teal.blue
digital solutions

from specs to code, smoothly

teal.blue
by: Maply SIT - VAT IT03820550162 - PEC maply.sit@legalmail.it
Headquarters: via Pasubio 5, Dalmine (Bergamo), 24044, Italy
For all enquiries: hello@teal.blue

1) Problem statement

1.1) Speaking the same language

Having all stakeholders in a digital project speak the same language is not easy.
Clients, project managers, UX designers, developers and testers have different ways to describe problems and solutions. This means in practice that there is ample room for misunderstandings. When misunderstandings emerge only late in a project, these might lead to all sorts of bad outcomes, including delays, increased costs, unhappy users, layoffs, legal actions… and stress.

Besides speaking role-specific languages like code, flow diagrams and UI mockups, however, all stakeholders share a common language: *natural language* (like English or Italian).

1.2) Behavior-driven development

Behavior-driven development provides a common language.
Behavior-driven development (BDD) leverages the untapped power of natural language to bring all stakeholders around the same table, discuss about product features, and find a shared understanding of what to expect from the product's behaviour.

A common BDD practice is to use standardised language patterns to describe a sequence of expected preconditions, an action, and its expected outcomes, describing a user's interaction with a product or system. Such sequence is called a *scenario*. One or more scenarios constitute a *feature*.

A quick ATM (cash machine) example:

```
Feature: Withdraw money


	Scenario: valid card, sufficient money
		GIVEN I have 30 Euros in my bank account
		AND my bank card is valid
		WHEN I try to withdraw 20 Euros from the ATM
		THEN I should be given 20 Euros
		AND I should be told 10 Euros are left on the account
```

The first benefit of describing features in this format (called *Gherkin*) is that important issues to be discussed should surface early on. For instance, the example above does not list all necessary preconditions for this use case to succeed (working data connection,

enough money in ATM), nor deals e.g. with the user's authentication method (PIN, fingerprint, bodyscan?).
The second benefit is that the use of a standardised language makes it possible to *link natural language specifications with application code*. This can be achieved with tools like *Cucumber*.

1.3) Shortcomings of current solutions

But natural language and code are usually linked via a manually maintained "glue" layer, and manually written APIs (function / method signatures).
Currently, the connection between scenarios written in natural language (tests) and application code (APIs) is fairly arbitrary. Usually, a string of natural language which contains placeholders for parameters is manually associated to a function or method call (API) by writing glue code. The name and signature of the function or method itself, which is part of the application programming interface, is usually invented by the person responsible for designing the API.

Test code (manually maintained "glue" layer):

```
…
Given('I have {float} Euros or more in my bank account', function (accountMoney) {
  this.availableMoney = accountMoney;
});

// When('I try to withdraw {float} Euros from the ATM', function (wantedMoney) {
  this.actualAnswer = withdraw(wantedMoney, this.availableMoney);
});

Then('I should be given {float} Euros', function (expectedAmount) {
  assert.equal(this.actualAnswer[0], expectedAmount);
});

Then('I should be told {float} euros are left on the account', function
(expectedMoneyLeft) {
  assert.equal(this.actualAnswer[1], expectedMoneyLeft);
});
```

Application code:

```
// manually written API
function withdraw(money, availableMoney) {
// manually written implementation
  var moneyLeft = availableMoney - money
  if (moneyLeft >= 0) {
    return [money, moneyLeft]
  } else {
    return [0, moneyLeft]
  }
}
```

With this approach, there is no guarantee that the natural language and the API “mean the same thing” or, at least, are strongly coupled. The only piece of data that they actually share are the input parameters (like `wantedMoney` in the example above).

2) Proposed solution

2.1) Narrowing the spec/code gap

We are looking for a toolkit to narrow the gap between project specifications and APIs.
What if the gap between natural language and application programming interfaces was narrower?
What if `I try to withdraw 20 Euros from the ATM` could semi-automatically be converted to a function with a signature like `withdraw(user, amount, source)`?

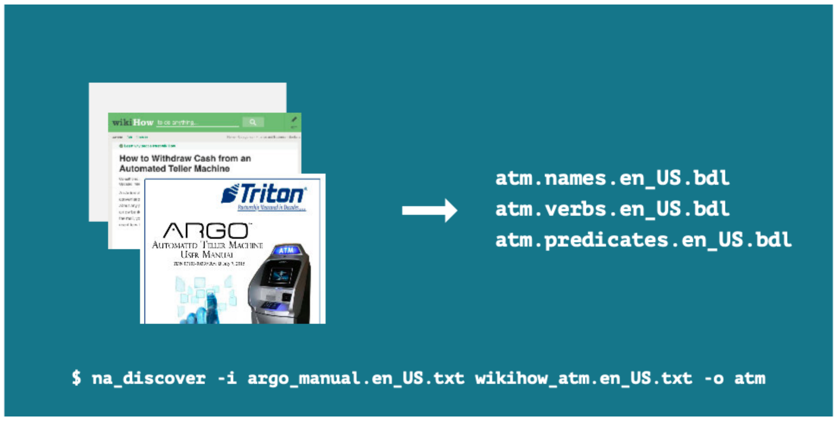
The aim of this project is to provide a proof-of-concept toolkit to narrow the gap between project specifications and APIs. We will name such a toolkit *NaturalAPI*.

Thanks to the right mix of **natural language processing** and **code generation**, *NaturalAPI* should allow the next generation of application developers to write APIs that are more consistent, more predictable, and more maintainable. By writing feature scenarios in controlled but natural language, project stakeholders will be able to turn project requirements (use cases, business entities) into entry points for actual code (the application’s high level API).

***NaturalAPI* should allow programmers to focus on developing functionality, rather than replicating business domain modelling efforts.**

As part of *NaturalAPI*, 3 proof-of-concept tools shall be developed:

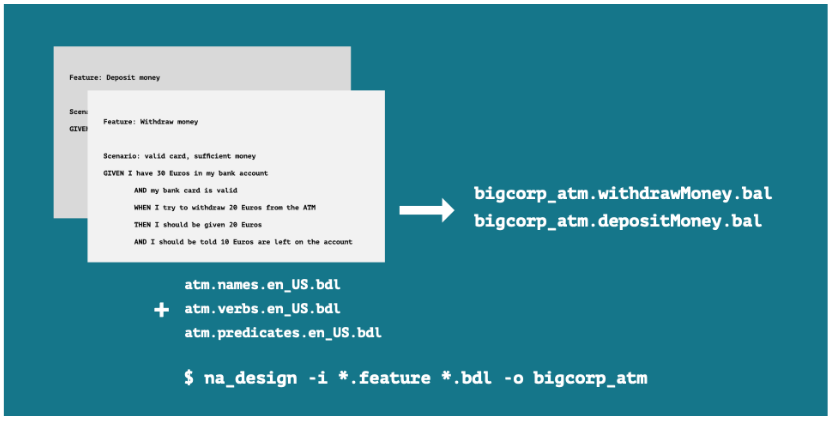
- A *business domain language (BDL) extractor*, called **NaturalAPI Discover**, will extract potential business entities (objects / names), processes (actions / verbs), and combinations thereof (predicates) from business-relevant, unstructured text documents.


wikiHow
How to Withdraw Cash from an Automated Teller Machine
Triton
ARGO
Automated Teller Machine User Manual
ATM

```
atm.names.en_US.bdl
atm.verbs.en_US.bdl
atm.predicates.en_US.bdl
```

```
$ na_discover -i argo_manual.en_US.txt wikihow_atm.en_US.txt -o atm
```

- A *feature and scenario parser*, called **NaturalAPI Design**, will create a *business application language* (BAL) API on-the-fly from Gherkin documents and an available business domain language (BDL).



```
Feature: Deposit money

Scen
GIVE
```

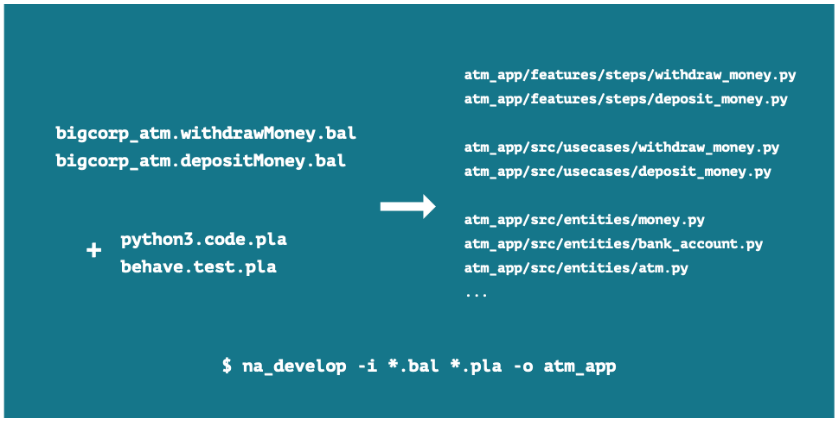
```
Feature: Withdraw money

Scenario: valid card, sufficient money
GIVEN I have 30 Euros in my bank account
        AND my bank card is valid
        WHEN I try to withdraw 20 Euros from the ATM
        THEN I should be given 20 Euros
        AND I should be told 10 Euros are left on the account
```

```
  atm.names.en_US.bdl
+ atm.verbs.en_US.bdl
  atm.predicates.en_US.bdl
```

```
bigcorp_atm.withdrawMoney.bal
bigcorp_atm.depositMoney.bal
```

```
$ na_design -i *.feature *.bdl -o bigcorp_atm
```

- A *language exporter*, called **NaturalAPI Develop**, will convert the business application language (BAL) to test cases and APIs in one of the available programming languages and frameworks, supporting both the creation of new code repositories and the update of existing ones.



```
bigcorp_atm.withdrawMoney.bal
bigcorp_atm.depositMoney.bal

+ python3.code.pla
  behave.test.pla
```

```
atm_app/features/steps/withdraw_money.py
atm_app/features/steps/deposit_money.py

atm_app/src/usecases/withdraw_money.py
atm_app/src/usecases/deposit_money.py

atm_app/src/entities/money.py
atm_app/src/entities/bank_account.py
atm_app/src/entities/atm.py
...
```

```
$ na_develop -i *.bal *.pla -o atm_app
```

NaturalAPI should take full advantage of existing open-source and commercial technologies to provide added value to its users without reinventing the wheel.

2.2) Natural Language Processing

The toolkit should leverage powerful natural language processing.
Rather than relying on regular expression or similar string parsing techniques, *NaturalAPI* should rely on advanced *natural language processing* (NLP) techniques and business domain knowledge to:

1. Find combinations of verbs and nouns (i.e. predicates) in *Gherkin* natural language specifications
2. Normalise and convert predicates into free functions or object methods with their arguments
3. Find recurring function arguments and generate corresponding objects and properties

To reach the goal, the toolkit can leverage state-of-the-art natural language tools (dependency parsers, part-of-speech taggers, stemmers, etc.) which are compatible with its license.

2.3) Code generation

***The toolkit* should generate complete APIs and automated tests.**
Thanks to its highly specialised domain-specific language, *NaturalAPI* should be able to generate complete and easily maintainable application programming interfaces, and related integration and unit tests, covering popular programming languages and frameworks.
To reach the goal, the toolkit can leverage state-of-the art API generation tools and specifications, such as the Open API specification.

3) Project overview

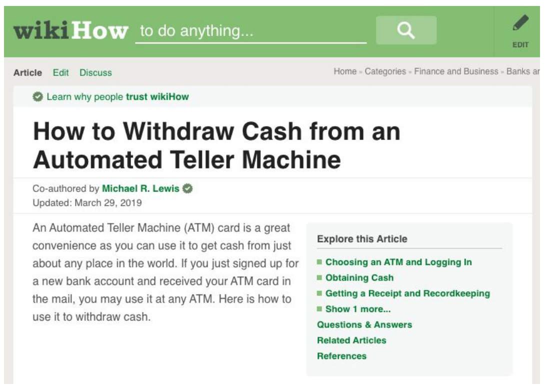
3.1) Expected workflow

***NaturalAPI* should provide a seamless workflow from specs to code.**
A typical *NaturalAPI* workflow should be as follows.

A) Project stakeholders will decide on the natural language to be used to document product features / scenarios.
Typically, but not necessarily, product features will be drafted in US English or British English.

```
Feature: Withdraw money

        Scenario: valid card, sufficient money
                GIVEN I have 30 Euros in my bank account
                AND my bank card is valid
                WHEN I try to withdraw 20 Euros from the ATM
                THEN I should be given 20 Euros
                AND I should be told 10 Euros are left on the account
```

B) A set of textual documents which are relevant to the business domain will be selected as input to *NaturalAPI Discover.*
For good results, such documents should be focussed on the tasks to be modelled. Good examples include high level project specifications, instruction manuals, *WikiHow* articles, *Wikipedia* articles, tutorials, textbooks, etc.


wikiHow to do anything...

EDIT

Article Edit Discuss

Home » Categories » Finance and Business » Banks ar

Learn why people **trust wikiHow**

How to Withdraw Cash from an Automated Teller Machine

Co-authored by **Michael R. Lewis**
Updated: March 29, 2019

An Automated Teller Machine (ATM) card is a great convenience as you can use it to get cash from just about any place in the world. If you just signed up for a new bank account and received your ATM card in the mail, you may use it at any ATM. Here is how to use it to withdraw cash.

Explore this Article
- Choosing an ATM and Logging In
- Obtaining Cash
- Getting a Receipt and Recordkeeping
- Show 1 more...

Questions & Answers
Related Articles
References

Triton®
Partnership Measured in Decades...

ARGO™
Automated Teller Machine
User Manual
TDN 07103-00339 Rev. B July 7, 2015

C) *NaturalAPI Discover* will generate a set of candidate actions (verbs), objects (nouns), and combinations thereof (predicates), automatically extracted from the documents collected in the previous step.
The output of this step will be named a *business domain language* (BDL).
If the project language is not US English, *NaturalAPI Discover* will translate the documents into US English before generating candidate verbs, nouns and predicates, as US English is the de-facto standard for writing internationally shareable code.

VERBS	freq	NOUNS	freq	PREDICATES	freq
withdraw	35	transaction	12	withdraw cash	6
select	22	cash	11	select amount	4
press	5	withdrawal	9	check balance	4
type	3	bank account	6	insert card	3
…		…		…	

D) Project stakeholders will draft a set of BDD features and scenarios to document all use cases that should be supported by the system to be created.
In doing so, they might consult the business domain language of *NaturalAPI Discover* to choose the most appropriate and consistent terminology (verbs, nouns and predicates).

```
Feature: Withdraw ~~money~~ cash


        Scenario: valid card, sufficient ~~money~~ cash
                GIVEN I have 30 Euros in my bank account
                AND my bank card is valid
                WHEN I try to withdraw 20 Euros from the ATM
                THEN I should be given 20 Euros
                AND I should be told 10 Euros are left on the account
```

E) *NaturalAPI Design* will take BDD feature descriptions, a business domain language (BDL) and an optional business ontology (BO), and generate business application language (BAL) suggestions to be interactively evaluated by API designers.
The output of this process will be named a business application language (BAL). During the interactive process, API developers will have a chance to define additional API aspects, such as: dictate required / optional input objects; merge, group or split actions and objects, etc.
Optional resources will include *business ontologies* of domain-relevant concepts, which will expand nouns into structured objects (see the `bankCard` example below).

```
withdraw(cash)
        cash - type: currency; min: 0

isValid(bankCard)
        bankCard - type: paymentMethod;
                id - type: string (required)
                expirationDate - type: date (required)

give(cash)
        cash - type: currency; minValue: 0

process(transaction)
        transaction - type: transaction
        ...
```

F) *NaturalAPI Develop* will take the business application language (BAL), together with any additional optional resources, and convert it to a common programming language / framework API.
The tool will also generate corresponding integration tests for the provided BDDs in one of the supported frameworks (e.g. *Cucumber*). It will provide an interactive session, so that programming-language specific details not already specified in the optional resources can

be defined by the API developer. These programming-specific aspects will be stored in a *programming language adapter* (PLA).

The toolsuite will be designed so that changes in any of the resources (input documents, BDD features, BDLs, BALs and PLAs) can be easily propagated downstream.

4) Product specification

4.1) Features

4.1.1) Mandatory requirements

Notice: scenarios for the following features are just a starting point, and might be refined and adjusted by project stakeholders in the initial project phase, and also later if required. The scenarios can optionally be used to perform automated testing of *NaturalAPI* features.

4.1.1.1) *NaturalAPI Discover*

Extract_bdl_from_documents.feature

As an application spec writer, I want to extract a business domain language (BDL) from a set of documents, so that I can write better features/scenarios and create a business application language (BAL).

```
Scenario: one or more documents - en-US
  GIVEN there are one or more documents containing potential BDL candidates
  AND all the documents are written in en-US
  WHEN I extract the BDL from the documents
  THEN I am given a list of business-domain related nouns, verbs, and predicates
  AND I am given the frequency across all documents of each noun, verb and predicate
  AND I am not given very common and non business-domain related nouns, verbs, predicates
```

4.1.1.2) *NaturalAPI Design*

Generate_bal_from_bdl_and_bo.feature

As an API designer, I want to interactively generate a business application language (BAL) from a business domain language (BDL) and an optional business ontology (BO), so that I can write an API which closely reflects the natural language specification (features/scenarios) and the business domain.

```
Scenario: no business ontology
  GIVEN there is a business domain language
  WHEN I generate a business application language from the business domain language
  THEN I am asked to confirm each business application language suggestion
  AND finally I am given a business application language with all suggestions I confirmed

Scenario: business ontology
  GIVEN there is a business domain language
  AND there is a business ontology
  WHEN I generate a business application language from the business domain language and the
business ontology
  THEN I am asked to confirm each business application language suggestion
  AND I am given the option to substitute a suggested object with a related ontology object
  AND finally I am given a business application language with all suggestions I confirmed
```

4.1.2.3) *NaturalAPI Develop*

Generate_api_from_bal_and_pla.feature

As an API designer/developer, I want to interactively generate a programming language API from a business application language and an optional programming language adapter, so that I can implement the application spec in the most convenient language / framework with little effort.

Notice: the following programming language / test framework combination is just an example

```
Scenario: Python / behave - no programming language adapter
  GIVEN there is a business application language
  AND there is a set of related BDD feature scenarios
  WHEN I generate an API from the business application language
  THEN I am asked to confirm each API suggestion (method names, argument types)
  AND finally I am given an API in the “Python” language with all suggestions I confirmed
  AND finally I am given a test API in the “behave” framework for the given feature scenarios
```

4.1.2) Optional requirements

Optional requirements might include e.g. adding support for writing specifications in a language different from en-US; creating workflows to connect the three steps (discover-design-develop) together; etc.

4.2) Delivery modes

4.2.1) Mandatory requirements

Each *NaturalAPI* feature shall be accessible through **at least two modes** among the following:

- a Command Line Interface
- a minimal Graphical User Interface
- a web REST interface

All delivery modes should share the same logic layers (see next section). The delivery modes can be implemented in any programming language/framework of choice. The delivery modes shall be accessible through **at least one popular desktop platform** (e.g. Ubuntu Linux, macOS, Windows, or in-browser).

4.2.2) Optional requirements

Optional requirements might include e.g. adding a third delivery mode to one or more features; creating a richer Graphical User Interface, etc.

4.3) Application architecture and code quality

4.3.1) Mandatory requirements

The logic layers should be deployed in one of the following ways:

- as a library (static or dynamic)
- as part of the same executable as the chosen delivery modes
- as an independent process/service, locally or remotely

In any case, logic layers shall have no dependency on any specific delivery mode.
For the implementation of the logic layers, a *Clean Architecture* approach is suggested, but any structured and motivated approach to modern software architecture is welcome.
The development team is encouraged to practice The Joel Test.

4.4) Input and output

4.4.1) Mandatory requirements

Notice: all textual input/output resources should have UTF-8 encoding and Unix line breaks.

4.4.1.1) NaturalAPI Discover

```
Input
        documents
                Format: plain text
                file pattern: [documentName].en_US.txt
Output
        list of lemmatized verbs and frequencies
                format: CSV
                file name pattern: [projectName].names.en_US.bdl
        list of lemmatized nouns and frequencies
                format: CSV
                file name pattern: [projectName].verbs.en_US.bdl
        list of lemmatized predicates and frequencies
                format: CSV
                file name pattern: [projectName].predicates.en_US.bdl
```

4.4.1.2) NaturalAPI Design

```
Input
        feature scenarios
                format: gherkin .feature
        list of lemmatized verbs and frequencies
        list of lemmatized nouns and frequencies
        list of lemmatized predicates and frequencies
        ontology
                format: OWL v2
Output
        business application language
                Format: OpenAPI v3 JSON Objects (esp. OperationObject, ParameterObject)
                File name pattern: [projectName].[operationId].bal
```

4.4.1.3) NaturalAPI Develop

```
Input
        business application language
        programming language adapter
Output
        language/framework-specific API
        language/framework-specific BDD test entry points
```

5) Intellectual property rights

The proposing company's aim for this project is evaluating the proof of concept's (PoC) feasibility. All the code and artifacts will remain property of the development team (the students). The development team will grant the proposing company a perpetual, free of charge license to use the PoC's artifacts and to inspect the related source code.
The proposing company encourages the development team to release the PoC under a permissive Open Source license (e.g. MIT, Apache) and to publish the code on an easily accessible public repository (e.g. on github.com).

6) Helpful resources and definitions

6.1) Natural language processing

Natural language processing -
https://en.wikipedia.org/wiki/Natural_language_processing
An overview of NLP's subdisciplines and tasks.

Dependency parsing - Stanford parser - https://nlp.stanford.edu/software/nndep.html
Brief definition of dependency parsing, state-of-the art parser.

6.2) Behavior-driven development

Introducing BDD - https://dannorth.net/introducing-bdd/
A brief intro to BDD by its initiator.

Gherkin - https://cucumber.io/docs/gherkin/
The de facto standard to write feature specifications in natural language.

Hiptest - https://hiptest.com/
An online tool for writing BDD features with code generation capabilities.

Cucumber - https://cucumber.io/docs
A framework for automated BDD testing.

6.3) API and DLS generation

OpenAPI Specification - https://github.com/OAI/OpenAPI-Specification
A recognized standard to write API specifications.

Swagger - https://swagger.io/
Code generation tools based on OpenAPI.

OWL v2 ontologies - https://www.w3.org/OWL/
W3C format for ontology representation.

6.4) Application architecture & code

Clean Architecture -
https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html
A principled way to application architecture.

The Joel test -
https://www.joelonsoftware.com/2000/08/09/the-joel-test-12-steps-to-better-code/
Useful checklist for more effective software projects.

7) About teal.blue

teal.blue
digital solutions

teal.blue, a Maply SIT brand, is a small digital solutions provider based in the Bergamo area, with headquarters at POINT (Polo per l'innovazione scientifica e tecnologica della provincia di Bergamo) in Dalmine. Its customers include businesses from the industrial, entertainment and medical sectors.

teal.blue's core competence is the development of cross-platform applications based on the Qt framework with rich user interfaces for desktop, mobile and embedded systems.