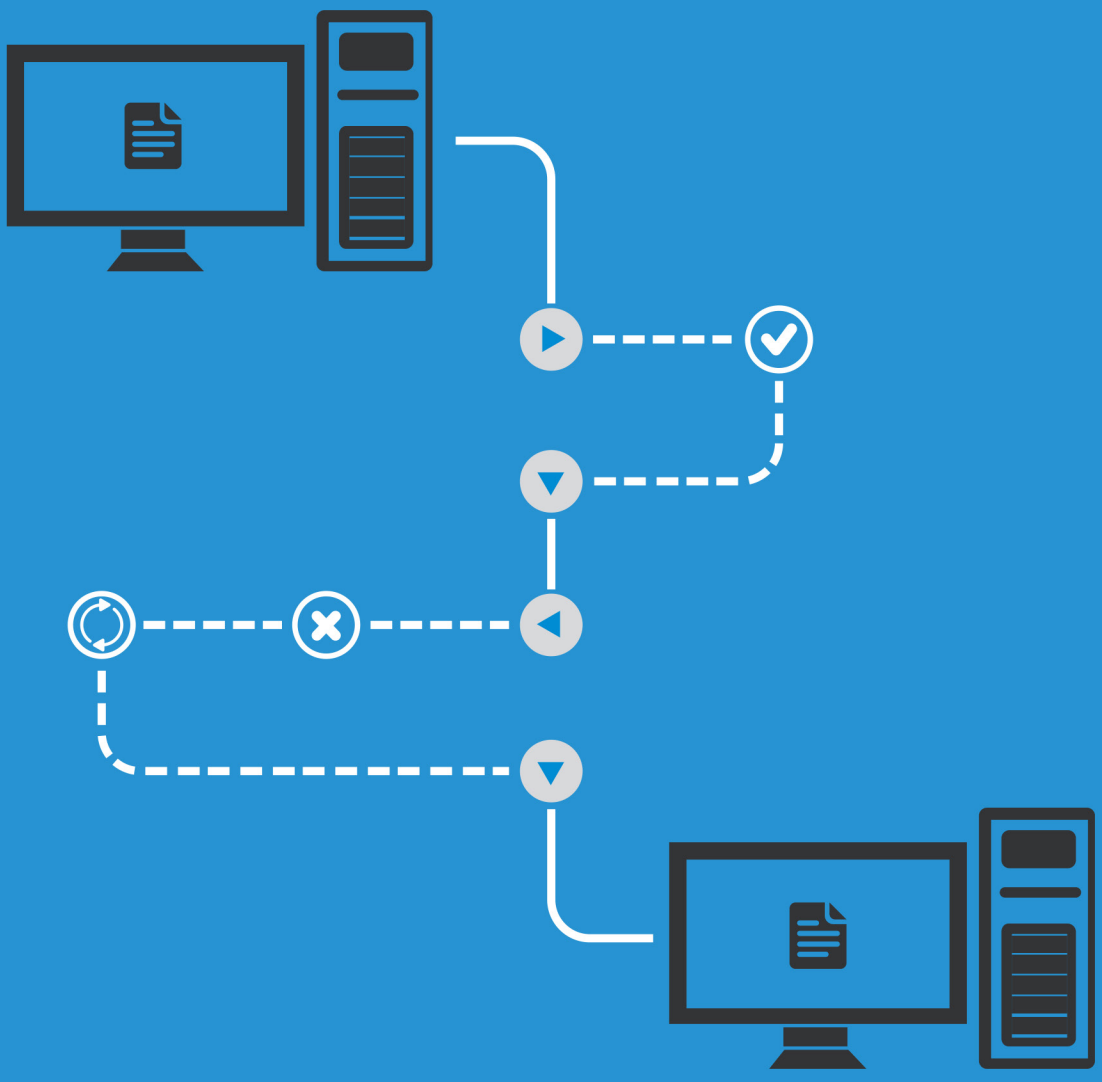
# DECISIONS

# INTERCEPTOR RULES

## Intelligent Business Rules in a Web 3.0 World



**Carl Hewitt**, Founder and Chief Architect
**Kevin Lindquist**, Director of Sales and Marketing
**Decisions.com**

# About the Authors

## Carl Hewitt
**Founder, Chief Architect**

In addition to being the founder of Decisions and other prior workflow ventures, Carl has been an innovator in object oriented programming for business since the 90's. Rumor has it he was raised by marketeers.

## Kevin Lindquist
**Director of Sales and Marketing**

Kevin has created and executed against growth strategies with many up and coming technology companies. He likes to think that he can run faster than Duane, sail better than Carl, and maintains a larger vocabulary than Heath, but in reality he is just happy to be here.

### Carl

Prior to starting Decisions in 2009, Carl began in the mid 90's as an innovator in object oriented programming for business. He started his first technology company in 1998 (oop.com) which developed a graphical configuration technology and rules engine. The company was sold to NetDecisions, a global technology company and private equity fund in 2001. As the CTO over technology ventures at NetDecisions, Carl also organized and created Fluency, a voice recognition technology based on the oop.com platform. In 2003, recognizing the potential of workflow in conjunction with other configuration technologies, Carl formed Transparent Logic. Using a .net based platform, Transparent Logic that delivered a fully graphical platform for creating workflows, rules and form building suite. Transparent Logic was sold to Altiris/Symantec in January 2008. At Symantec, Carl was the Senior Director in charge of R&D for the workflow team. During his time at Symantec, he created the next generation Symantec Service Desk, based on the workflow technology.

### Kevin

A Crocker Innovation Fellow with a degree from Brigham Young University, Kevin has a rich history in identifying pains and applying technology solutions. He has worked with a number of startups and success stories in the software and hardware space from Silicon Valley, The Silicon Slopes, and abroad including: Square, ASUS, and Fundly along with consultative roles at other Utah technology startups.
He has been mentored by a number of highly successful individuals, including an Entrepreneur in Residence at Disney, multiple venture capitalists, and thought leaders in the world of technology and entrepreneurship. Kevin has a deep appreciation for entrepreneurship and a passion for internet technology. He is constantly seeking ways to contribute to the next generation of web enabled tools, a primary reason for his involvement with Decisions.
Follow Kevin Lindquist on Twitter: @GrowthHackerGuy

DECISIONS

# The Rise of the API

Yes, the arguments and clichés abound as we start talking about APIs and Web 2.0. Less obvious to most who are engaged in this discussion is that systems have been doing the same thing in a software-to-software environment that social media has been doing in a human-to-human environment for some time. Older and more mature industries that had to communicate data have been doing this in a standard way long before Facebook came around. Examples of this are things like HL7 or support for basic file format types like comma separated files or XML. What has been quietly changing over the last decade is the fact that companies and systems have been formalizing the communication patterns and mechanisms to allow system-to-system interactions. The rise of the API is creating both opportunities and challenges.

## API Challenges

While the structure of the data can be formalized and agreed upon (JSON, XML, etc) it's much harder to make sure that the data that is sent inside this structure makes sense or uses the same semantics between the systems. This is often referred to as dirty data and any systems open to an API has to decide how to deal with it. Some systems accept the dirty data and expect a person to clean it up after it comes in. Other systems just reject it, putting additional work and load on the entire eco-system.

## API Opportunities

This dirty data problem provides an ideal place to start thinking about how to make your systems smarter. Instead of assigning somebody to sweep the dirty data under the rug, let's take a step back and look at the bigger picture. Consider the business rule engine. It can solve a number of problems, including: formalizing logic, making decisions in a consistent way, automatically making the decisions and adjustments that a person would normally have to do and more. Traditionally accomplishing these goals meant finding a way to insert the rule engine into the software or workflow engine that allowed these rules to be evaluated at the appropriate spot. Some software experts have recognized this and created hooks or even built in a rule/workflow engine into the product offering. Most have not. Even the ones that have often have used engines with legacy approaches that limit the usefulness of the engine. Inserting logic at the edges of the system, where systems are talking to each other, allows for greater flexibility in how rules are created, managed, structured, and utilized. We call this model of rule deployment the Decisions Interceptor Pattern.



# API Security Management

**86.5%**
Of organizations will have an API program in place within 5 years

**43.2%**
Currently have an API program

**27%**
Implementing one in the next year

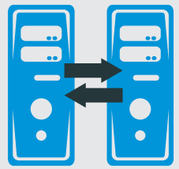
**12%**
Implementing one in the next 2-3 years

**4.3%**
Implementing one in the next 4-5 years

**13.5%**
No plans to implement an API strategy


## Top Business Drivers of API Program Adoption:


Mobility programs are driving **71.9%**



Partner connectivity is driving **69.1%**

Cloud integration is driving **67.6%**

Enabling internal developers is driving **66.9%**

Fostering external developer ecosystems is driving **55.4%**

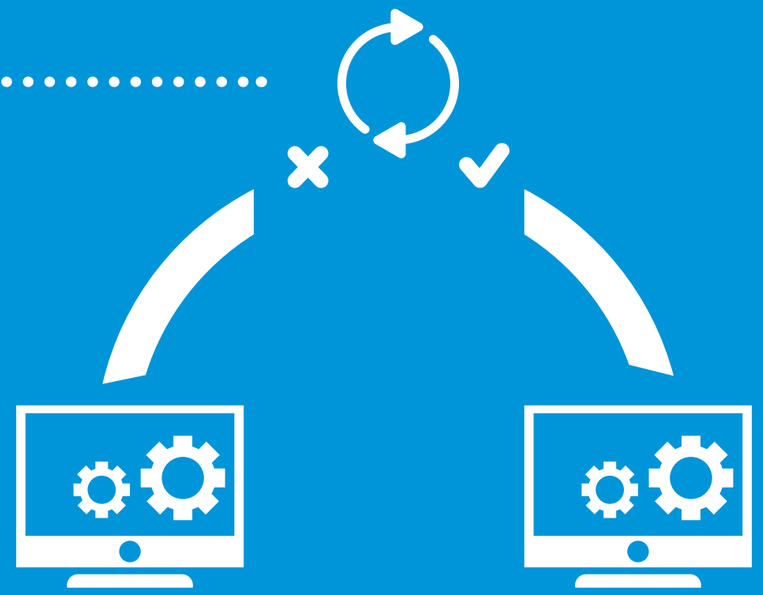
# What is the Decisions Interceptor Pattern?

At its essence, the Decisions Interceptor Pattern allows for business rules to be 'learned' and evolve based on real data interactions in the system. By putting the rule engine at the point where data is transmitted, data can be captured for rule analysis or intercepted. Intercepted data can be 'worked' (read forms/tasks/routing) by a user or be used as the basis to create rules that fit into the rule set for this interception point.

# Business Rules at the *Seams* of the System

The use of business rules often involves selecting the point in a work-flow or transaction that a rule fits in. For instance, using rules to do loan approvals in a 'traditional' application would mean inserting, at the right spot, a call to the rule engine and handle the results of the rule execution when the engine returns a result. While this approach is totally valid it requires architecting or adapting the calling application to incorporate the use of automated business rules.

Placing rules at the system boundaries provides a natural integration point for the execution of rules. Since data is in transit between applications the problematic understanding of exactly what the data means 'inside' the applications is minimized and it allows a great spot to change or augment the data that is going between applications. In the world of medical software, this could mean intercepting records of a doctor's visit from the EMR (electronic medical record) before it gets into the billing system. This same pattern is applicable to many industry verticals. Using the interceptor pattern, we can easily stop, modify or branch out a message between systems.



# Interceptor Rules

If you think about the data 'flowing' over the rule engine, there are two natural interactions:

1. one of them is to 'stop' an API message from going through based on the data,
2. the other is to record the fact that an API message was passed for later analysis and rules.

If we write a rule that 'intercepts' the API message, a user can be directed to review and write a rule based on the data. An example of this might be getting messages where an account ID is not found. The account ID might not be found based on a different schema or structure of the ID. The rules could be used to 'intercept' these. Additional rules could be used to correct this data or provide additional user interaction.

## Separating the Structure of the Data from the Data Itself

One of the biggest challenges novice or business oriented rule writers have is the idea of extracting the actual data from the structure of the data. While it makes sense, especially to programmers to say if 'Persons First Name' is 'Joe' I do something, this tie from structure to data is less obvious. But the ability to write a rule - against the structure - but seeing actual values of a data structure help make this very concrete. If a rule writer can see, for instance, that prior insurance company is 'Nationwide' then we might want to do something - like forcing a user task, changing some data, or sending for additional evaluation.

Seeing the data that actually comes across a specific point will allow a rule writer to see what values are in what spot of a data structure. While in a complex data structure there may be a few different spots that have potential data (billing zip, shipping ship, fulfillment zip, etc.) you can actually see the data being sent in. Given the fact that systems interacting with a specific API might send in slight variations of what is expected, seeing the real data while writing the rules makes the task much easier, and generally makes the rule much more effective in the end.

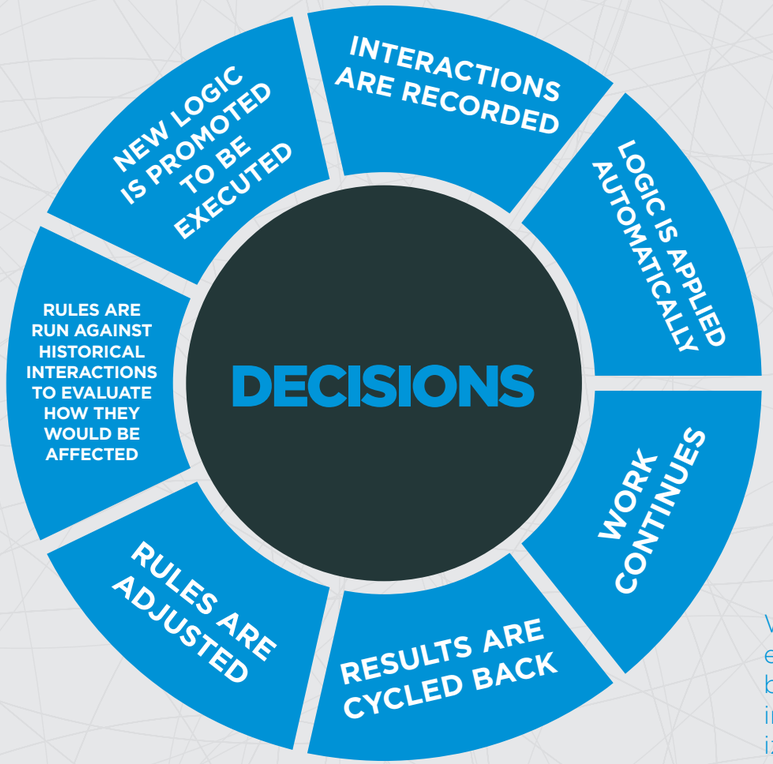
## Business Rule 'Sets' in the Decisions Interceptor Pattern

By definition, Interceptor Rules are less defined by 'the rule' and more by the 'point that a set of rules are executed'. The nature of an interceptor rule set is that rules should evolve based on additional data as they are used. While a set might start off with a set of very basic rules, as additional details are discovered about the interaction, more rules are added.

The rules in the rule set have to be in order, because they might trigger 'actions'. For example, if you have a rule that modifies data you would need to first have a rule that identifies the data and stops it, then the rule that modifies it, and finally sends the data back into the stream for the same rule that intercepted it to let it pass through.

# Writing Rules against Historical Data

Writing rules against live data 'about to be sent' to a system is a differentiating part of the pattern. This works because we have identified a specific point at which a set of rules can execute and know the data that is present. The 'normal' use of this pattern is to have rules that 'stop' or 'intercept' the interactions that will break or not be handled properly in the target systems, however, since we know the point, it's also possible to record off transactions that were not stopped but in a later process could be identified as a problem. The 'holy grail' of process/rule optimization is a cycle where:




While any rule engine (or even programmer oriented business logic) can be used in this pattern... the formalization of the pattern into something out the box can provide a massive strategic advantage and cost savings as the speed and accuracy of rule/process creation and optimization increases.

# What Results Do You Want?

Interceptor rules assume less about the structure of the rule and more about where the rule executes and the data that is present when this happens. When a rule is run, there are a number of specific actions that can take place. These actions may stop the process or trigger some other type of interaction. The rule is used to decide if an action applies. For instance:

- **account number is not valid**
- **format of procedure code is wrong**
- **amount of transaction is above a certain limit**
- **data is missing**

Given the true/false of a rule being applicable, there are a number of actions/results that can take place. Here is a list of the different types of rules that can be implemented that incorporate the central ideas of the Decisions Interceptor Pattern:

***Interceptor Rule:*** The process is stopped and the message is routed to an analyst to be evaluated and additional rules are created. This is the logical step to take when you can identify what is wrong, but need to take, case by case, the creation of automatic remediation. Remediation can take two forms, one is a 'one off' change where this person adjusts the data. The other is creation of a rule/process to fix or adjust this specific scenario.

***Data Change Rule:*** The most straightforward of adjustments is to change data. Someone puts in a zip without a city or state; adjust the data for the city or state. Someone sends in a message that is wrong because they were assuming the wrong data structure (days instead of years... so 200 years as a project timeline is not logical, for instance); automatically adjust the data. Someone sends in an ICD9 code instead of an ICD10 code; change the data before sending it on.

***Workflow Trigger Rule:*** For more complex data transformation, to do assignments or other processing, a workflow can be run.

***Re-Routing Rule:*** Rules can be used to direct data to different branches of the containing flow. For instance, if we are using rules to classify and route inbound communication, service requests can be routed to support team and sales leads can be routed to the sales team.

DECISIONS