Make your database code sing!

How to increase your coding productivity 10X or more

Jim Nasby - Enova Financial

The Problem

Procedural Languages have improved vastly since the 1970s

Procedural Improvements

Procedural Improvements

- First there were libraries and `#include`

Procedural Improvements

- First there were libraries and `#include`
- That evolved to Object Oriented Programming, which led to...

Procedural Improvements

- First there were libraries and `#include`
- That evolved to Object Oriented Programming, which led to...
- code that is easy to factor, which means

Procedural Improvements

- First there were libraries and `#include`
- That evolved to Object Oriented Programming, which led to...
- code that is easy to factor, which means
- code re-use is easy!

Why is code re-use important?

“Society advances by increasing the complexity of what people can do without thinking”

In today’s world...

In today's world...

- Your car starts when you turn the key (no messing with mixture, ignition timing, etc)

In today's world...

- Your car starts when you turn the key (no messing with mixture, ignition timing, etc)
- You throw the clothes in the washing machine

In today’s world...

- Your car starts when you turn the key (no messing with mixture, ignition timing, etc)
- You throw the clothes in the washing machine
- You don’t worry about getting across the country, you worry about getting to the airport

Code re-use allows you to do more complex things without thinking

What’s improved with database coding since 1970?

• • •

Not much!

One of the most used tools for database coding is still

CUT, PASTE and REPLACE!

Much database development is done by pasting the same code over and over because we lack things like classes

Ex: Lookup table

```
CREATE TABLE customer_status(
  customer_status_id int PK
  , customer_status text
UNIQUE
);
```

Process

Process

- Find another place where a lookup table was created

Process

- Find another place where a lookup table was created
- Copy and Paste it

Process

- Find another place where a lookup table was created
- Copy and Paste it
- Replace “customer” with something new

Problems

Problems

- Tedious

Problems

- Tedious
- Time consuming

Problems

- Tedious
- Time consuming
- Error prone

The problems get worse as complexity increases

Actual lookup table

Actual lookup table

- Table

Actual lookup table

- Table
- Permissions

Actual lookup table

- Table
- Permissions
- __get_id(), __get_text(), __get()

Actual lookup table

- Table
- Permissions
- __get_id(), __get_text(), __get()
- Unit tests

When dealing with real-world code duplication, it becomes almost impossible not to mess it up

It's also not possible to add a new feature to ALL your duplicated code without a lot of extra work

How do we change this?

Real change here would require serious changes to our RDBMS... like adding support for classes

... but I'm NOT PATIENT!

... but I’m NOT PATIENT!

So let’s see what we can do with what we already have.

Our weapons!

- Helper functions
- Meta-programming
- Breaking one database into components
- Data inheritance

Helper functions

Don’t cut and paste - Create functions!

Helper functions

Helper functions

- array_length
- is_empty_or_null
- parameter_replace
- string_or_array
- table_full_name
- table_schema_and_name

Just don’t repeat yourself!

Metacode

Computers are really good at repetitive tasks...

... so let’s make them write code for us!

Metacode Goals

Metacode Goals

- Make it EASY to create new database objects

Metacode Goals

- Make it EASY to create new database objects
- Allow us to TRACK objects that we have created

Metacode Goals

- Make it EASY to create new database objects
- Allow us to TRACK objects that we have created
- Enable MODIFYING objects that have been created

Goal: Easy to create

Allow a single function call to create a number of objects for us

code.lookup_static()

code.lookup_static()

- Create a lookup table to normalize a text field, ie: a status code

code.lookup_static()

- Create a lookup table to normalize a text field, ie: a status code
- Create all our indexes

code.lookup_static()

- Create a lookup table to normalize a text field, ie: a status code
- Create all our indexes
- Assign permissions on the table

code.lookup_static()

- Create a lookup table to normalize a text field, ie: a status code
- Create all our indexes
- Assign permissions on the table
- Call other metacode functions that create “__get()” functions for our new lookup table

Goal: Easy to create

Have a single function call handle ALL the details for an object

code.function()

code.function()

- Create a database function

code.function()

- Create a database function
- Make it easy to set custom function permissions

code.function()

- Create a database function
- Make it easy to set custom function permissions
- Make it easy to add a comment to the function

code.function()

Metacode makes this EASY by removing the need to cut and paste the function parameters over and over.

Goal: Easy to Track

Allow for tracking of objects created by metacode

Tracking

Tracking

- Tracked objects are built from *templates*

Tracking

- Tracked objects are built from *templates*
- A *template* contains %TAGS% that are replaced to give us our final SQL that creates objects

code.lookup_static()

code.lookup_static()

```
SELECT code.lookup_static( ‘loan_status’ );
```

code.lookup_static()

```
SELECT code.lookup_static( ‘loan_status’ );
```

Uses the template:

```
CREATE TABLE %status_name% (
%status_name%_id smallint PRIMARY KEY
, %status_name% citext UNIQUE);
```

code.lookup_static()

```
SELECT code.lookup_static( ‘loan_status’ );
```

Uses the template:

```
CREATE TABLE %status_name% (
%status_name%_id smallint PRIMARY KEY
, %status_name% citext UNIQUE);
```

Which gives us this SQL:

```
CREATE TABLE loan_status (
loan_status_id smallint PRIMARY KEY
, loan_status citext UNIQUE);
```

Tracking

Tracking

- The metacode system stores templates

Tracking

- The metacode system stores templates
- When you use a template to create something, the system remembers the template and parameters that you used

Tracking

- The metacode system stores templates
- When you use a template to create something, the system remembers the template and parameters that you used
- This way, you can always see what database objects have been created by metacode

Tracking

- The metacode system stores templates
- When you use a template to create something, the system remembers the template and parameters that you used
- This way, you can always see what database objects have been created by metacode
- Tracking is optional (ie: code.function())

Goal: Allow for Modifying

Because everything can be tracked, it can also be modified

Modifying

Modifying

- All templates are versioned

Modifying

- All templates are versioned
- Template versions store *upgrade templates*

Modifying

- All templates are versioned
- Template versions store *upgrade templates*
- *Upgrade templates* allow upgrading existing metacode objects (ie: loan_status) to a newer version

Modifying

- All templates are versioned
- Template versions store *upgrade templates*
- *Upgrade templates* allow upgrading existing metacode objects (ie: loan_status) to a newer version
- Templates also tell us how to drop objects

Our weapons!

- Helper functions
- Meta-programming
- Breaking one database into components
- Data inheritance

Components

Make it easy to re-use large amounts of code in different databases

Components are *libraries* of database code that are used in multiple databases

#include

Components

Components

- A component is comprised of a number of database schemas and all the objects in those schemas

Components

- A component is comprised of a number of database schemas and all the objects in those schemas
- Each component has a set of specific roles for object ownership and permissions

Components

- A component is comprised of a number of database schemas and all the objects in those schemas
- Each component has a set of specific roles for object ownership and permissions
- All code and unit tests for a component are kept together, and separated from other components

Component Examples

Component Examples

- Your *helper functions* and other tools will work in ALL your databases... so make them a component!

Component Examples

- Your *helper functions* and other tools will work in ALL your databases... so make them a component!
- Basic tracking of personal information (name, addresses, phone numbers)

Component Examples

- Your *helper functions* and other tools will work in ALL your databases... so make them a component!
- Basic tracking of personal information (name, addresses, phone numbers)
- Accounting / General ledger

Data Inheritance

Re-use your code AND your data

Table inheritance

Table inheritance

- Feature built-in to Postgres

Table inheritance

- Feature built-in to Postgres
- A child table inherits it's definition from one or more parent tables

Table inheritance

- Feature built-in to Postgres
- A child table inherits it's definition from one or more parent tables
- A child can add it's own unique definition

Table inheritance

- Feature built-in to Postgres
- A child table inherits it's definition from one or more parent tables
- A child can add it's own unique definition
- By default, data in child tables will show up when you query a parent

Inheritance Example

Inheritance Example

- Customers have different ways to pay (bank account, debit card, Paypal, etc)

Inheritance Example

- Customers have different ways to pay (bank account, debit card, Paypal, etc)
- Some fields are common to all methods

Inheritance Example

- Customers have different ways to pay (bank account, debit card, Paypal, etc)
- Some fields are common to all methods
- Parent table: payment_instrument(
payment_instrument_id
, customer_id
, payment_instrument_type_id);

Inheritance Example

Inheritance Example

- Child table: bank_account(
  routing_number
  , account_number)
  INHERITS(payment_instrument)

Inheritance Example

- Child table: bank_account(
  routing_number
  , account_number)
  INHERITS(payment_instrument)

- Child table: debit_card(
  card_token
  , expiration_date)
  INHERITS(payment_instrument)

Inheritance Downsides

Inheritance Downsides

- Some things (ie: indexes) do not inherit

Inheritance Downsides

- Some things (ie: indexes) do not inherit
- Sometimes you want something inherited by only certain tables

Inheritance Downsides

- Some things (ie: indexes) do not inherit
- Sometimes you want something inherited by only certain tables
- No cross-table unique indexes

Inheritance Downsides

- Some things (ie: indexes) do not inherit
- Sometimes you want something inherited by only certain tables
- No cross-table unique indexes
- No foreign keys referring to parent table

Metacode to the rescue!

Inheritance Metacode

Inheritance Metacode

- Allows defining things that you want added to all (or most) child tables of a parent

Inheritance Metacode

- Allows defining things that you want added to all (or most) child tables of a parent
- Uses %tag% replacement

Our weapons!

- Helper functions
- Meta-programming
- Breaking one database into components
- Data inheritance

Ask yourself: “What am I repeating over and over?”

Case-study: lookup tables

- Table, permissions
- Marked as seed data
- code.get, code.get_id, code.get_text
- All of this is unit tested

Case-study: lookup tables

Case-study: lookup tables

- Framework development: ~24 hours

Case-study: lookup tables

- Framework development: ~24 hours
- Development of code.lookup_table_static and 3 other metacode functions: ~16 hours

Case-study: lookup tables

- Framework development: ~24 hours
- Development of code.lookup_table_static and 3 other metacode functions: ~16 hours
- 97 uses (and growing)

Case-study: lookup tables

- Framework development: ~24 hours
- Development of code.lookup_table_static and 3 other metacode functions: ~16 hours
- 97 uses (and growing)
- Minimum 15 minutes for cut and paste x 97 uses = 24 hours

Case-study: lookup tables

Case-study: lookup tables

- Development of code.lookup_table_dynamic: ~8 hours

Case-study: lookup tables

- Development of code.lookup_table_dynamic: ~8 hours
- 17 uses (and growing)

Case-study: lookup tables

- Development of code.lookup_table_dynamic: ~8 hours
- 17 uses (and growing)
- Minimum 30 minutes for cut and paste x 17 uses = 8.5 hours

The real difference

The real difference

- Say you get REALLY good at cut and paste

The real difference

- Say you get REALLY good at cut and paste
- Down to 5 minutes!

The real difference

- Say you get REALLY good at cut and paste
- Down to 5 minutes!
- How long does it take to type

  ```
  SELECT code.lookup_table_static
  ( ‘cnu’, ‘loan_statuses’,
  ‘loan_status’ );
  ```

The real difference

- Say you get REALLY good at cut and paste
- Down to 5 minutes!
- How long does it take to type

  ```
  SELECT code.lookup_table_static
  ( ‘cnu’, ‘loan_statuses’,
  ‘loan_status’ );
  ```

- 16 seconds - 19x faster!

The real difference

The real difference

- How long does it take to type
  `SELECT code.lookup_view ( ‘loans’ );`

The real difference

- How long does it take to type

  ```
  SELECT code.lookup_view
  ( ‘loans’ );
  ```

- 8 seconds

The real difference

- How long does it take to type

  ```
  SELECT code.lookup_view
  ( ‘loans’ );
  ```

- 8 seconds
- Now you have a denormalized view on that table, and you CAN NOT cut and paste that!

Ask yourself:“What am I repeating over and over?”

Use our weapons to work smarter

Use our weapons to work smarter

... and give us more time at the bar!

“Wow, that’s awesome Jim! Where can I get all this cool stuff?!!”

http://pgfoundry.org/projects/enova-tools/

http://pgfoundry.org/projects/enova-tools/

Questions?

http://meetup.com/Chicago-PostgreSQL-User-Group/