Decisions as a service for application centric real time analytics

Patrick Tendick*
Avaya Labs Research
211 Mt Airy Rd
Basking Ridge, NJ 07920
ptendick@avaya.com

Audris Mockus
Avaya Labs Research
211 Mt Airy Rd
Basking Ridge, NJ 07920
audris@avaya.com

ABSTRACT

The need for application-level intelligence can not be easily satisfied with existing architectures or methodologies that separate methods and tools for application developers and data scientists. We aim, therefore, to develop a framework (an architecture and a methodology) to make it possible to add intelligence capabilities to existing applications (decision-enablement) and to facilitate building new decision-enabled applications. The proposed approach starts by instrumenting the existing code with logging and instrumented decision points. The execution of the application produces information that is initially used to reengineer its behaviour and then the decision points are used to conduct search-based experimentation to optimize its behavior. This lightweight instrumentation allows the application developer and data scientist to fully exploit their capabilities, with the framework providing the glue needed to put their work together easily and transparently. In particular, the analytic capabilities, such as analysis of operational data are better dealt in the decision-making part, without complicating the mechanics of how the application functions. We plan to apply this framework to decision enable existing systems and to build new systems from scratch and measure the effectiveness of the approach and of the resulting products.

1. INTRODUCTION

Increasingly, organizations are looking to predictive analytics to guide interactions with customers and other users of software applications. These applications have grown to include IVR applications, Web apps, mobile apps, automated chat, and online gaming. The goal is to use the increasing volume of available data to learn how to alter the behavior of apps in real time. While traditional data analysis and learning is inherently an offline process, making decisions in real time is fundamentally different, but may involve the evaluation of a previously fitted model or classifier [11]. To this end, several scoring or decisioning engines have emerged in recent year, providing the ability to apply a previously fitted model or a set of decision rules to new data in milliseconds [1, 4].

Despite these advances, however, there are many challenges. Interactive applications are almost always event driven, which means that the modules and routines of the app execute at unpredictable times and in an unpredictable order, according to the whims of the user. Still, analysts and data scientist generally take a decision centric approach, in which a specific type of decision is considered outside of the context of the application. In this approach, the data scientist typically develops a model that somehow produces a prediction or score, and the application developer must adapt this output to the app using an interface specified by the data scientist. A decision centric approach is harder to implement on the application side, however, and tends not to address the range of decisions made by the application nor leverage the data that is available to the app in real time. To address these issues, we propose an application centric approach to predictive analytics and real time decisions. Under this approach, an application requests a decision that is meaningful to the app's internal logic, using a single interface that is independent of the tools used by the data scientist. The proposed approach generates and leverages massive amounts of semi structured data to improve the performance of interactive applications.

2. OPPORTUNITIES AND CHALLENGES

2.1 Evidence based design and A/B testing

Commercial enterprises have always sought to increase business performance and improve user experience. However, for applications targeted toward consumers, the need to provide a compelling user experience is much greater. Unlike traditional transaction processing systems for which the user base consists of business employees, consumers may simply opt to not use a Web site or mobile app if it does not provide a satisfactory experience. The principle of evidence based design says that we should use empirical observations to guide software implementation [6]. A common, concrete example of this is the practice of performing *A/B testing* to choose between two or more interface design elements based on the random assignment of candidate designs to users, with the resulting data being used to select the design that results in the best expected outcome, e.g., click through rates or conversions [7].

*

DOI: 10.1145/1235

2.2 Next best action, next best offer, and online decision making

Whereas the result of evidence based design and A/B testing is typically a static change in the software application logic or content, we may instead use data to guide dynamically how a system reacts or makes decisions in real time. For example, *next best action* is the principle of using predictive models or machine learning to select the best action to take at the current instant, given all of the information currently available. For example, an IVR (interactive voice response) application in a call center could make a decision about whether to continue asking a caller questions or simply route them to a human operator, based on all of the inputs the caller has provided so far. Such a decision could be based on a decision rule that tries to balance call outcomes (successful handling of the call) against the cost of having an operator decide how to route their call. *Next best offer* is a special case of next best action that seeks the best product or service to offer a customer, based on all of the information known at the moment. For example, a customer booking an airline flight on a travel site might be offered travel insurance or a first class upgrade, depending on things the site knew about the customer previously, and also on inputs the customer has made while selecting the flight.

To this end, *scoring engines* are server applications that evaluate predictive models or machine learning algorithms on new observations. Most scoring engines now have the capability to return results for a single new observation in real time, typically with latency measuring in milliseconds. Both commercial and open source scoring engines are available, and most are available as RESTful Web services [4]. Scoring engines are available that work directly with analytical languages or frameworks like R or SAS, and also based on the PMML standard for describing predictive models [10]. Scoring engines can provide numeric outputs that estimate propensity to default on a loan, for example, or they can provide qualitative outputs that correspond to recommended decisions, like whether to offer a certain type of insurance policy.

Decisioning engines, in contrast, are specifically intended to render business decisions, typically through the execution of business rules. For example, the decision to offer a home loan often involves the application of business rules that describe lending policies. Decisioning engines may also evaluate statistical models or execute other algorithms in the process of providing a decision. Henceforth, we will use the term scoring engine generically to refer to either a scoring engine or a decisioning engine. Since a scoring engine or decisioning engine typically uses some external tools to obtain the models or decision rules, we will use the term *scoring framework* generically to refer to a scoring or decisioning engine and other tools used in conjunction with it.

2.3 The data scientist versus the application developer

The use of data driven methods (statistical models or machine learning) to improve application performance in real time typically requires the involvement of two disparate roles: The data scientist and the application developer. The data scientist collects data, analyzes it, and fits models or performs other types of learning. Typically, the data scientist seeks to make predictions based on *predictors*, which are known attributes of the user or interaction that can be shown to predict future outcomes. The application developer, in contrast, is responsible for delivering and maintaining a working operational system. These two roles have very different skill sets, tools, philosophies, and worldviews, and follow very different processes that use very different terminology. These differences present formidable obstacles to successful implementation of data driven decisions in software applications. If we can understand these differences, perhaps we can provide a better way of approaching real time decision making.

Typically, the data scientist or analyst must

- Collect data on outcomes (the things to be predicted, e.g., the customer buys something) and predictors (the variables used to predict the outcome, e.g., customer demographics, products viewed, purchase history).
- Understand how the application works and how the data relates to it,and identify decisions that the application makes or could make.
- Synchronize the predictors with the decisions (specify what data is known at the point in time that a prediction should be made).
- Extract data sets for analysis.

Data scientists typically rely on data obtained offline through batch processing to train predictive methods. For this reason, efforts to use predictive methods to make decisions in applications often do not incorporate data that can be obtained from the app in real time, either because that data is not available to the data scientist offline, or because the data that can be obtained offline cannot be captured in real time, even if that data has changed.

In addition, interactive applications like Web apps, mobile apps, and IVR systems are typically event driven, that is, the application code consists of a collection of event handlers that respond to user inputs. Such systems do not necessarily behave in predictable ways, since they are responding to unpredictable inputs by users. For example, a Web site must be able to respond to the user clicking on any one of many possible choices, at unpredictable times and in unpredictable orders. Contrast this with a traditional outbound email marketing campaign, in which a server simply executes a batch job by reading from a list of tasks in a prespecified order.

Typically, the application developer must

- Know what predictors to capture and capture them.
- Modify the application to make predictions for the decision points.
- Understand what is being predicted and relate the predictions to possible actions.

3. DECISIONS AS A SERVICE

3.1 Application centric decision enablement

The approach currently used with scoring frameworks is *decision centric*, that is, an analyst uses these frameworks with a specific prediction problem or decision in mind. Applications must then conform to the interface for the appropriate engine and decide how to incorporate the output of

that engine into their processing. In contrast, we propose an *application centric* approach, in which an application identifies the decisions it must make, and then queries a simple API or service to provide those decisions. We call this approach *application centric decision enablement*, which is provided through a *decision framework*. Specifically, the decision framework consists of the following components:

- One or more *applications*, each of which is part of a larger *system*, of which there could be many. The applications make decisions at points called *decision points*, which are specific locations in the application code at which the decisions are made. For example, an e-commerce application might need to make a Boolean decision as to whether it should offer a customer free shipping.

- *Decision APIs* that the app calls at the decision points to provide decisions. For example, the e-commerce app might call an API to obtain a Boolean decision that is whether to offer free shipping. We will refer to the calling of an external API not associated with a specific scoring framework to obtain a decision as *decision injection* (DcI) after dependency injection [12].

- A *decision broker*, a server that acts as an intermediary between applications and scoring frameworks. In a typical scenario, an application would make a decision by calling a decision API, which would in turn make a call (a decision request) to a decision broker, which could then call a scoring engine and pass the results (the decision response or recommendation) back to the application through the decision API. We will refer to the calling of an external service not associated with a specific scoring framework to obtain a decision as *decisions as a service* (DaaS).

- Scoring frameworks, which would be invoked by the decision broker.

- *External data sources* like databases, data warehouses, transaction processing systems, or message streams that contain data relevant to the decision.

- A *data store*, a database or other repository that would store decision requests and responses, plus other data, for further analysis. Depending on the applications and decisions, the repository could range into petabytes in size.

Figure 1 shows the architecture of the decision framework. The goal of application centric decision enablement is to enable an application to request decisions that make sense in terms of its internal logic, based on the information it has at the time. The decisions could be of any data type, including Boolean, numeric or string valued, or URL or object valued. The values returned could represent yes/no decisions, the index of a choice from a list, a list of options to offer the end user, or HTML links or content to display.

When calling the API, the application provides all of the relevant information it has collected about the customer and the session e.g., demographics, session duration, products viewed, up to that point in time. This information could be provided explicitly by the app, in the sense that the app has to pass it to the API somehow, or it could be provided implicitly through session variables or some similar mechanism provided by the application framework in which the app runs. For example, the Java servlet framework provides the capability for servlet apps to store session variables that are then carried by the session. A decision API for the servlet framework could simply read the session attributes automatically, and marshal them when invoking the decision broker. Due to the event driven nature of interactive applications, most frameworks for developing them have a mechanism for storing session attributes, and that mechanism can be leveraged by the decision API to obtain application data automatically.

Modern interactive applications like Facebook and Amazon operate at Web scale, and business decisions need to be made in real time, so the decision broker needs to be lightweight and horizontally scalable, and have low latency. In addition, interactive applications like Web apps, online games, mobile apps, IVR apps, and chat bots typically do not simply engage in isolated interactions with a user. Instead, each interaction (which would typically correspond to a single invocation of an event handler) is part of a larger session that may encompass multiple applications. For example, in a single session with an e-commerce site, the user may view several Web pages for products, each time invoking a handler for a page request, place a product in their shopping cart (another handler), check the status of existing orders, go to checkout, etc. In this context, the notion of an application is somewhat vague and arbitrary, since the entire site could be viewed either as one application, or as a collection of apps (shopping, order status, account settings, checkout, etc.). The apps might be written in a single language and be hosted on a single hardware and software platform, or the apps might be written in several different languages and be distributed across several platforms. In contrast, the notion of a session is usually pretty precise, since sessions must be managed precisely by the environment in which the apps run. For this reason, we will also concern ourselves with the notion of a system, that is, an entity that encompasses user sessions and within which applications run. Business enterprises are often concerned with providing a unified customer experience, one that takes into account everything that has happened to the user recently. For this reason, the decision API and decision broker should also keep track of entire user sessions and not just individual decision requests. This includes tracking the beginning and end of the session, and keeping track of the application from which each decision request is made. The broker should also try to keep track of the time a user spends in a particular application, something we will call an application session. One session may be broken into many application sessions, each associated with an application, and a given application may be associated with multiple applications sessions in a given session.

The application centric approach has several advantages:

- It enables applications to focus on incorporating decisions into their logic, without worrying about how those decisions are made.

- It focuses the analyst on making decisions that are relevant to the application and are based on data that is available at the time the decision is made.

- By moving decisions out of application logic, systems become more agile and robust. For example, a decision

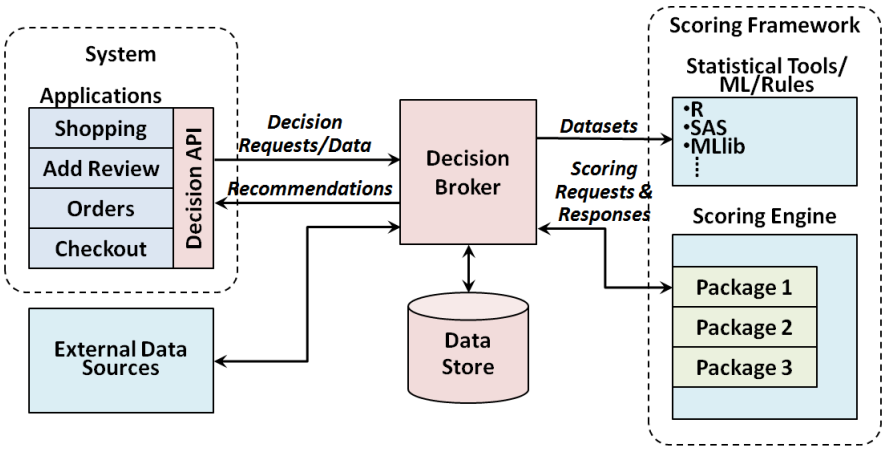

Figure 1: Architecture of the decision framework

about whether to grant a loan that is to be consumed by an online loan application can be easily modified to include additional business rules without changing the loan app itself. Or, the decision logic can be changed in response to changing business goals. For example, the goals of the business might shift from maximizing revenues to maximizing profits or to increasing customer satisfaction.

3.2 Data collection

An important part of application centric decision enablement is the collection of data. The purpose of this data is multifold:

- If the goal is next best action or next best offer, then the decision framework needs to collect predictor variables, that is, variables that predict the outcome of the interaction.
- In addition, if the goal is to make a decision that leads to the best outcome, then the decision framework also needs to collect variables that represent the outcome of the interaction.
- The decision framework also needs to collect variables representing the recommendations made.
- The application might not actually use the recommendation provided, however. In this case, the decision framework would need to collect the action actually taken by the application.
- If a decision rule is to be based on a statistical model or classifier, then the decision framework needs to collect observations that can then be used to construct training sets.
- If a decision rule is based on a predictive model or classifier, then it must consume predictor variables in real time to provide a decision.

All of the predictor variables must be known at the decision point corresponding to a decision. However, outcome variables might not be known until later in the session or not until after the session has ended. For example, we probably would not know when someone requests to view a page for a product that the person has purchased the product until later in the session or at a later date.

Data can be captured and integrated at several points in the decision framework. Specifically, data can be captured in the application, in the broker, or in a scoring framework. For example, in an e-commerce application, the broker might do a lookup in a CRM system to see if the customer has exceeded a certain volume of purchases. There are tradeoffs between the different integration points. For example, if data is not required by the application logic beyond making a decision, then that data should probably be integrated in the broker or in the scoring framework. On the other hand, data integrated in a scoring framework would probably not be available for use outside of that framework. In addition, data may require transformation or feature extraction before being passed to a scoring framework. For this reason, the decision broker is an ideal point for data integration.

Data captured by the broker includes predictor variables, outcome variables, decision requests, recommendations, and actions taken. This data is stored in the data store for further analysis. For Web scale systems, this data could be quite voluminous.

3.3 Discovery

The decision broker only knows about applications, systems, and decision points through the decision requests it receives. One of the functions of the decision broker is to discover the systems, applications, and decision points associated with decision requests, and store the resulting metadata. This metadata may then be used to configure the broker to respond in different ways to handle different decisions on behalf of applications. In addition, the broker needs to discover the variables that are being provided by the applications and through data integration.

3.4 Experimentation

To obtain useful predictions, it is not enough simply to collect a large amount of data. In the shipping cost example, we would not be able to learn from the data to whom and when to offer free shipping if the site never offered free shipping or if it were only offered to members of a loyalty program. An often overlooked component of statistical reasoning is *experimentation*. To understand whether and how free shipping makes a difference, we really need to offer it to

randomly chosen customers, sessions, or interactions.

In the website optimization world, people use content experiments, also known as A/B testing, to choose between several alternative website designs. In the classical example, a website randomly assigns users, sessions, or interactions to one of two possible designs (A or B) [7]. After collecting data on a sufficient number of visits, the website selects the design that results in the best outcome, typically click-through rate or conversions. A/B testing uses a *stimulus-response* approach, an experimental approach in which the website provides a stimulus (design A or design B) to the user, and then observes the response. For example, a website might perform an experiment to identify the best background color for a web page, where A and B are two different color schemes, and the response is whether the user clicks on a link on the page.

In application centric decision enablement, we can use a stimulus-response approach also, but we can collect additional variables or covariates about the users, sessions, or interactions. For example, we can potentially find out not only which color scheme is best overall, but which one is better for different demographics (age, gender, locale, etc.). Support for experimentation is a key feature of the decision broker. The broker implements experiments by assigning treatments to users, sessions, or interactions. A treatment is a particular way of handling a decision request from an application. A treatment could always return the same recommendation, or it could return a randomized recommendation. Or, it could invoke a scoring or decisioning engine to evaluate a predictive model. Treatments could be assigned according to the

- User or customer – The customer always receives the same treatment each time. For example, we could randomly decide which customers will be offered free shipping, and then always give a customer the same shipping option (free or not free) every time they visit the site.

- Session or visit – The customer receives the same treatment throughout out a session. For example, we could randomly assign each session a shipping option, and then always give the customer the same option throughout that session. However, in the next session or visit, they could be given a different option.

- Page or interaction – The customer receives a possibly different treatment for every interaction they have. For example, the customer might get a different, random shipping option for each product page they view. Or, the customer might be offered a random shipping option for a particular product, and then be given the same option every time they view that product.

The collection of treatments and the scheme for assigning them to users, sessions, interactions, or other entities is called an *experimental design*.

3.5 Analysis and learning

Once we have performed an experiment to collect data on how users respond to different inputs, we can analyze the data and do learning (fit models, train classifiers, etc.). The broker has already identified the decision points and variables. Once the data is available, the broker can help an analyst to extract datasets for analysis and learning. The broker will probably collect more variables than the analyst needs, so the analyst must select the variables of interest. Also, the data collected by the broker tends to be semi-structured and follow a document tree structure (sessions contain application sessions, which contain decision requests, which contain complex application objects, etc.), so the data needs to be *flattened*, that is, converted into a flat, record format for consumption by analytical tools. The datasets are an input to scoring frameworks, and the results of the learning process are scoring *packages* that can be invoked by the decision broker. In terms of the mechanics, the broker could extract the data directly, or the data could be extracted from the data store using a tool like map-reduce or Spark.

After analysis and learning has been performed to obtain scoring packages, the broker must be configured to invoke the package as part of a treatment that it applies in response to a decision request. To invoke a scoring package, the broker must marshal the variables required by that package, which entails a similar flattening process to that used when extracting datasets.

3.6 The decision development process

Under application centric decision enablement, we can define a decision development process, similar to a software development process, consisting of the following phases:

- Business analysis – Analysts obtain an understanding of relevant systems and applications, and also of the decisions those applications make or could make.

- Instrumentation – Using decision APIs, application developers make minor modifications to the applications to capture data and incorporate decisions based on input from the business analysis phase, test the modified apps, and put the modified applications into production.

- Discovery – Based on decision requests submitted by applications, the decision broker discovers the systems and applications and also what decisions the applications make and what data is captured.

- Experimentation – Analysts uses the information obtained in the business analysis and discovery phases to devise experiments to obtain data for analysis.

- Learning – Analysts use the decision broker to identify predictor and outcome variables, extract datasets for analysis and learning, and use those datasets to fit models, train classifiers, or obtain decision rules, using the scoring frameworks.

- Implementation – Analysts construct packages to deploy in scoring engines based on activities in the learning phase, test and deploy the packages, and configure the decision broker to put those packages (treatments) into production.

- Evaluation – On an ongoing basis, analysts extract further datasets, which could be stored in a data warehouse or data lake, to evaluate the performance of the packages and treatments deployed. The analysis of this data provides feedback into the experimentation and learning phases.

These phases could be executed in sequential order as in a waterfall model, or they could be used in a more incremental or agile approach. The analysts may be different in some phases. For example, the analysts in the business analysis phase may be different from the analysts in the learning or evaluation phases. Note that each phase involves only one role, and the phases are loosely coupled, that is, execution of a phase typically does not require detailed knowledge of what was done in previous phases. For example, in the implementation phase, the application developer may choose to instrument the decisions identified in the business analysis phase, or may instrument different decisions instead. Whichever is the case, those decisions will be discovered during the discovery phase. This solves the problem that occurs when business functionality does not map to system implementation in a straightforward way.

4. RELATED WORK

Apart from the work cited above, the proposed system design builds upon existing ideas. For example, using program logs to design a better text editor traces back to at least to 1985 [3]. More generally, a good overview of logging in data intensive applications is presented by Olinear et al, [9].

Search-based software engineering, see for example [5], is an approach to use iterative optimization techniques in software design and engineering. While the main issue for the widspread adoption of search-based software engineering was a lack of clear objective functions to be optimized, this work suggest ways such functions could be constructed using observed behavior of the program.

Reverse engineering [2] is an approach to recover the design of the system for maintenance purposes. Behavior recovery based on the decision points can be thought as an approach to reverse engineer program's behavior.

The data obtained in the course of the program execution is, by nature, not data from a well designed measurement apparatus. As any other operational data, it requires special attention to treat incorrect, missing, and decontextualized events [8].

5. FUTURE WORK

While the proposed approach stems from a long industry experience of adding analytic capabilities to existing products, it needs to be validated by building a number of systems using the approach and obtaining developer feedback. For example, should the client libraries look like? The client libraries do a lot more than just invoke Web services, e.g., if a WS call times out, the library must still make a recommendation of some kind. It is not presently clear how different client platforms or frameworks compare, e.g., Java servlets vs. PHP. This affects how the client libraries work and how much they can do. Furthermore, it is not clear what kinds of experimental designs might someone want to do, and how best to support them. Finally, what does the collected data look like, and how do we approach it?

6. SUMMARY

We propose a framework for instrumenting existing systems and designing new systems with the ability to make operational data-based decisions in real time. The main objective of the framework is to separate concerns of the data analyst from concerns of the application designer by developing the concept of decision as a service and outlining ways to implement it. The data scientist does not need to be concerned with the precise logic of the solution, especially the logic of individual applications. The event-based nature of such applications would not provide much insights into the usage patterns, especially given the complex environment the application is running in. The user behavior is logged via the decision points and the resulting data can be both analyzed off-line or through experiments with real users to discover desired relationships and used, in conjunction with scoring engines, to make optimal decisions.

7. REFERENCES

[1] John Chaves, Chris Curry, Robert L Grossman, David Locke, and Steve Vejcik. Augustus: the design and architecture of a pmml-based scoring engine. In *Proceedings of the 4th international workshop on Data mining standards, services and platforms*, pages 38–46. ACM, 2006.

[2] Elliot J Chikofsky, James H Cross, et al. Reverse engineering and design recovery: A taxonomy. *Software, IEEE*, 7(1):13–17, 1990.

[3] Michael Good. The use of logging data in the design of a new text editor. In *ACM SIGCHI Bulletin*, volume 16, pages 93–97. ACM, 1985.

[4] Alex Guazzelli, Kostantinos Stathatos, and Michael Zeller. Efficient deployment of predictive analytics through open standards and cloud computing. *ACM SIGKDD Explorations Newsletter*, 11(1):32–38, 2009.

[5] Mark Harman and Bryan F Jones. Search-based software engineering. *Information and Software Technology*, 43(14):833–839, 2001.

[6] Barbara A Kitchenham, Tore Dyba, and Magne Jorgensen. Evidence-based software engineering. In *Proceedings of the 26th international conference on software engineering*, pages 273–281. IEEE Computer Society, 2004.

[7] Ron Kohavi, Randal M Henne, and Dan Sommerfield. Practical guide to controlled experiments on the web: listen to your customers not to the hippo. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 959–967. ACM, 2007.

[8] Audris Mockus. Engineering big data solutions. In *ICSE'14 FOSE*, 2014.

[9] Adam Oliner, Archana Ganapathi, and Wei Xu. Advances and challenges in log analysis. *Communications of the ACM*, 55(2):55–61, 2012.

[10] Rŏsler R. Predictive modeling using r and the openscoring-engine – a pmml approach. Data * Science + R. Available at: http://things-about-r.tumblr.com/post/37861967022/predictive-modeling-using-rand-the.

[11] Patrick H Tendick, Lorraine Denby, and Wen-Hua Ju. Statistical methods for complex event processing and real time decision making. *Wiley Interdisciplinary Reviews: Computational Statistics*, 8(1):5–26, 2016.

[12] Hong Yul Yang, Ewan Tempero, and Hayden Melton. An empirical study into use of dependency injection in java. In *Software Engineering, 2008. ASWEC 2008. 19th Australian Conference on*, pages 239–247. IEEE, 2008.