codeborne
well-crafted software

Database Refactoring

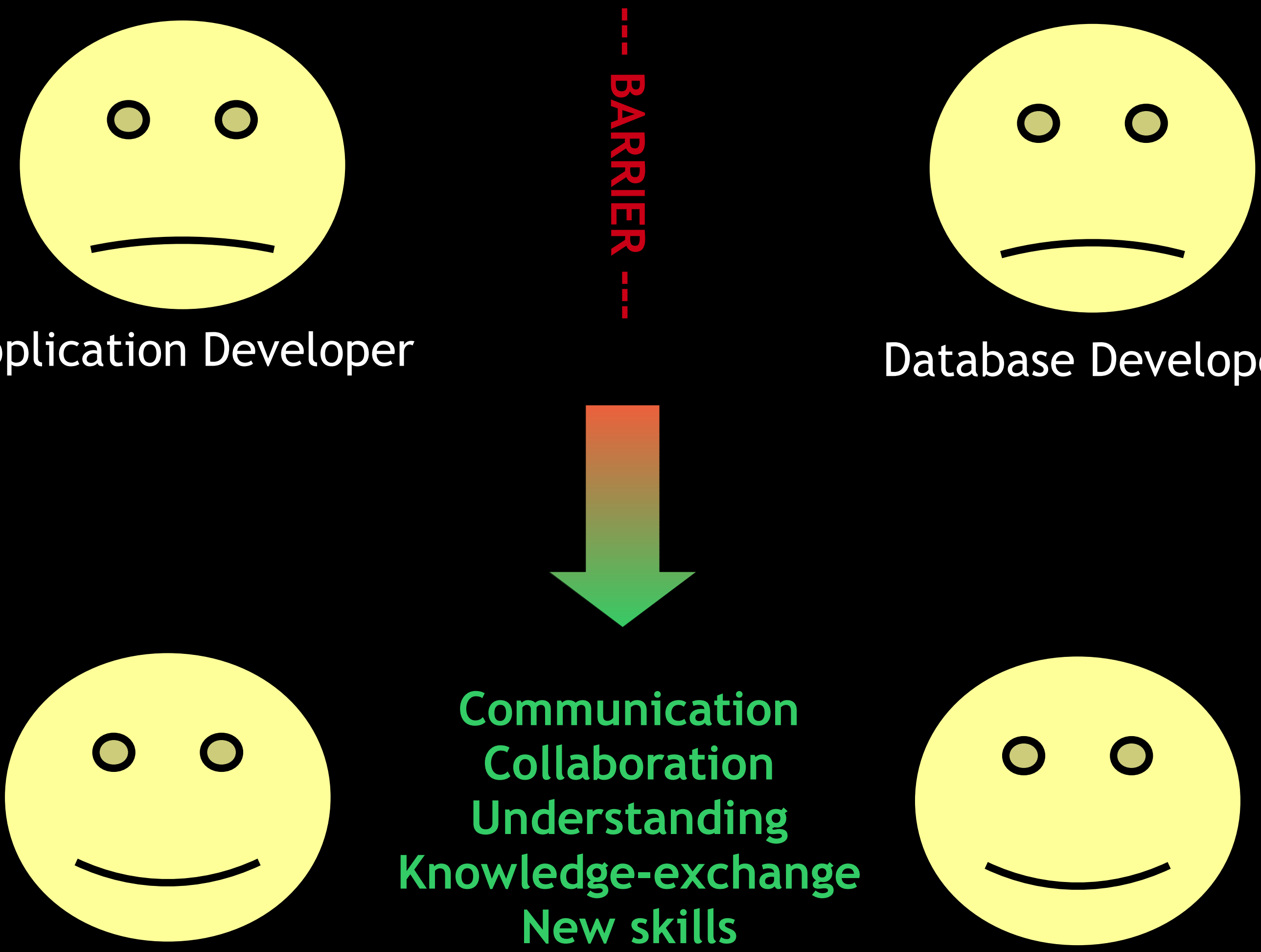
keeping up with evolution

Devclub.eu
25.03.2011

Anton Keks

A few words of warning…

Avoid overspecialization



Application Developer

--- BARRIER ---

Database Developer

Communication
Collaboration
Understanding
Knowledge-exchange
New skills

Developer

Developer

Refactoring

In Maths, to “factor” is to reduce an expression to it's simplest form

In CS, is the disciplined way to restructure code

- Without adding new features
- Improving the design
- Often making the code simpler, more readable

Definition: Code Refactoring

- A small change to the code to improve design that retains the behavioural semantics of the code
- Code refactoring allows you to evolve the code slowly over time, to take evolutionary approach to programming

Definition: Database Refactoring

- A simple change to a schema that improves its design while retaining *behavioural* and *informational* semantics
- A database includes both structural aspects as well as functional aspects

It's

Refactoring

not

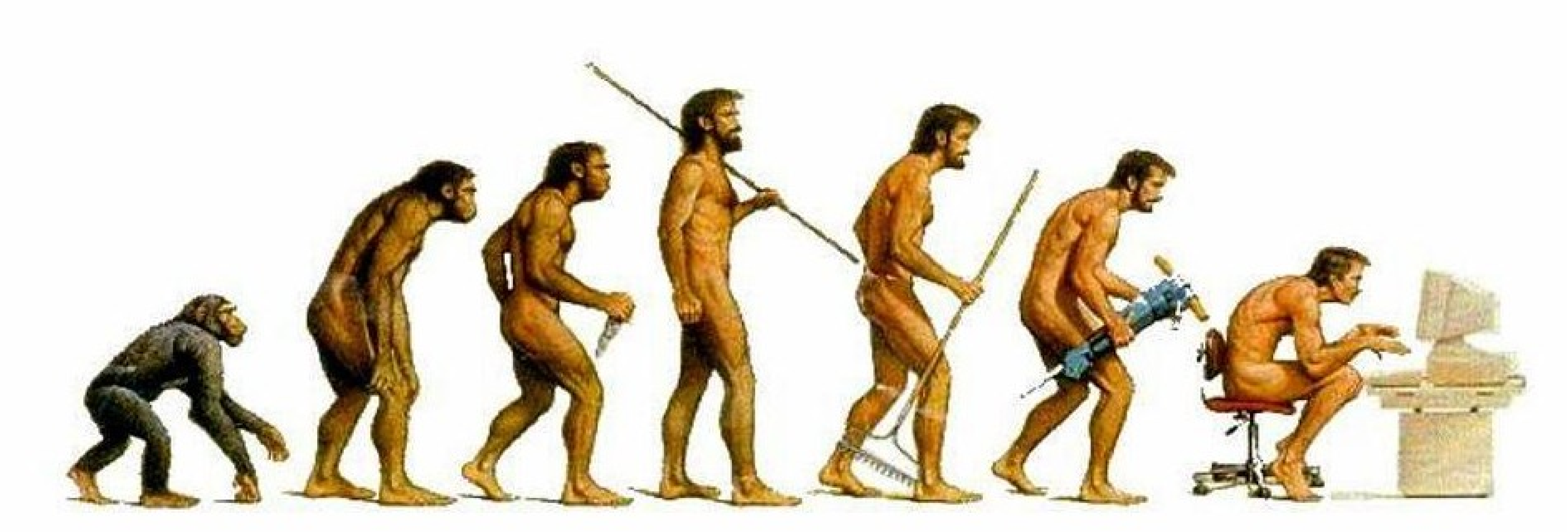
Refucktoring

Why refactor?

- To safely fix existing legacy databases
  - They are here to stay
  - They are not going to fix themselves!
- To support evolutionary development
  - Because our business, our customers are changing
  - The world around our software is evolving
- Prevent over-design
  - Simple, maintainable code and data model

Leads to Evolutionary Design

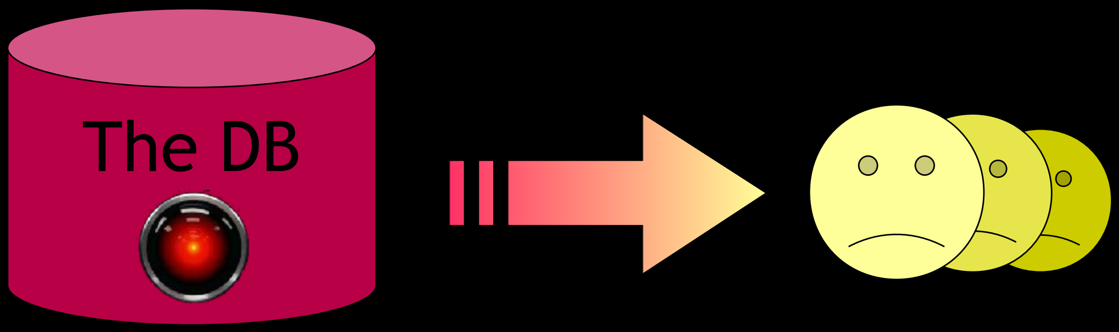
- Small steps
- The simplest design first
- Unit tests of stored code (or avoid it!)
- Design is final only when the code is ready



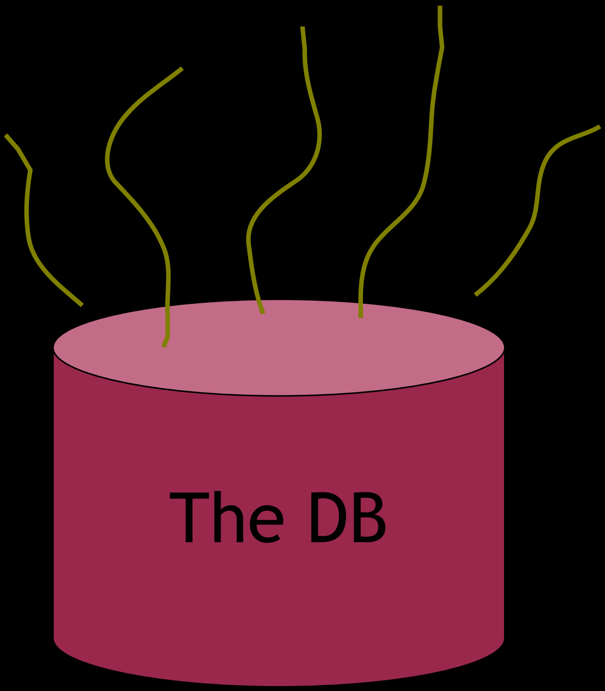
Otherwise, it can happen that

Database is not under control

It lives its own life and is controlling us



Database Smells



Database Smells

- All known code smells also apply to stored code as well, including:
  - Monster procedures
  - Spaghetti code
  - Code duplication
  - IF-ELSE overuse
  - Code ladder
  - Low cohesion
  - etc

Database Smells

- Database schema can add to the musty odour
  - Multi-purpose table / column
  - Redundant data
  - Tables with many columns / rows
  - “Smart” columns
  - Lack of constraints
  - Fear of change

Fear of change

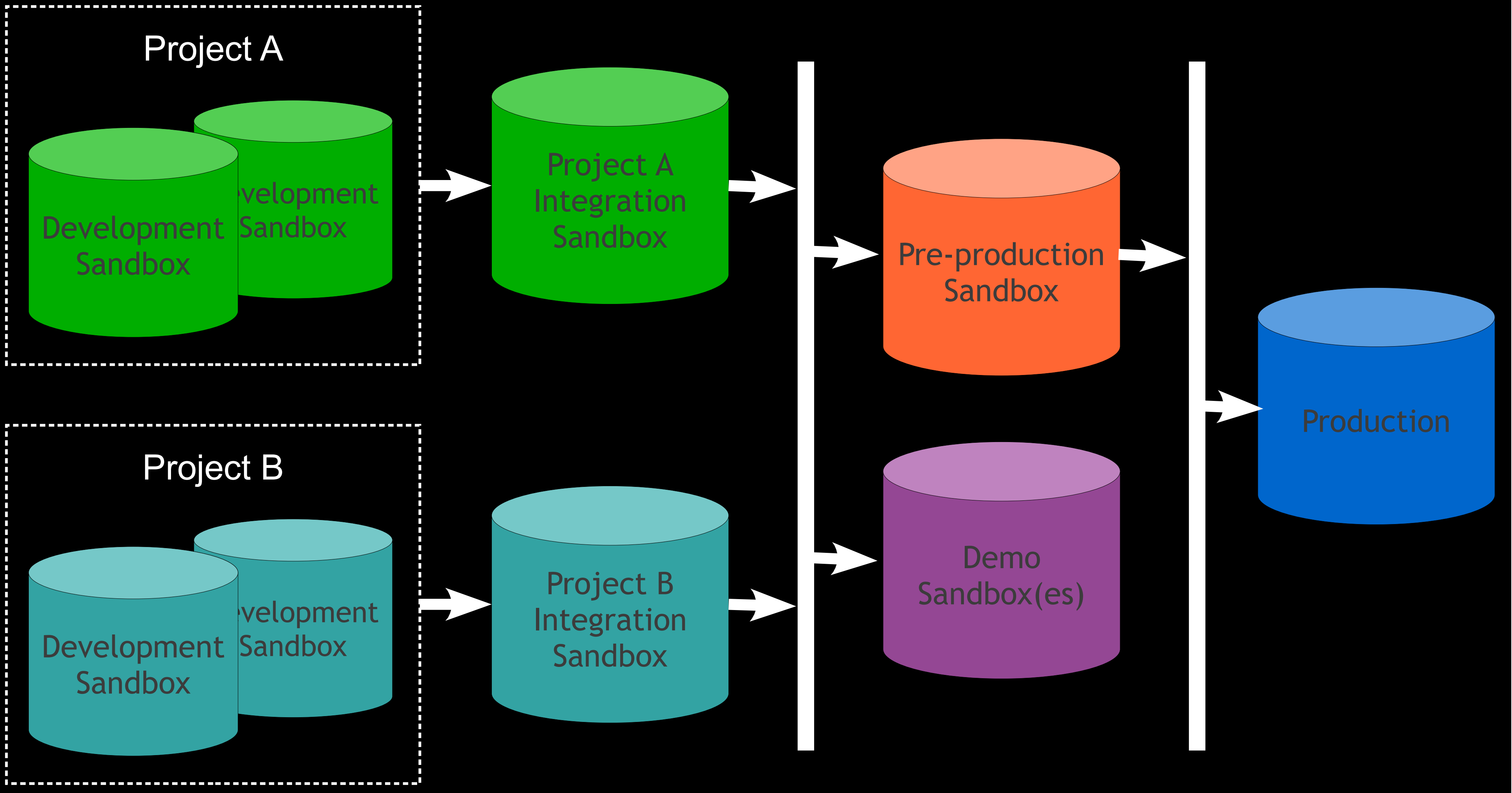
- The strongest of all smells
- Prevents innovation
- Reduces effectiveness
- Produces even more mess
- Over time, situation gets only worse

How to do it right?

- Start in your development sandbox
- Apply to the integration sandbox(es)
- Install into production

“Keep out of **my** unstable development DB!”

Sandboxes (1)

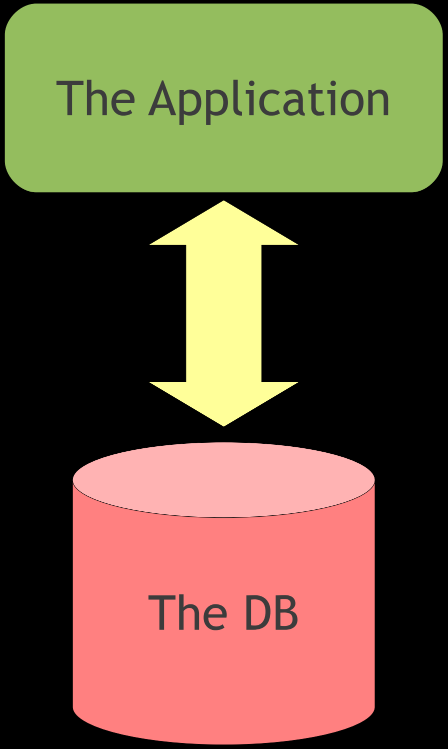


frequent deployment

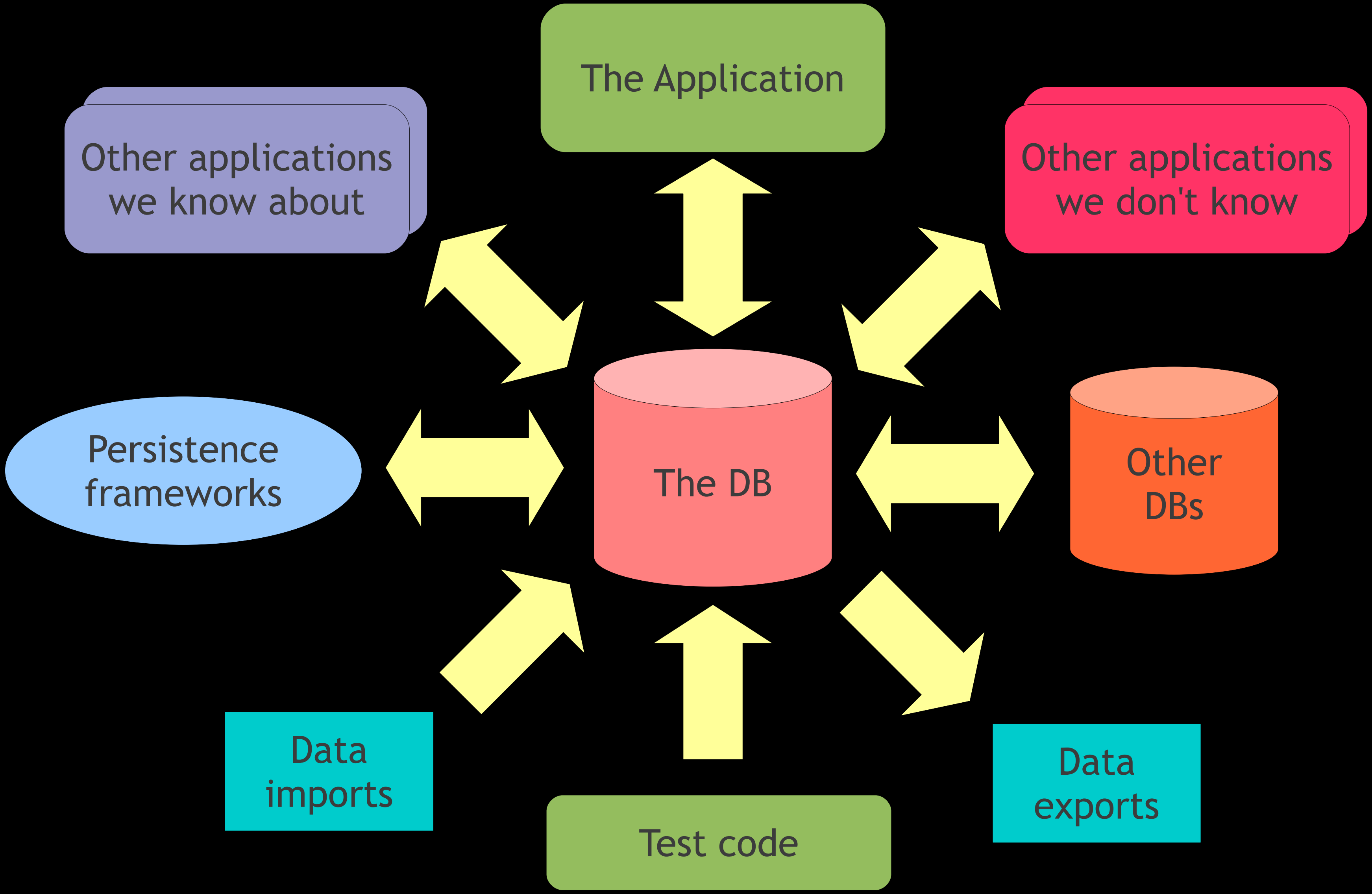
controlled deployment

highly controlled deployment

Best case scenario (easiest)



Worst case scenario (hardest)



Trivial case

- Can we rename a column in our DB?
  - Without breaking 100 applications?
- If we can't do something trivial, how can we do something important?
  - If we can't evolve the schema, we are most likely not very good at developing applications

Testing (2)

- Do we have code in the DB that implements critical business functionality?
- Do we consider data an important asset?
- … and it's all not tested?

- Automatic regression tests would help
- Proper refactoring cannot happen without them

Database Unit Tests

- Too complex?
- No good framework?

```
create or replace package dbunit
is
    procedure assert_equals(expected number, actual number);
    procedure assert_equals(expected varchar2, actual varchar2);
    procedure assert_null(actual varchar2);
    procedure assert_not_null(actual varchar2);
    ...
end;

create or replace public synonym dbunit for dbunit;
grant execute on dbunit to public;
```

Running Unit Tests

- Anonymous PL/SQL code
- No need to change the DB
- Assertions *raise_application_error* with specific message if tests fail
- *Rollback* at the end
- Runnable with any SQL tool
- Or with **ant**

PL/SQL Unit Test example

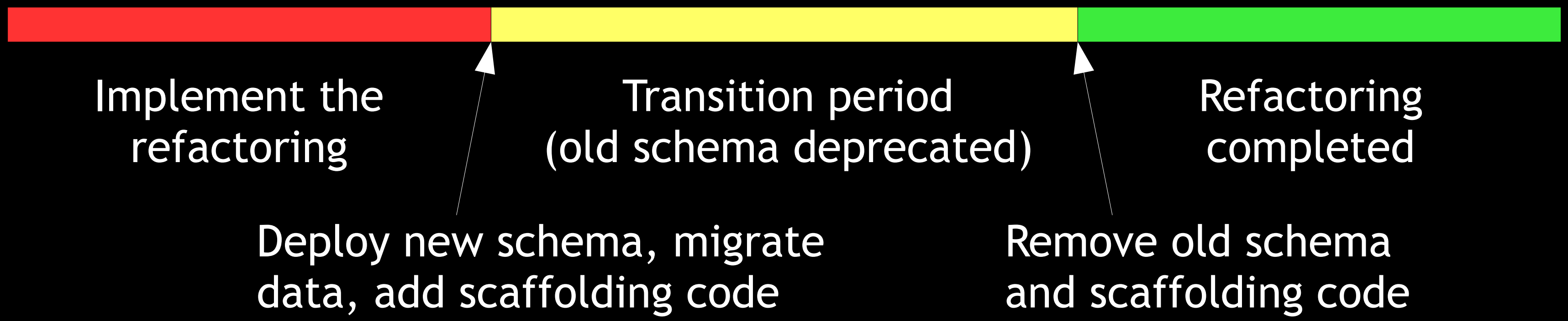
```
declare
   xml XmlType;
begin
--@Test no messages in case of no changes
   xml := hub.next_message(0);
   dbunit.assert_null(xml);

--@Test identification number change message
   hub_api.ident_number_changed('123', '007', 'PERSONAL_CODE',
                                'LV', '888', current_timestamp);
   xml := hub.next_message(1);
   dbunit.assert_xpath('123', '/hub/party/@source_ref', xml);
end;
```

How to deal with coupling?

- Big-Bang approach
  - Usually, you can't fix all 100 apps at once
- Give up
  - And afford even more technical debt?
- Transition Window approach
  - Can be a viable solution

Transition Window (3)

- Deprecate the old schema
  - Write tests if not present
  - Decide on the removal date, communicate it out
- Create the change
  - Make the old schema work (scaffolding code)
- Run the tests



Implement the refactoring

Transition period (old schema deprecated)

Refactoring completed

Deploy new schema, migrate data, add scaffolding code

Remove old schema and scaffolding code

Dealing with unknown applications

- It's easy to eliminate all usages in
  - the DB itself
  - the application you are developing
- Log accesses to the deprecated schema
  - Helps to find these 'unknown' applications

Changelog (4)

- Doing all this needs proper tracking of changes
- Write *delta-scripts* *(aka migrations)*
    - To start the transition period
    - To end the transition period (these will be applied on a later date/release)
- Same scripts for
    - Updating sandboxes
    - Deployment to production

What to refactor in a DB?

- Databases usually contain
  - **Data** (stored according to a **schema**)
  - Stored **code**
- Stored code is no different from any other code
  - except that it runs inside of a database
- Database schema
  - Data is the **state** of a database
  - Maintaining the state needs a different approach from refactoring the code

Upgrade/Downgrade Tool

- Upgrade tool will track/update the changelog table automatically
  - Each DB will know it's state (version)
  - It will be easy to upgrade any sandbox
- Downgrading possibility is also important
  - Delta scripts need to be two-way, i.e. include undo statements
  - It will be easy to switch to any other state
    - e.g. in order to reproduce a production bug

Sample refactoring script

```
-- rename KLK to CUSTOMER_ID
ALTER TABLE CUSTOMER ADD COLUMN CUSTOMER_ID NUMBER;
UPDATE CUSTOMER SET CUSTOMER_ID = KLK;
-- keep KLK and CUSTOMER_ID in sync
CREATE TRIGGER ...;

--//@UNDO
DROP TRIGGER ...;
ALTER TABLE CUSTOMER DROP COLUMN CUSTOMER_ID;

-- this will go to another script for later deployment
-- finish rename column refactoring
DROP TRIGGER ...;
ALTER TABLE CUSTOMER DROP COLUMN KLK;
--//@UNDO
...
```

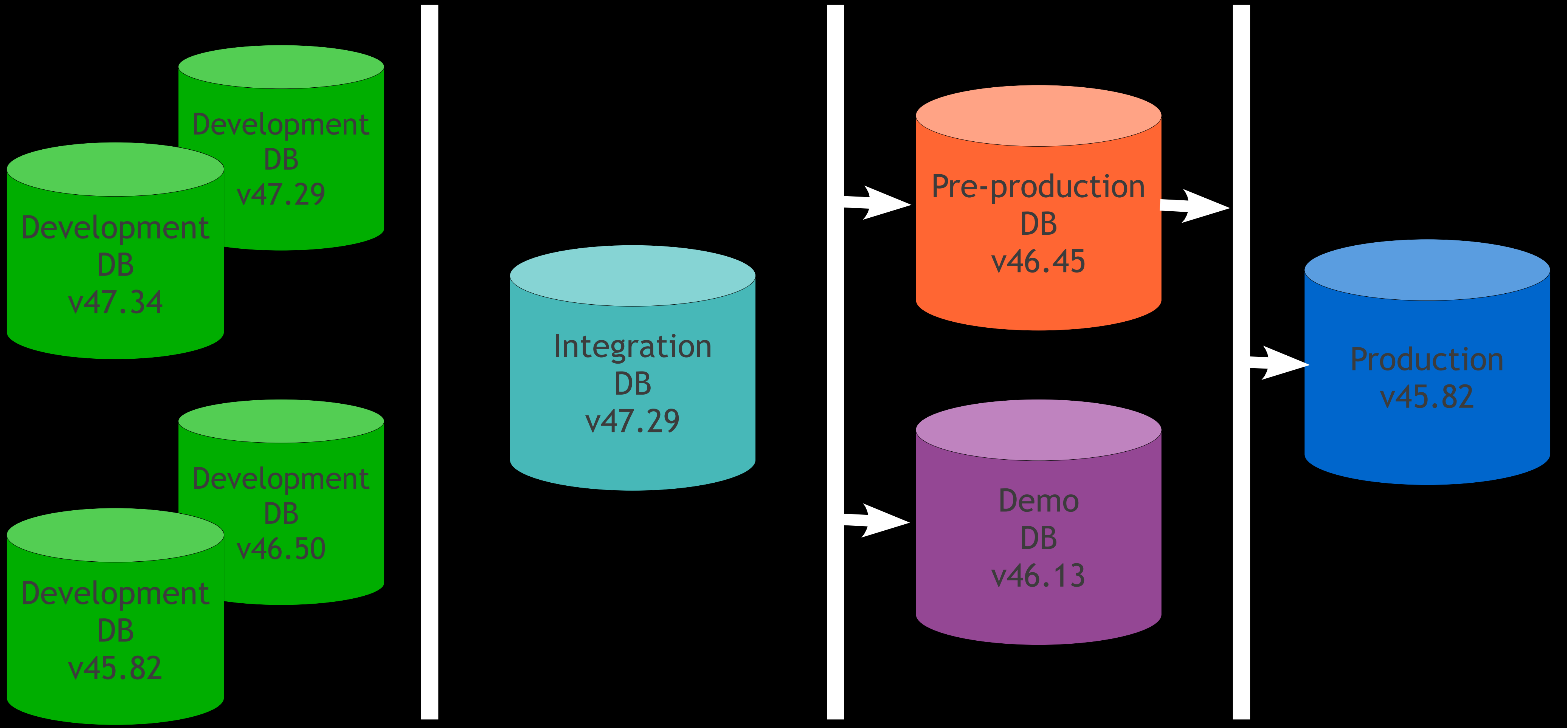
dbdeploy

- http://dbdeploy.com
- Very simple
- Runnable from ant or command-line
- Delta scripts
  - Numbered standard .sql files
  - Unapplied yet delta scripts run sequentially
  - Nothing is done if the DB is already up-to-date

liquibase

- http://liquibase.org
- More features, more complex
- Runnable from ant or command-line
- Delta scripts
  - In XML format (either custom tags or inline SQL)
  - Many changes per file
  - Identifies changes with Change ID, Author, File
  - Records MD5 for detecting of changed scripts

Versioning



Each DB knows its release/version number and can be upgraded/downgraded to any other state

Proper Versioning

- Baseline (aka skin)
    - Delta scripts (migrations)
    - Code changes
- Branch for a release
- New baseline after going to production
- The goal of versioning a database is to push out changes in a consistent, controlled manner and to build reproducible software

Continuous Integration (5)

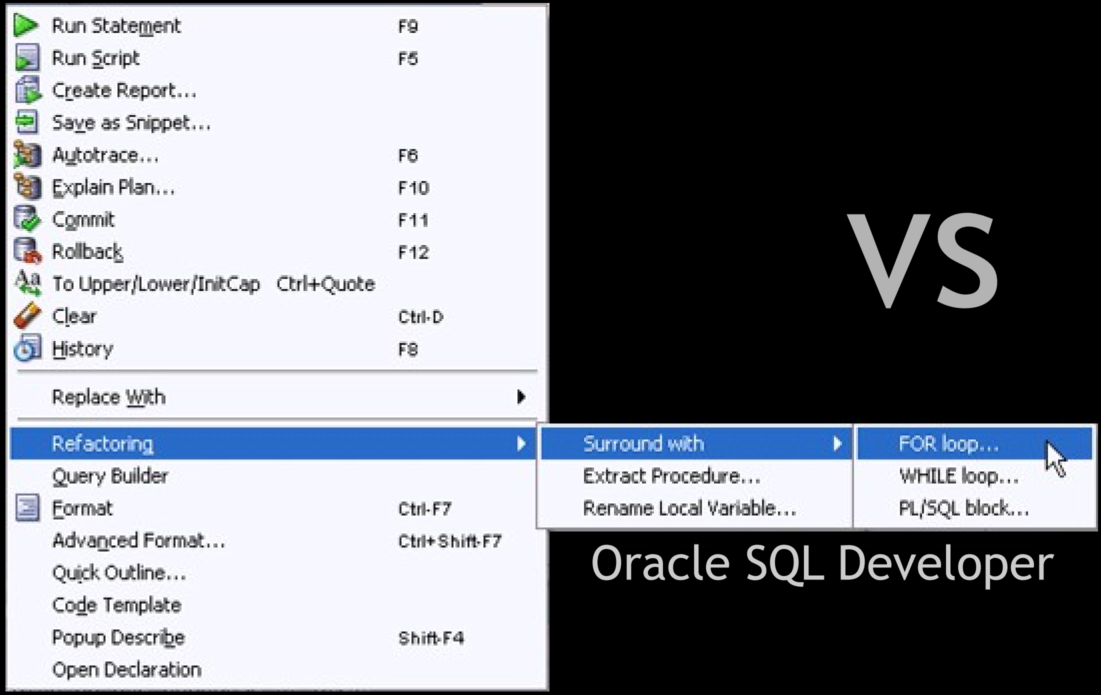
- CI server will verify each commit to the VCS
  - By deploying it into an integration sandbox
  - And running regression tests
  - Fully automatically
- All the usual benefits
  - Better quality, Quick feedback
  - Build is always ready and deployable
  - Developers are independent
  - No locking, no overwriting changes!!!

Teamwork (6)

- Developers
    - Must work closely with Agile DBAs
    - Must gain basic data skills
- Agile DBAs/DB developers
    - Must be embedded into the development team
    - Must gain basic application skills

Tools

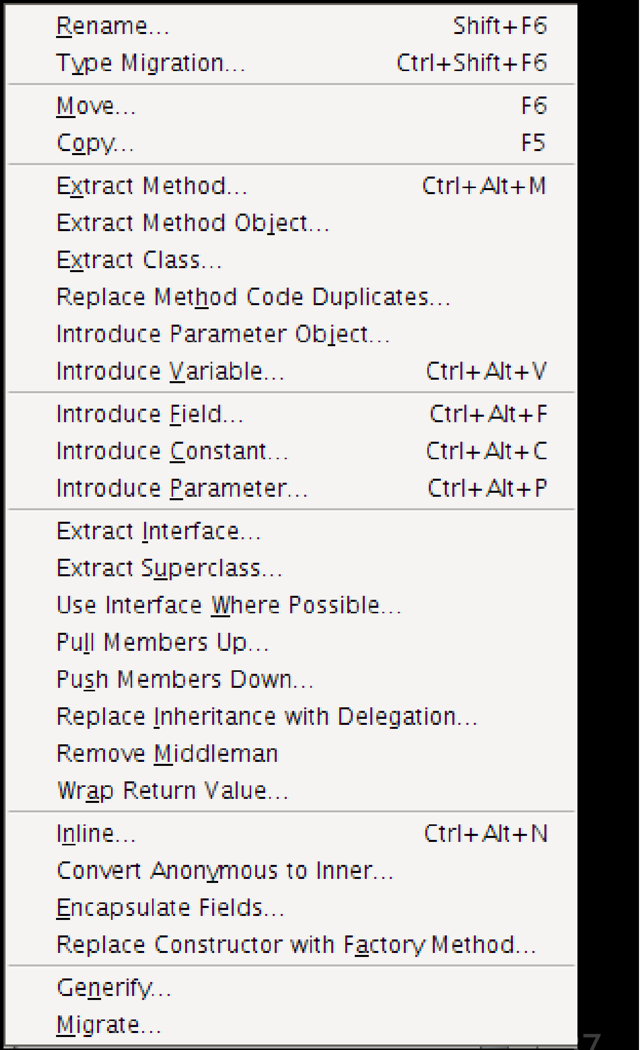
- Delta scripts
  - dbdeploy, liquibase, deltasql
  - Easy to write our own!
- PL/SQL code



VS

Oracle SQL Developer

IntelliJ IDEA (Java)


Rename... Shift+F6
Type Migration... Ctrl+Shift+F6
Move... F6
Copy... F5
Extract Method... Ctrl+Alt+M
Extract Method Object...
Extract Class...
Replace Method Code Duplicates...
Introduce Parameter Object...
Introduce Variable... Ctrl+Alt+V
Introduce Field... Ctrl+Alt+F
Introduce Constant... Ctrl+Alt+C
Introduce Parameter... Ctrl+Alt+P
Extract Interface...
Extract Superclass...
Use Interface Where Possible...
Pull Members Up...
Push Members Down...
Replace Inheritance with Delegation...
Remove Middleman
Wrap Return Value...
Inline... Ctrl+Alt+N
Convert Anonymous to Inner...
Encapsulate Fields...
Replace Constructor with Factory Method...
Generify...
Migrate...

Enabling database refactoring

(1) Development Sandboxes

(2) Regression Testing

(3) Transition Window approach

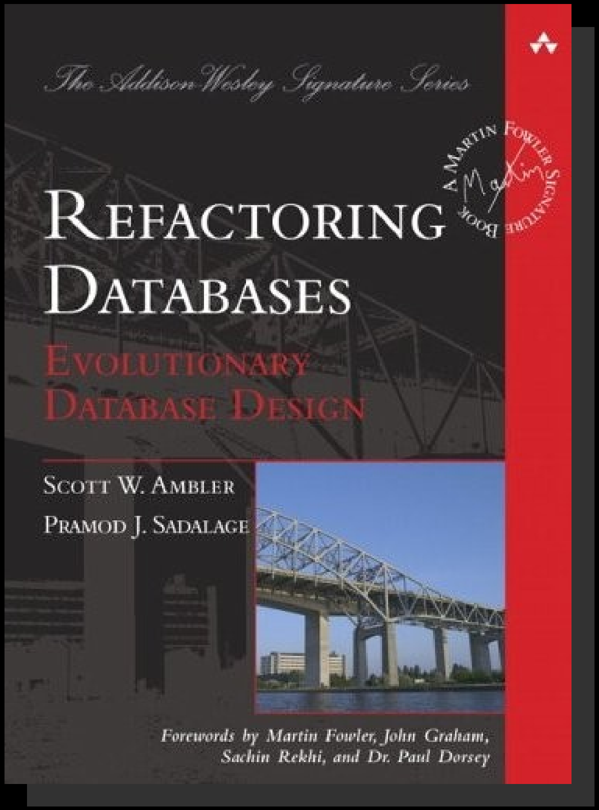
(4) Versioning with Changelog & Delta scripts

(5) Continuous integration

(6) Teamwork & Cultural Changes

The Catalog

- Scott Ambler and Pramod Sadalage have created a nice catalog of DB refactorings
- http://www.ambysoft.com/books/refactoringDatabases.html
- Classification
  - Structural
  - Data Quality
  - Referential Integrity
  - Architectural
  - Method (Stored code)
  - Transformations (non-refactorings)


The Addison Wesley Signature Series
A Martin Fowler Signature Book
Refactoring Databases
Evolutionary Database Design
Scott W. Ambler
Pramod J. Sadalage
Forewords by Martin Fowler, John Graham, Sachin Rekhi, and Dr. Paul Dorsey

Best practices

- Refactor to ease additions to your schema
- Ensure the test suite is in place
- Take small steps
- Program for people
- Don’t publish data models prematurely
- The need to document reflects a need to refactor
- Test frequently

(according to Scott Ambler)