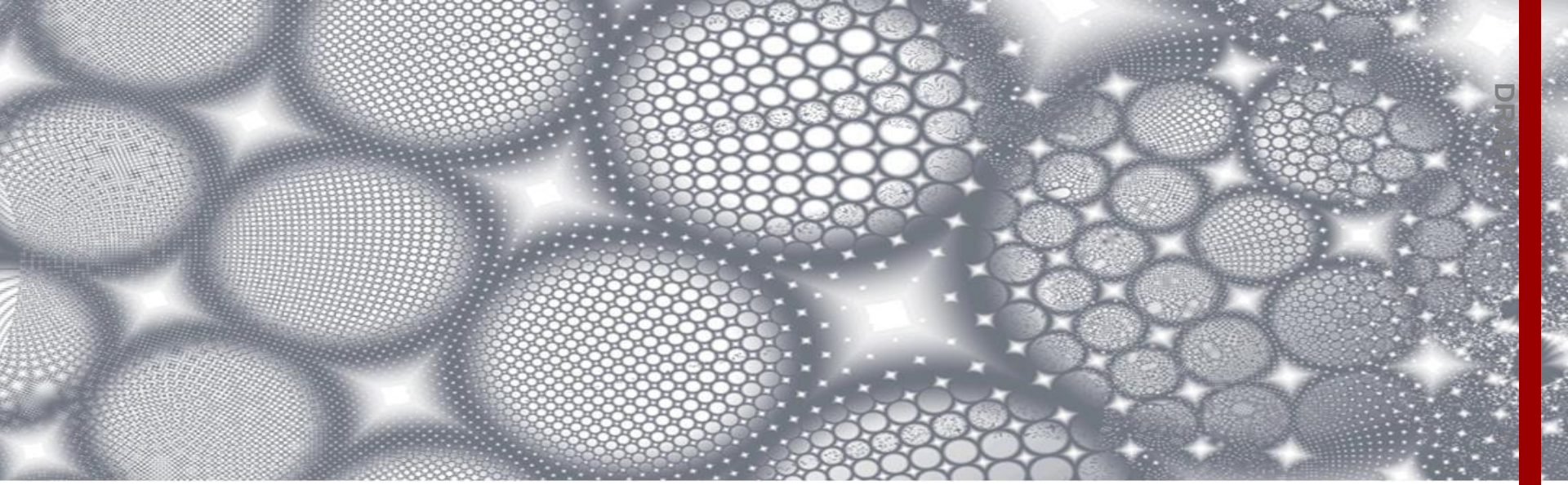


MATLAB-Based Policy Simulator

Regulatory & Risk Analytics (RRA)

Prepared by Seth Aslin

Date: October 2013

HSBC GlobalRisk

PUBLIC

Presentation Outline

- Background and context to **Project Navigator**
- General project objectives
- Project scope
- Overall approach (and rationale)
- MATLAB-based policy simulator
- Lessons learned (so far!)

Background / Context

- 2012 saw HSBC initiate a period of strategic consolidation from a commercial, operational and technical perspective

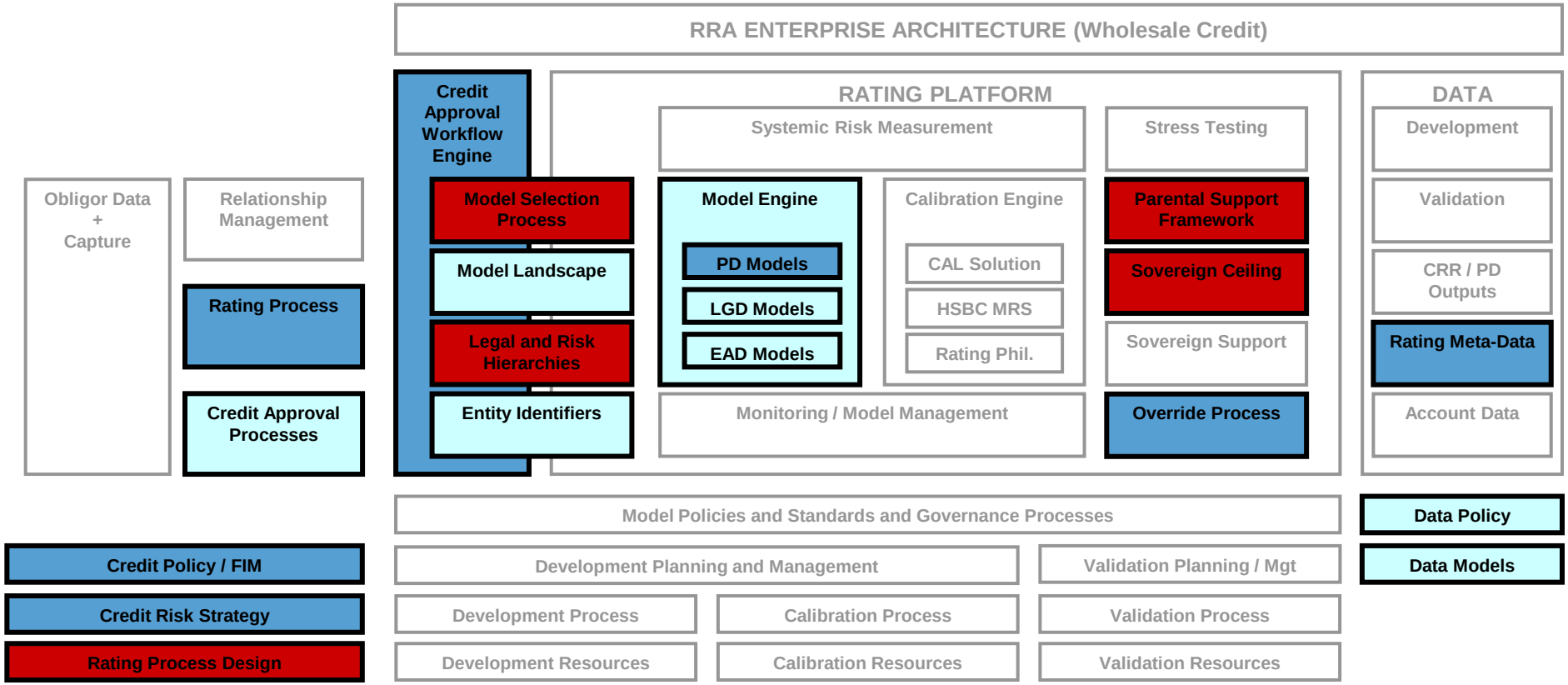
- For the global risk function, **Regulatory Risk and Analytics**, this has meant:
  - a stronger focus at the centre of the organisation in London, coordinating risk-related projects in multiple countries and within multiple regulatory jurisdictions
  - Increasing the level of standardisation across risk methodologies, technologies, governance formats and data models

- WCMR / RRA has an objective to **improve the 'consistency' of the global rating system** and has embarked on a programme of system-wide change to achieve this consistency

- Several separate projects contribute to the Consistency Programme – **Project Navigator** is one such initiative

- Project Navigator is focused on **improving <u>rating assignment</u>** – the process of selecting the appropriate 'path' for a rating to take through the various models and rating modifiers that comprise the overall rating system

Tactical Heat Map for **Project Navigator**



Project Objectives

- The project's high level objectives are:

(1) To formalise and improve the logic used in determining a given wholesale rating's ***assignment strategy***, including the effects of any ***rating modifiers***, towards the final rating;

(2) To automate this logic via algorithms embedded in the banks primary rating systems

Project Scope

- The following aspects of the rating system are in scope:
  1. **Rating assignment strategy** – the overall strategy that determined the *basis* for a rating, including any models, modifiers, etc used in the determination of the rating
  2. **Rating Model Selection** – the process of objectively selecting a model to rate an obligor based on (1) the borrower's attributes and (2) the collective scope set of the current PD models
  3. **Parental Support Framework** (the 'PSF') – the replacement of a stand-alone rating with a parent's rating (or notched derivation)
  4. **Sovereign Ceilings** – the application of a ceiling to obligor ratings due to (principally) transfer and convertibility risk ('T&C' risk)
  5. **Risk Hierarchies/Networks** – recognising the effects of control, and economic interdependency, through networks of relationships
  6. **Rating event audit trails** – implementing an information rich body of meta-data that captures and accurately reflects all of the attributes, decisions and thresholds that led to the final rating

Overall Project Approach

- Solve all related problems within the same cycle (= ‘anything to do with rating assignment’)
- Form a cross-functional team from Credit Strategy, Policy, Analytics, Technology
- Project attributes:
  - High complexity (especially in the interaction between components)
  - Unknown levels of regionally-based variation in requirements and regulatory constraints
  - Rapid changes anticipated throughout project
  - Aggressive timelines set by senior management
- Our conclusion: use a simulator … a **policy simulator** … in a **rapid prototyping mode**
- A simulator implies coding … but what sort of code exactly?
  - In what language?
  - With what structure?
  - Developed in what environment?

Further Considerations

Points to consider:

1. modular-but-interrelated nature of the core problems…
2. unknown location of final solution(s)…
3. the requirement to conceive, implement and test certain algorithms…
4. a need to somehow process a large number of policy ideas (rating scenarios)…
5. the need to visualise results quickly and easily…
6. the need for several people to work on the problem **simultaneously**, passing fragments of code around and then integrating them into the simulator…
7. plus a maintainability burden due to high levels of changes within the project…

… all implied that **(1) an object-oriented approach in (2) an interpreted environment** would be ideal

- MATLAB in its object-oriented mode is almost perfectly designed for this sort of work

De-romanticising the Development Process

Traditional (idealised) development cycle:…

1. Business problem identified
2. High level solutions generated
3. Formal requirements elicited
4. Functional specification drafted
5. **Development**
6. Internal testing
7. User Acceptance Testing (UAT)
8. Release to production

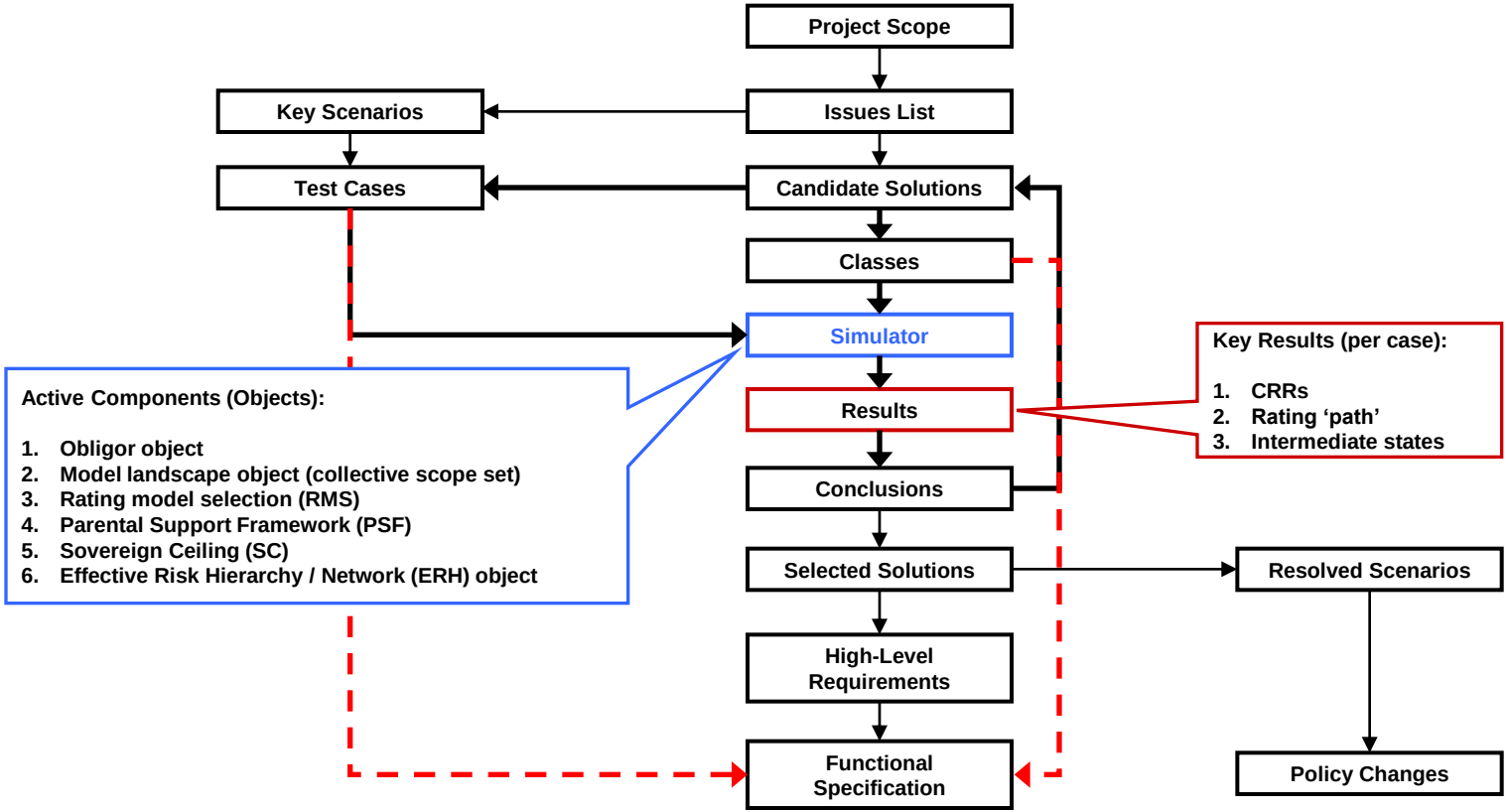
…and what can happen in practice:

1. Identify business problem
2. High level solutions generated
3. Formal requirements elicited
4. Functional specification drafted
5. **Development**
6. … logical problems in business concept uncovered
7. … minor/major revisions to key concepts
8. ... *re-commencement* of development
9. Internal testing
10. User Acceptance Testing (UAT)
11. … further concept revisions arising from UAT
12. … remedial development work
13. … *additional* (hopefully final) round of UAT
14. Release to production
15. … bugs identified from the live environment
16. … remedial programming
17. … re-release

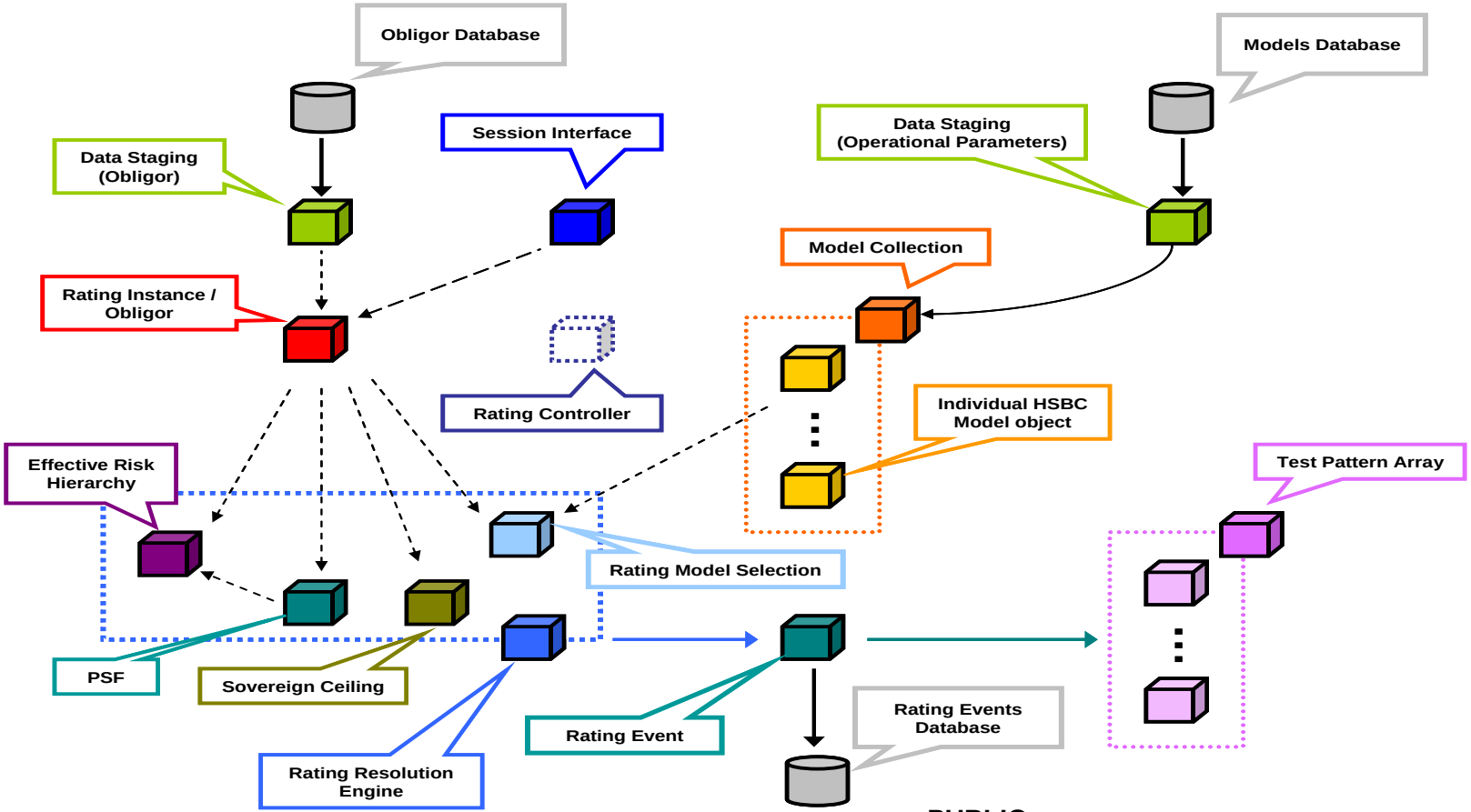
High-Level Structure of Problem Solving Process

- **Identify** issues (central themes, specific problems, etc)
- **Frame** issues as rating scenarios requiring a formal resolution
- **Examine** and set the HSBC ‘house view’ on key issues
- **Define** the solution for each rating scenario (potentially involving a wider group of stakeholders)
- **Convert** rating scenarios into benchmark test cases and/or map them on to actual cases
- **Generate** an array of additional test cases to probe solution boundaries
- **Code** the solutions into the simulator (directly altering modules to reflect solutions, policies, etc)
- **Run** the test cases through the simulator, collate
- **Review** strategies and solutions

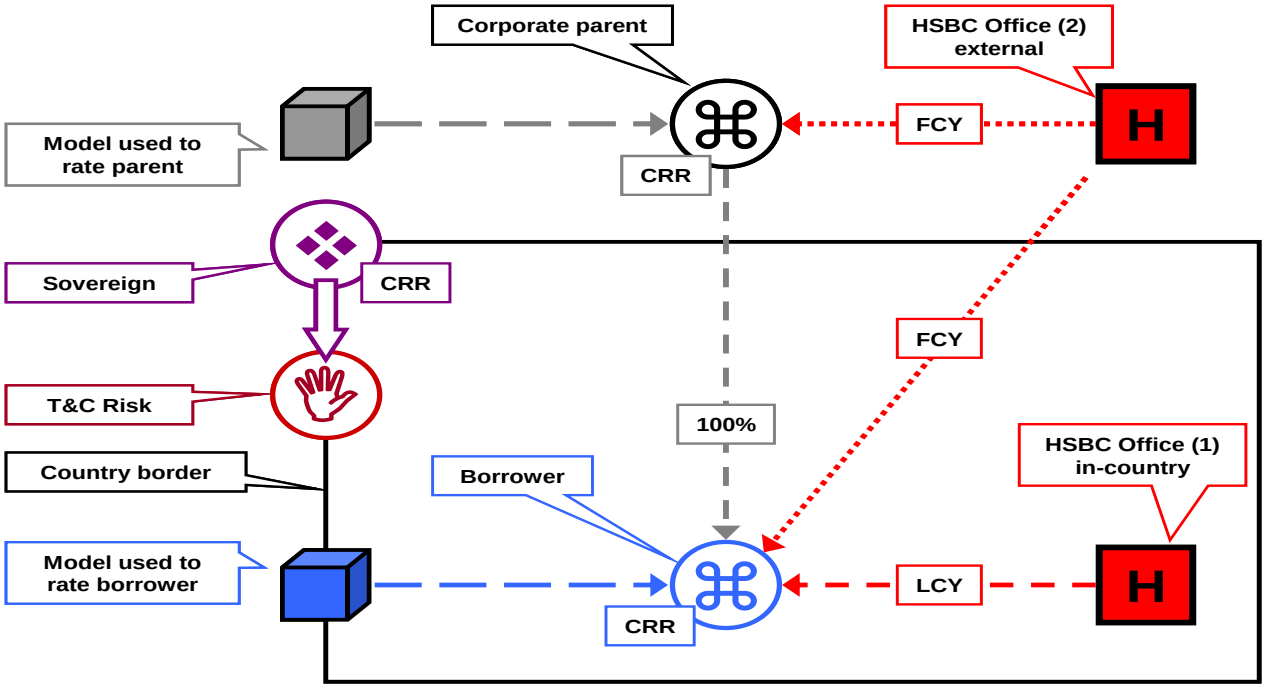
Central Work Cycle



Class Library Overview



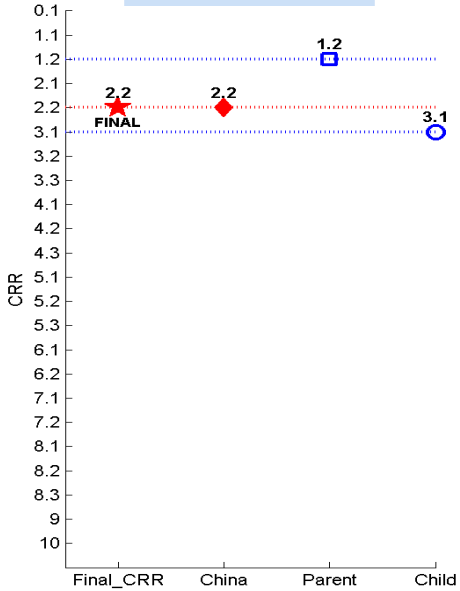
Visual Scenario Template



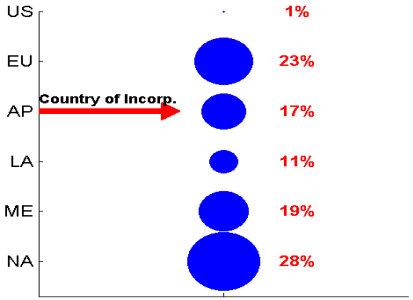
Test case 001 (example only)

SERIAL NO: TestCase-001

CUSTOMER RISK RATINGS



DISTRIBUTION OF TURNOVER BY REGION



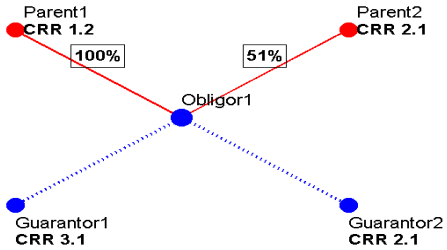
RECOMMENDED MODEL(S) TO RATE OBLIGOR

GLCS Model	**100%**
Sovereign PD Model	**50%**
Banks PD Model	**20%**
Mid-Market PD Model	**10%**

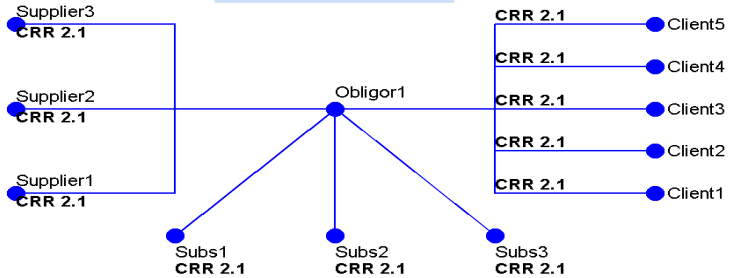
INDUSTRY

- 30% - 4500 - Construction
- 25% - 5500 - Hotel
- 20% - 5200 - Retail
- 15% - 7000 - Real Estate
- 10% - 5000 - Auto

EFFECTIVE RISK HIERARCHY (ERH)



NETWORK PERSPECTIVE



Approved by Working Group: YES/NO

Navigator Class Library

- Relatively large number of classes given the number of sub-problems involved
- High levels **encapsulation** of every functional aspect of the Navigator project
- **Namespaces** (packages) and class folders used to organise code
- Methods written to keep the '**public interface**' as clean / standard / logical as possible
- Delegation principle followed to keep the classes as loosely coupled as possible
- **Collections of objects** used for managing the rating model selection problem
- Classes become natural 'locations' or 'hooks' for key algorithms (e.g. classification, matching, etc)
- 'Composition' based relationships fairly common (very little 'inheritance' at this stage)
- Boundary elements (classes) are mock objects for future interfaces, and points of integration

Next Stages

- The base library has been developed for all sub-problems plus a foundational structure for running test cases
- In the next phase, we begin **running policy test cases, refining the code, mocking up interfaces** to simulate the key user cases
- Now that more of the geographically-specific requirements are known, the class library will also undergo a metamorphosis that accommodates these regional variations
- To avoid creating a mess of rigid code, we will now start using **design patterns** more liberally across the simulator design
- We will be using the **‘Strategy pattern’** to give certain modules the flexibility they need to choose a bespoke set of parameters, constraints or algorithms based on the attributes of the Obligor object
- We will simplify our control of the overall application using the **‘Observer pattern’** to coordinate the interaction between classes
- This last step will allow us to **keep all of the modules loosely coupled** – a desirable quality given that the final set of solutions will probably be distributed across
- All of the design patterns we have encountered so far (mainly from JAVA texts, but also Python and Ruby texts) seem to translate relatively easily in to MATLAB

Lessons Learned

- MATLAB is a brilliant prototyping environment – especially for non-programmers!
- Scripting is seductive but the discipline of classes/objects is worth the effort
- Class libraries are time-consuming to create in the first place but the modular structure has already shown benefits
- MATLAB's object-oriented syntax is clean, simple and easy to understand
- We have found pure 'test driven development' (TDD) to be a challenging approach to maintain in the face of (very) high ambiguity and (rapidly) changing requirements
- We prefer a sort of loose, informal prototyping that proceeds from idea → scripts & functions → script-facilitated classes → classes wrapped in a GUI-managed session (a quasi-app)
- We've learned through experience to integrate as continuously as possible
- GUIs are extremely simple to code in MATLAB – we recommend building GUIs for any repetitive task (and make the decision early to build them)
- Important to position the prototype correctly in people's minds