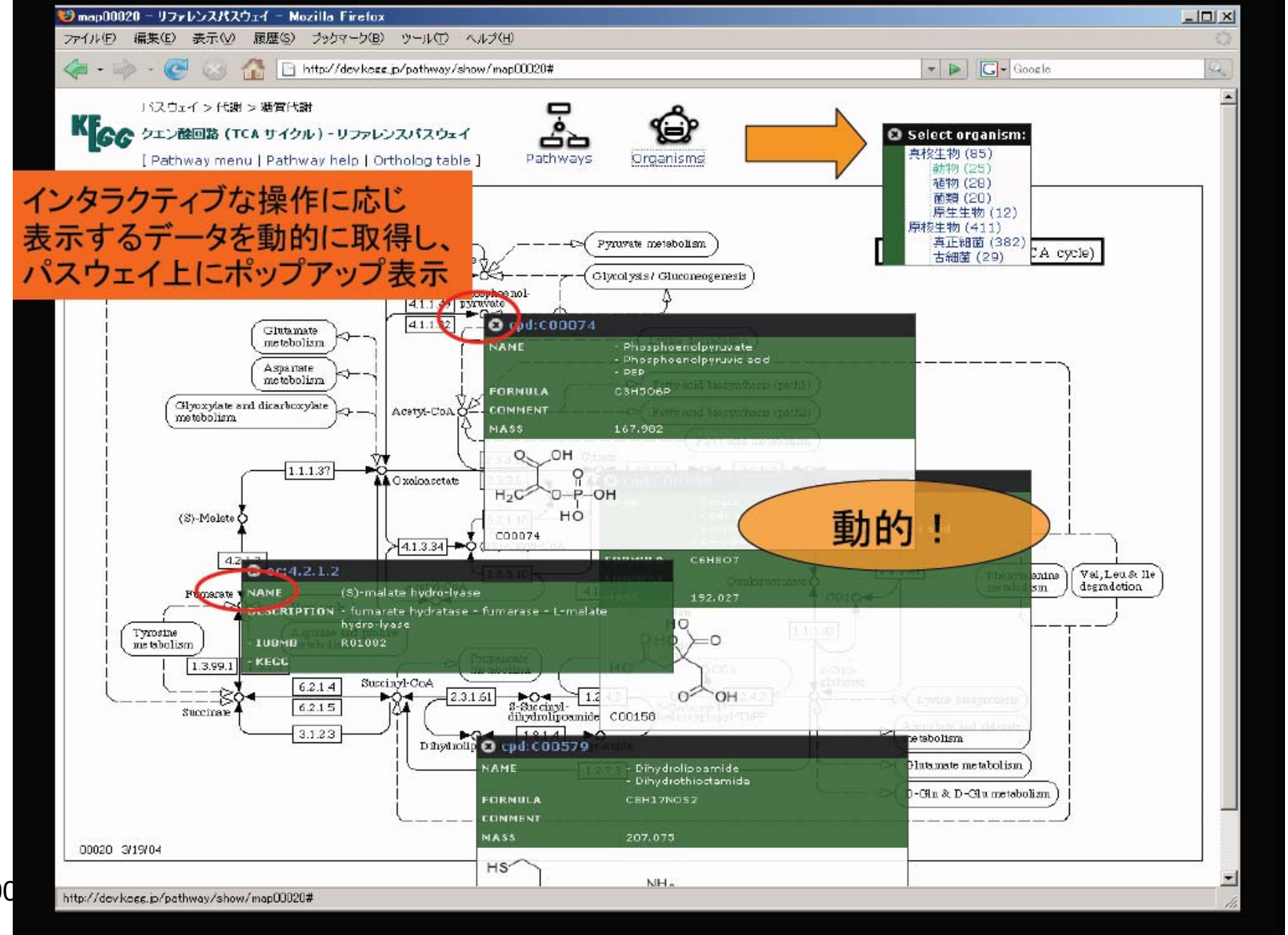
自然科学の研究分野とアジャイル開発

- Railsを使ったアジャイル開発手法は、科学本部の開発スタイルに合うのでは？
  - あいまいな仕様、すぐに変わる仕様
  - どういうUIがいいのか、だれもさっぱりわからない（先例がある訳でない）
  - 本質的に複雑なモデルを扱う（SIB2とか）
  - 極めて少ないリソース（人的・金銭的）
- 地球科学や生命科学分野でもRailsは使われている
  - Gfdnavi: 地球科学（地球流体データベース）
    - GFD電脳倶楽部（京大とか）
    - http://www.gfd-dennou.org/arch/davis/gfdnavi/
  - KEGG: バイオインフォマティクス（ゲノムデータベース）
    - オープンバイオ研究会(片山君: shackon)
    - http://open-bio.jp/?meeting6-hands-on-KEGG-on-Rails

サンプルその1: Gfdnavi（地球物理）



サンプルその2：KEGG（生命科学）



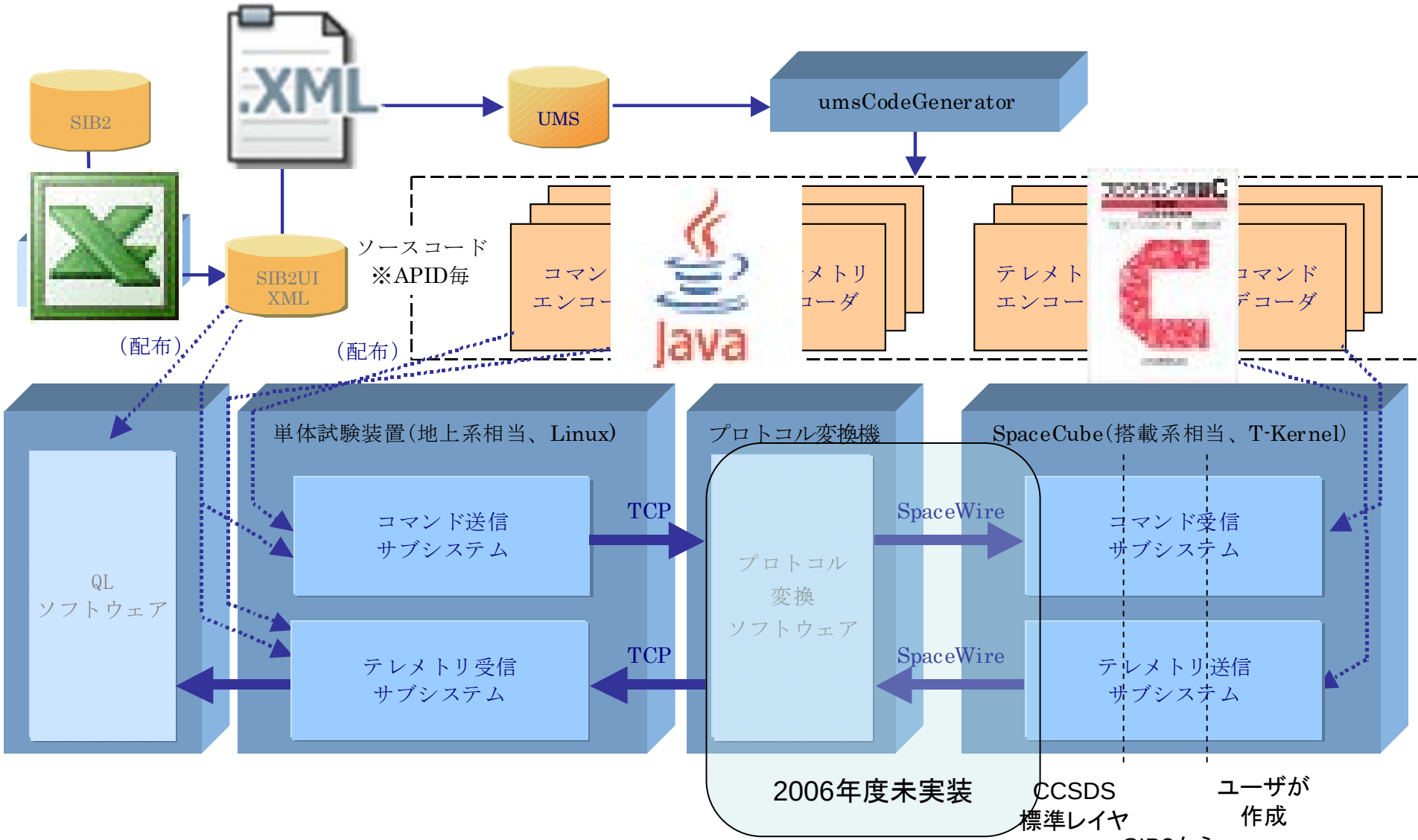
後半部: SIB2入力I/Fプロトタイプ開発

- SIB2/GSTOSとは
- SIB1Railsプロトタイプによって得られたもの

SIB2/GSTOSとは

- 衛星運用室では、次世代衛星情報ベース(SIB2)のモデル化と、汎用衛星試験運用ソフトウェア(GSTOS)の開発を進めている。
  - SIB2: 衛星情報ベース（Satellite Information Base 2）
  - GSTOS: 汎用衛星試験運用ソフトウェア（Generic Satellite Test and Operation Software）
    - 昨年度は、UMSコードジェネレータベースのものを開発
  - SIB2UI: SIB2定義をするためのユーザインタフェース
    - 昨年度に、Excel + VBマクロベースの入力I/Fを開発
    - SIB1の定義方法（入力用SIB）も Excelベース
- SIB2モデル化の基本的な概念設計は終了しており、レビュー段階にある。

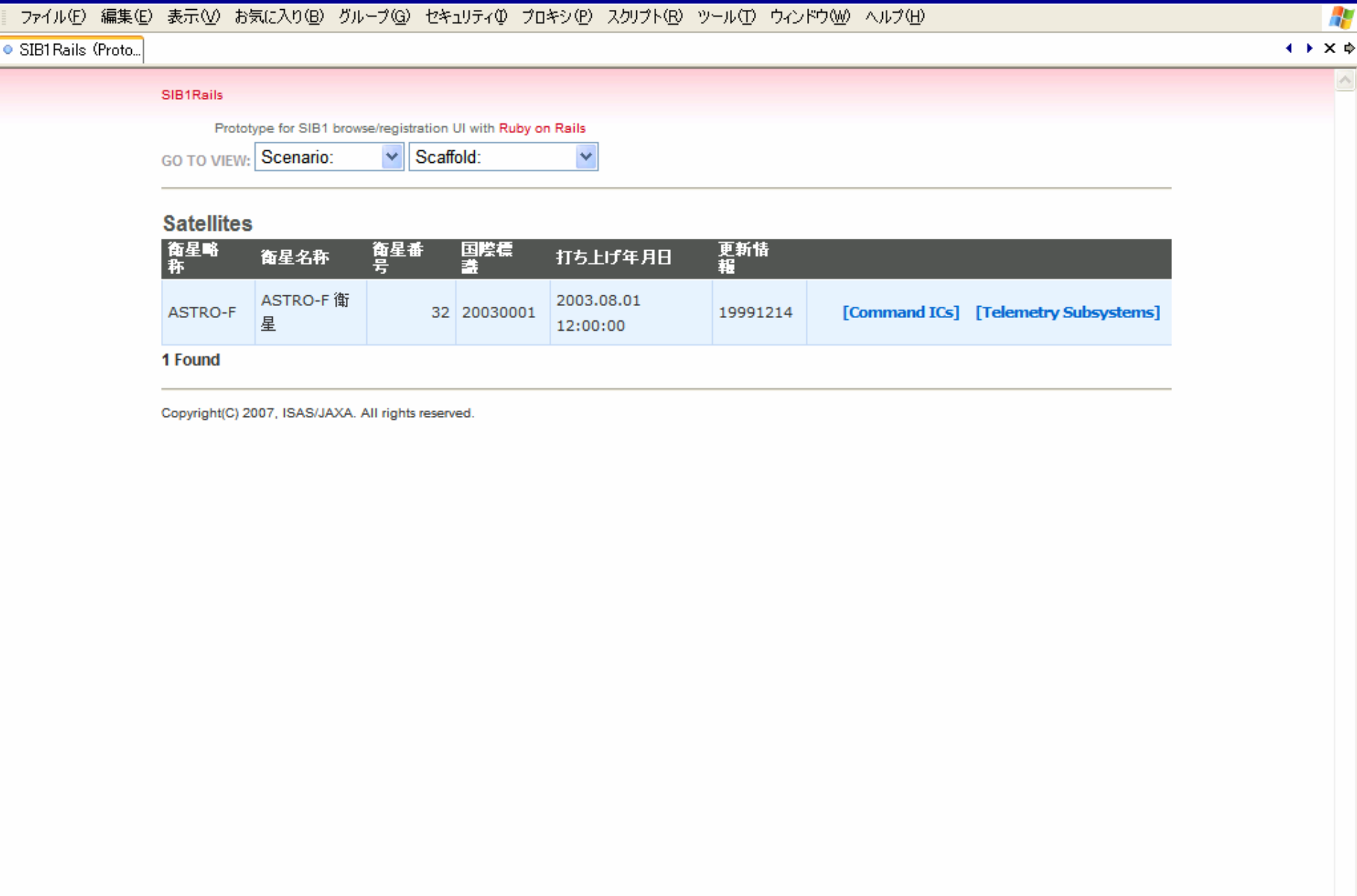
2006年度のSIB2UI+コード生成部の構成



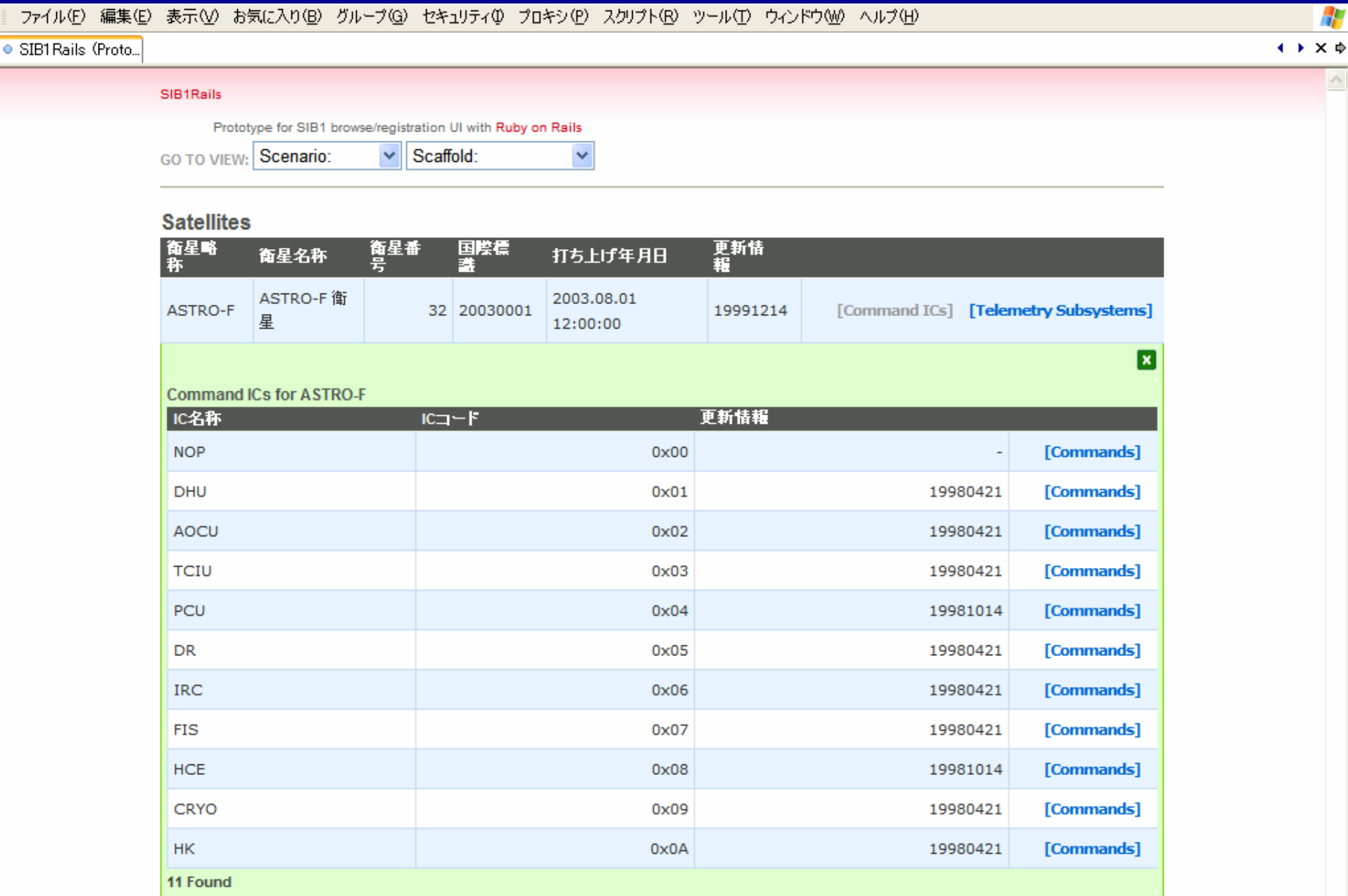
Webアプリプロトタイプ: SIB1Rails

- SIB2UI が Excel ベース?
  - ビューの切り方の問題で果てしない議論になった
  - それくらいなら、いっそ Webアプリ化してしまう方が良いのでは？というのが最初の動機
- 最初のプロトタイプ: SIB1Rails
  - まずは、SIB1で Rails化の試験をしてみた。
  - モデル化の源泉情報は出力用SIBの定義情報を独自に解析してみたもの
    - →OracleでのSIB定義とはかなり違っているはず
  - ユーザインタフェース部分は、ActiveScaffoldプラグインを利用して手抜き（その分、モデル定義に集中）
  - UIはUIで別途考える必要がある（未着手）

SIB1Railsデモ(1)衛星一覧



SIB1Railsデモ（2）コマンドIC一覧



SIB1Railsデモ（3）DHUコマンド一覧


ファイル(F) 編集(E) 表示(V) お気に入り(B) グループ(G) セキュリティ(I) プロキシ(P) スクリプト(R) ツール(T) ウィンドウ(W) ヘルプ(H)

SIB1 Rails (Proto...

Satellites

衛星略称	衛星名称	衛星番号	国際標識	打ち上げ年月日	更新情報	
ASTRO-F	ASTRO-F 衛星	32	20030001	2003.08.01 12:00:00	19991214	[Command ICs] [T

Command ICs for ASTRO-F

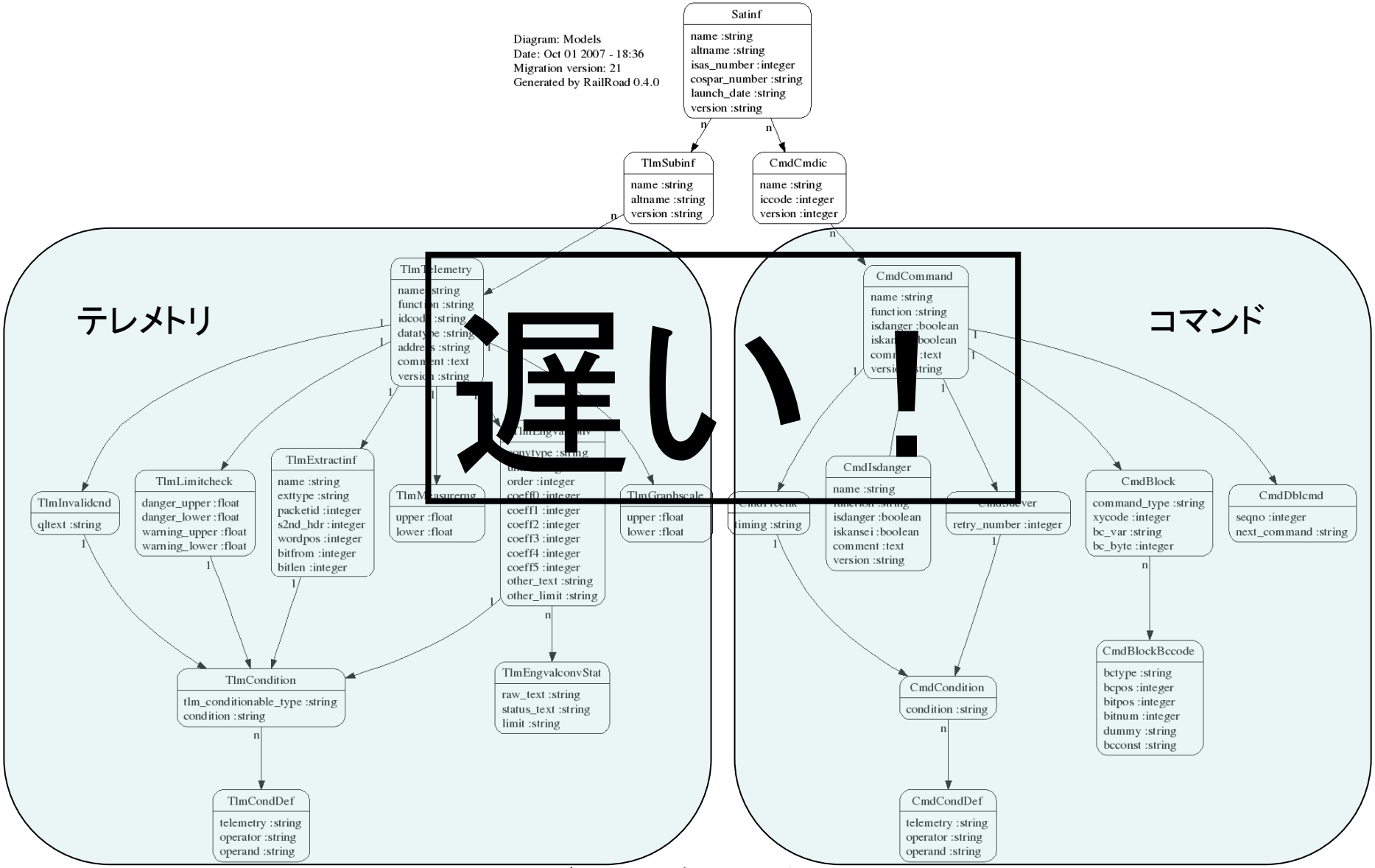
IC名称	ICコード	更新情報
NOP	0x00	
DHU	0x01	199804

Commands for DHU

コマンド名称	ファンクション	危険コマンド識別	衛星管制識別	コマンドコード指定	ダブルコマンド指定	送信前チェック用情報	サクセスベリファイ用情報
DHU_A-BUS_GD	A-BUS_GUARD	false	false	DC|0x08	-	-	-
DHU_B-BUS_GD	B-BUS_GUARD	false	false	DC|0x09	-	-	-
DHU_A-BUS_GD_CNL	A-BUS_GUARD_CANCEL	false	false	DC|0x0A	-	-	-
DHU_B-BUS_GD_CNL	B-BUS_GUARD_CANCEL	false	false	DC|0x0B	-	-	-
DHU_BUS_ACC_ENA	BUS_ACCESS_ENABLE	false	false	DC|0x20	-	-	CHK|4
DHU_BUS_ACC_DIS	BUS_ACCESS_DISABLE	false	false	DC|0x21	-	-	CHK|4
DHU_TLM_ROM_A_ON	TLM-ROM_A_ON	false	false	DC|0x25	-	-	-
DHU_TLM_ROM_B_ON	TLM-ROM_B_ON	false	false	DC|0x26	-	-	-
DHU_TLM_ROM_SW_ENA	TLM_ROM_SWITCH_ENABLE	false	false	DC|0x27	-	-	CHK|4
DHU_TLM_ROM_SW_DIS	TLM_ROM_SWITCH_DISABLE	false	false	DC|0x28	-	-	CHK|4
DHU_REP_STOP	REPRODUCE_OFF	false	false	DC|0x29	-	-	CHK|4

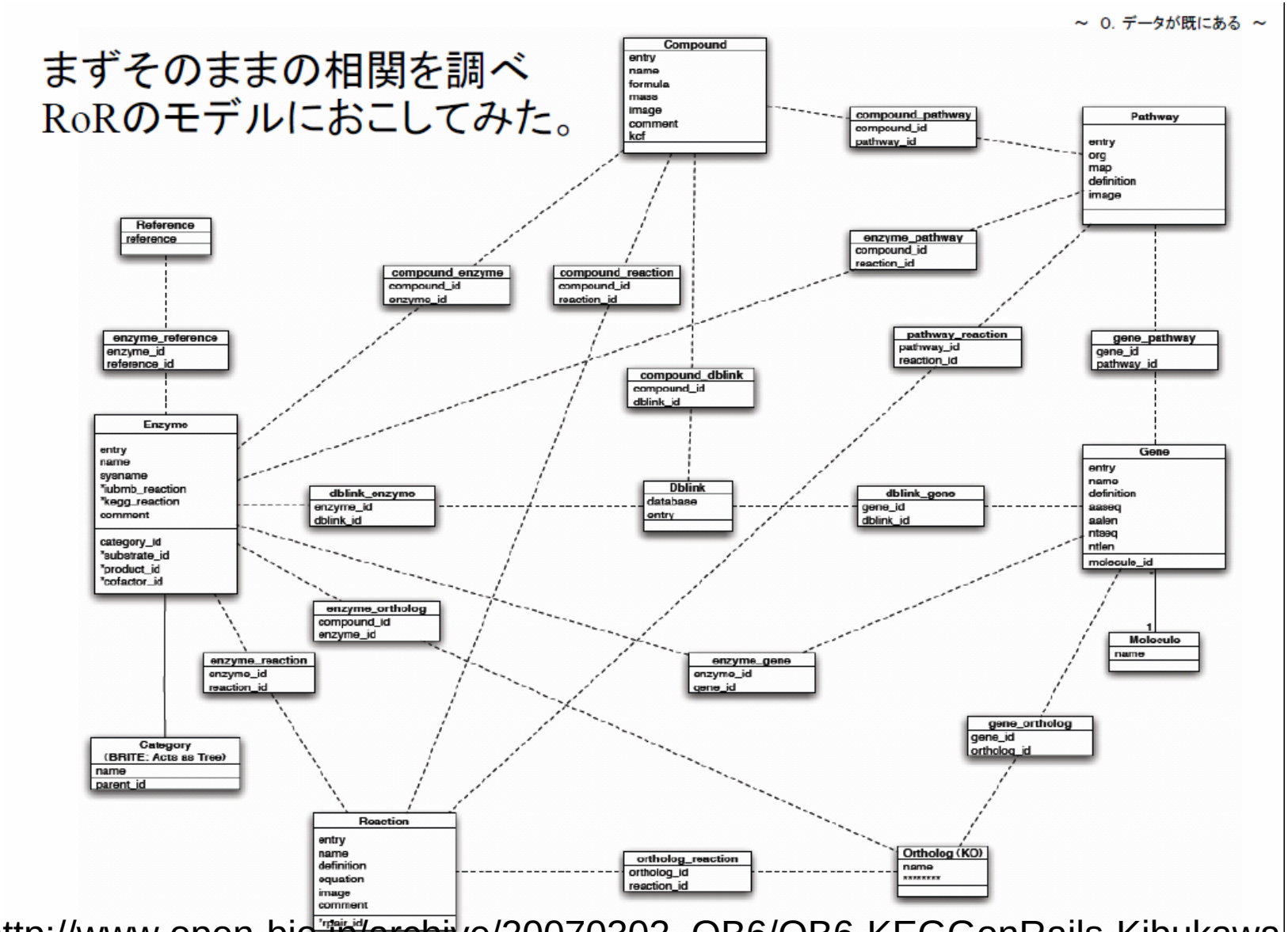
SIB1Railsデモ（4）詳細（サクセスベリファイ）

クラス図詳細（app/models/*.rbから生成）



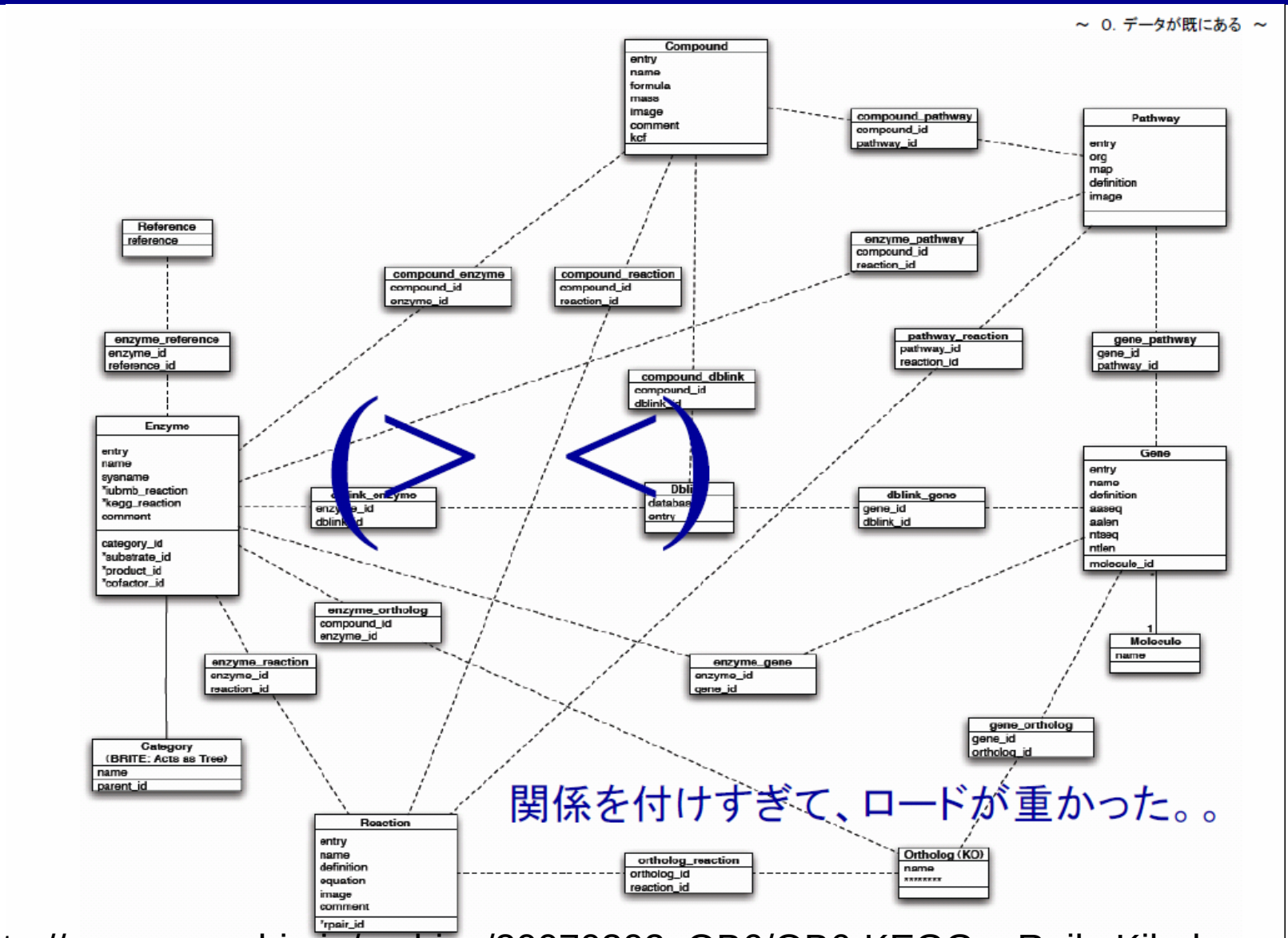
KEGGの例（1）

まずそのままの相関を調べ
RoRのモデルにおこしてみた。



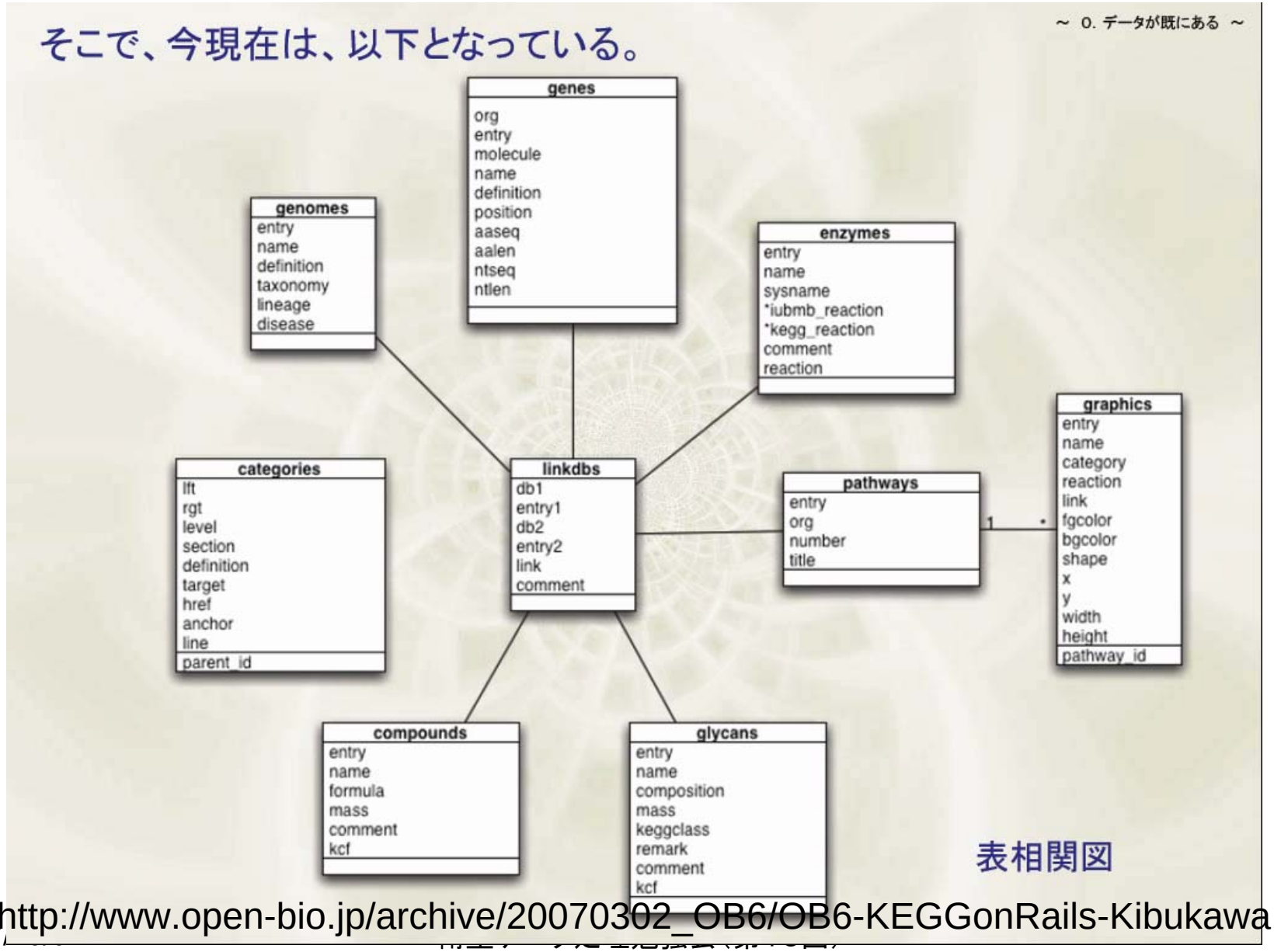
http://www.open-bio.jp/archive/20070302_OB6/OB6-KEGGonRails-Kibukawa.pdf

KEGGの例（2）



http://www.open-bio.jp/archive/20070302_OB6/OB6-KEGGonRails-Kibukawa.pdf

KEGGの例（3）



http://www.open-bio.jp/archive/20070302_OB6/OB6-KEGGonRails-Kibukawa.pdf

つまり

- 複雑過ぎるモデル構成は、実装できない！
  - できないというと言い過ぎだが、必要以上のモデル化は実装に過度の負荷をかけてしまい、現実的な速度で動かない可能性が高くなる（かも）
  - SIB1Railsの場合は、ポリモルフィズム（条件付けのために使用）によるものか？
  - KEGGの事例が参考になる（かも）

- →SIB2のモデル化にあたっては、プロトタイピングを通じてモデルの見直しサイクルを取り入れる事が必要ではないか？

RailsをSIB2入力I/Fプロトタイプの試作

- 目的
  - SIB2 のデータ入力インタフェースを、Ruby on Rails を用いた Web アプリケーションとして試作する。
    - ※Ruby on Rails はWebアプリケーション開発のためのWebフレームワークの一つ
    - ※Ruby はオブジェクト指向のスクリプト言語
- 獲得目標
  - SIB2入力モデルのプロトタイプをWebアプリで実装する。試作サイクルを通じて得た知見を当初のデータモデルにフィードバックして修正を加え、最終的にはGSTOSや診断システムやコマンド計画系にも適用可能なデータモデル構築に役立てる。
  - プロトタイプを衛星機器開発者にも試用してもらい、SIB2モデルを考える際にどのような入力UIがわかりやすいかを探り、正式版の開発にフィードバックする。

技術的に新しい点

- フィードバックを頻繁に行う「アジャイルプロトタイピング」の特長を生かした新しい開発スタイルを、モデル開発とUI開発の両方に適用する点が新しい。
  - Rails コンポーネントの一つである ActiveRecord を利用するため、モデル構築が極めて簡単になる。机上で行ったモデル設計を具現化し、実装の実現可能性を検証し、プロトタイプの設計結果から問題点を洗い出して、本番のモデル開発時の設計品質を高めることにつながる。
  - 元々のモデルが多次元データであるのにも関わらず、Excelベースの入力UIは二次元である事から、どのように設計しても多くのビューを提供する事は難しい。と言って、SIB2入力のUIに対してどのようなUIが望ましいかについて、装置開発者のコンセンサスがある訳でもない。Rails ベースのWebインタフェースを用いれば、元のモデルから演繹できるビューを簡単に数多く提供でき、ユーザの意見に基いて改善を図ることができる。また、多少の技術力があれば、ユーザが必要とするビューをユーザが自力で作成する事もできる。

まとめ

- Railsの紹介
  - 特長、思想（CoC, DRY）、キーワードなど
  - サンプルアプリの作成
- アジャイルなプロトタイピングを試行してみたい
  - 科学本部での開発スタイルには向いてるのでは？
    - モデリングのように、複雑で訳のわからないもの
    - UIのイメージが涌かないもの
  - あくまでプロトタイピングにフォーカス。最終的な実装は Struts/Tsunagi という住み分けが可能