# Problems and prospects for bidirectional transformations

Perdita Stevens

University of Edinburgh

Keynote for Reversible Computation, July 2020

# Part 1
# Software Engineering and its problems

# In the beginning was the Software Crisis

**1996** Dagstuhl on History of Software Engineering.

*In the 1960s, the efficient and timely production and maintenance of reliable and useful software was viewed as a major problem. In the 1990s, it is still considered a major problem. The "software crisis" which was declared three decades ago persists, assuming it makes any sense to speak of a thirty year crisis. Although most would admit to some amelioration of the "crisis," steadily increasing requirements and ambitions have helped sustain it.*

*At the NATO conferences of the late sixties, the solution to the "crisis" was declared to be "software engineering." This, however, begged a number of questions. What is the nature of software as a technological medium? How does software development compare and contrast with other areas of technological practice. What is engineering? Is it sensible to speak of engineering software?*

*Answering these questions has been a difficult and tempestuous process which continues to this day.* Peter Shapiro

# The Rise and Fall of the Software Crisis


https://books.google.com/ngrams/

# The fundamental problem of software engineering

# Approaches to the problem

Subtly different, yet all fundamentally the same idea:

- abstraction, on every level – high level languages, interfaces, verification techniques, unit testing...
- separation of concerns – e.g., into models
- sequentialisation – e.g., YAGNI, bounded small releases.

**All about humans**

Getting chunks of processing done within the limited capacity of an individual human brain.

# The Software Crisis is dead...

Long live

**The Software Capacity Crisis!**

- hundreds of thousands of unfilled ICT positions in EU
- 1.6 *million* ICT professional jobs to fill in EU by 2030
- Capacity and hiring top lists of software companies' concerns

# The Software Crisis is dead...

Long live

The Software Capacity Crisis!

- hundreds of thousands of unfilled ICT positions in EU
- 1.6 *million* ICT professional jobs to fill in EU by 2030
- Capacity and hiring top lists of software companies' concerns
- yet unemployment rate for computer science graduates above that of other STEM subjects!

# The Software Crisis is dead...

Long live

**The Software Capacity Crisis!**

- hundreds of thousands of unfilled ICT positions in EU
- 1.6 *million* ICT professional jobs to fill in EU by 2030
- Capacity and hiring top lists of software companies' concerns
- yet unemployment rate for computer science graduates above that of other STEM subjects!

There is demand – for super-humans.

*S., The Future of Programming and Modelling: a Vision (to appear)*

# TANSTAAFL

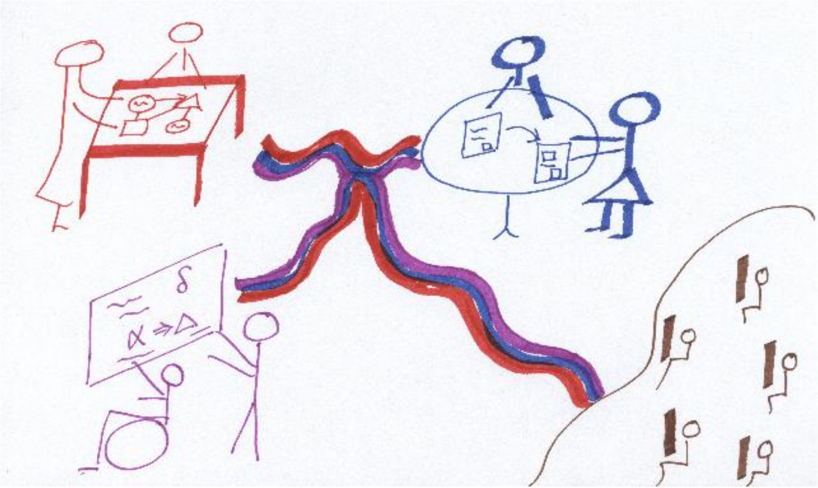
Splitting the overwhelming amount of work into chunks helps a lot.

But the difficulty then becomes integration of the chunks.

- Mythical Man Month;
- integration ("continuous" or "phase");
- paying off technical debt in sprint-based projects;
- MDE e.g. managing projects with multiple DSLs.

Today's techniques support individuals in temporarily focusing on one concern. That's not enough.

# In practice, models are not independent



# Part 1 conclusion

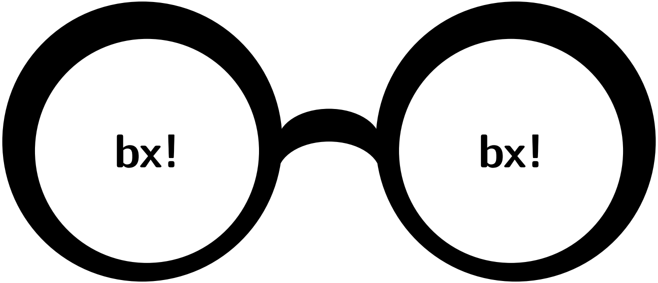
Separation of concerns 

Integration of concerns 

# Part 2
# Bidirectional transformations
# bx

# Bidirectionality is everywhere!

at least after you start looking at things that way



# Essence of bidirectionality

- multiple models that are
- live
- not orthogonal

bidirectional transformations maintain consistency between models

# What's a model?

Everything's a model!

# What's a model?

Everything's a model!

A model is an abstract, usually graphical, representation of some aspect of a system

# For example

- UML model
- database schema
- map of user's navigation between screens
- bunch of Java code
- bunch of JUnit tests.

A model supports the work of a particular group of people. Ideally, it records all and only the information they need to do their work.

So, having multiple models is a consequence of separation of concerns.

A model is a representation of a concern

# Models that are live

i.e. may need to be updated at some time in the future.

# Models that are live

i.e. may need to be updated at some time in the future.

As opposed to:

ideal refinement-based development, in which you may:

1. develop a model
2. derive a new model from it
3. "throw it over the wall"
4. never touch the original again.

E.g. write the JUnit tests; then freeze them, and write the code.

# Models that are live

i.e. may need to be updated at some time in the future.

As opposed to:

ideal refinement-based development, in which you may:

1. develop a model
2. derive a new model from it
3. “throw it over the wall”
4. never touch the original again.

E.g. write the JUnit tests; then freeze them, and write the code.

Mostly, life is not ideal.

# Models that are not orthogonal

There's no problem having several live models – if the information they record is completely independent.

Otherwise, dependencies must be managed somehow.

$$\text{JUnit} \longleftrightarrow \text{Java}$$

- Some changes can be made independently
- Others necessitate a change on the other side
- There may be many ways to restore consistency
- Some are better than others!

# Models that are not orthogonal

There's no problem having several live models – if the information they record is completely independent.

Otherwise, dependencies must be managed somehow.

$$\text{JUnit} \longleftrightarrow \text{Java}$$

- Some changes can be made independently
  – e.g. refactor the Java
- Others necessitate a change on the other side
- There may be many ways to restore consistency
- Some are better than others!

# Models that are not orthogonal

There's no problem having several live models – if the information they record is completely independent.

Otherwise, dependencies must be managed somehow.

$$\text{JUnit} \longleftrightarrow \text{Java}$$

- Some changes can be made independently
  – e.g. refactor the Java
- Others necessitate a change on the other side
  – e.g. change the name of a method
- There may be many ways to restore consistency

- Some are better than others!

# Models that are not orthogonal

There's no problem having several live models – if the information they record is completely independent.

Otherwise, dependencies must be managed somehow.

$$\text{JUnit} \longleftrightarrow \text{Java}$$

- Some changes can be made independently
  – e.g. refactor the Java
- Others necessitate a change on the other side
  – e.g. change the name of a method
- There may be many ways to restore consistency
  – e.g. change the method name in relevant tests, or delete relevant tests
- Some are better than others!

# The two tasks of bidirectional thinking

1. check whether all is well (consistency checking);
2. if not, fix it (consistency restoration).

# The two tasks of bidirectional thinking

1. check whether all is well (consistency checking);
2. if not, fix it (consistency restoration).

Choices include

- how much to articulate about "all is well";
- how much to automate consistency restoration
- what kind of fixes to consider – changing one model, changing both?
- what information to maintain in order to do all this – traces, history, deltas, edits...?

bx = bidirectional transformation
= artefact for automating those tasks, maybe partially

# Spoiler

We do not have
and are not likely to have
one true way to write bx

## Choose how much consistency to articulate

What should "all is well" mean for JUnit $\longleftrightarrow$ Java ?

1. The files compile together without error
2. ... and the JUnit file includes a test for every public method?
3. ... and all the tests pass?
4. ... and a certain coverage criterion is met?

More stringent $\Rightarrow$

- more informative
- less flexible
- more difficult to restore consistency
- more work potentially saved for the user.

# Choose how much consistency restoration to automate

Valid choices include:

- None
- All
- All except when things go wrong
- Partial

# Choose how much consistency restoration to automate

Valid choices include:

- None
  – automatic checking, but fixing done by humans
- All
- All except when things go wrong
- Partial

# Choose how much consistency restoration to automate

Valid choices include:

- None
  – automatic checking, but fixing done by humans
- All
  – fully automatic
- All except when things go wrong
- Partial

# Choose how much consistency restoration to automate

Valid choices include:

- None
  – automatic checking, but fixing done by humans
- All
  – fully automatic
- All except when things go wrong
  – fully automatic but may fail
- Partial

# Choose how much consistency restoration to automate

Valid choices include:

- None
  – automatic checking, but fixing done by humans
- All
  – fully automatic
- All except when things go wrong
  – fully automatic but may fail
- Partial
  – e.g. bx improves consistency, but may leave some work for humans

*S., Bidirectionally Tolerating Inconsistency: Partial Transformations, FASE'14*

# Levels of bx thinking

for integration of concerns

- thinking about consistency explicitly
- programming its checking and/or restoration (in a GPL)
- programming bidirectionally (in a bx language)

# Why a bx language?

You can write a bx in a GPL:

- one program to check consistency, e.g. type $M \times N \to \mathsf{Bool}$
- separate programs to restore consistency, e.g. type $M \times N \to M$

But these tasks are tightly coupled, so it pays to integrate their automation:

- avoid duplication of information
- guarantee sensible (predictable, dependable...) joint behaviour.

A bx program records in one artefact how to check and restore consistency.

# Programming bidirectionally

What would a great bx language be like?

What properties would it enforce?

View update: special case of bx, lens (Harmony, Foster/Pierce):

| | | | |
|---|---|---|---|
| $S$ | $\rightarrow$ | $V$ | concrete Source, abstract View |
| $s$ | $\mapsto$ | $v$ | i.e. $s$ is consistent with $v$ |

# Programming bidirectionally

What would a great bx language be like?

What properties would it enforce?

View update: special case of bx, lens (Harmony, Foster/Pierce):

| | | | |
|---|---|---|---|
| $S$ | $\rightarrow$ | $V$ | concrete Source, abstract View |
| $s$ | $\mapsto$ | $v$ | i.e. $s$ is consistent with $v$ |
| $s'$ | $\mapsto$ | ? | change $s$ to $s'$? restoring consistency is easy! |

# Programming bidirectionally

What would a great bx language be like?

What properties would it enforce?

View update: special case of bx, lens (Harmony, Foster/Pierce):

| | | | |
|---|---|---|---|
| $S$ | $\rightarrow$ | $V$ | concrete Source, abstract View |
| $s$ | $\mapsto$ | $v$ | i.e. $s$ is consistent with $v$ |
| $s'$ | $\mapsto$ | ? | change $s$ to $s'$? restoring consistency is easy! |
| ? | $\leftarrow\!\shortmid$ | $v'$ | change $v$ to $v'$? not so easy! |
| | | | ? had better be consistent with $v'$ correct |
| | | | but we'd also like back what was abstracted away |

# Programming bidirectionally

What would a great bx language be like?

What properties would it enforce?

View update: special case of bx, lens (Harmony, Foster/Pierce):

| | | | |
|---|---|---|---|
| $S$ | $\rightarrow$ | $V$ | concrete Source, abstract View |
| $s$ | $\mapsto$ | $v$ | i.e. $s$ is consistent with $v$ |
| $s'$ | $\mapsto$ | ? | change $s$ to $s'$? restoring consistency is easy! |
| ? | $\leftarrow\!\shortmid$ | $v'$ | change $v$ to $v'$? not so easy! |
| | | | ? had better be consistent with $v'$ correct |
| | | | but we'd also like back what was abstracted away |
| $s'$ | $\leftarrow\!\shortmid$ | $(v', s)$ | lets us get also $v' = v \Rightarrow s' = s$ hippocratic |

# Symmetrise

Why symmetrise?

- need to work with models that conceptually overlap
- want to choose how stringent the consistency we use is

$$M \xleftarrow{\overleftarrow{R}} M \times N \xrightarrow{\overrightarrow{R}} N$$

Each consistency restoration function must be

- correct : really does restore consistency
- hippocratic : does nothing if arguments already consistent

with respect to consistency relation

$$R \subseteq M \times N$$

Together, $R$, $\overrightarrow{R}$, and $\overleftarrow{R}$ form a bx.

# Authoritative model

In this formalisation, one model is authoritative at each consistency restoration: it does not change.

Informally: its consistency-relevant information overwrites corresponding information in the other model.

Sometimes, a synchronisation approach, where both models change, is better.

However, that tends to be more complicated, and there is a danger of "collapsing" both models into an uninformative consistent state.

Keeping one model unchanged "keeps us honest".

# Undoability, weak and strong

Correctness and hippocraticness are only part of what we want from a "good" bx.

Consider this sequence of model edits and consistency restorations:

$m \quad n \quad$ state of the models when we come in

# Undoability, weak and strong

Correctness and hippocraticness are only part of what we want from a "good" bx.

Consider this sequence of model edits and consistency restorations:

| | | |
|---|---|---|
| $m$ | $n$ | state of the models when we come in |
| $m$ | $n'$ | someone edits $n$ to $n'$ |

# Undoability, weak and strong

Correctness and hippocraticness are only part of what we want from a "good" bx.

Consider this sequence of model edits and consistency restorations:

| | | |
|---|---|---|
| $m$ | $n$ | state of the models when we come in |
| $m$ | $n'$ | someone edits $n$ to $n'$ |
| $m'$ | $n'$ | we restore consistency, so $m$ becomes $m'$ |

# Undoability, weak and strong

Correctness and hippocraticness are only part of what we want from a "good" bx.

Consider this sequence of model edits and consistency restorations:

| | | |
|---|---|---|
| $m$ | $n$ | state of the models when we come in |
| $m$ | $n'$ | someone edits $n$ to $n'$ |
| $m'$ | $n'$ | we restore consistency, so $m$ becomes $m'$ |
| $m'$ | $n$ | now someone puts $n'$ back to $n$ |

# Undoability, weak and strong

Correctness and hippocraticness are only part of what we want from a "good" bx.

Consider this sequence of model edits and consistency restorations:

| | | |
|---|---|---|
| $m$ | $n$ | state of the models when we come in |
| $m$ | $n'$ | someone edits $n$ to $n'$ |
| $m'$ | $n'$ | we restore consistency, so $m$ becomes $m'$ |
| $m'$ | $n$ | now someone puts $n'$ back to $n$ |
| ? | $n$ | and we restore consistency again. |

# Undoability, weak and strong

Correctness and hippocraticness are only part of what we want from a "good" bx.

Consider this sequence of model edits and consistency restorations:

| | | |
|---|---|---|
| $m$ | $n$ | state of the models when we come in |
| $m$ | $n'$ | someone edits $n$ to $n'$ |
| $m'$ | $n'$ | we restore consistency, so $m$ becomes $m'$ |
| $m'$ | $n$ | now someone puts $n'$ back to $n$ |
| ? | $n$ | and we restore consistency again. |

If you expect $? = m$, you expect some kind of *undoability*

# Undoability, weak and strong

Correctness and hippocraticness are only part of what we want from a "good" bx.

Consider this sequence of model edits and consistency restorations:

| | | |
|---|---|---|
| $m$ | $n$ | state of the models when we come in |
| $m$ | $n'$ | someone edits $n$ to $n'$ |
| $m'$ | $n'$ | we restore consistency, so $m$ becomes $m'$ |
| $m'$ | $n$ | now someone puts $n'$ back to $n$ |
| ? | $n$ | and we restore consistency again. |

If you expect $? = m$, you expect some kind of *undoability*

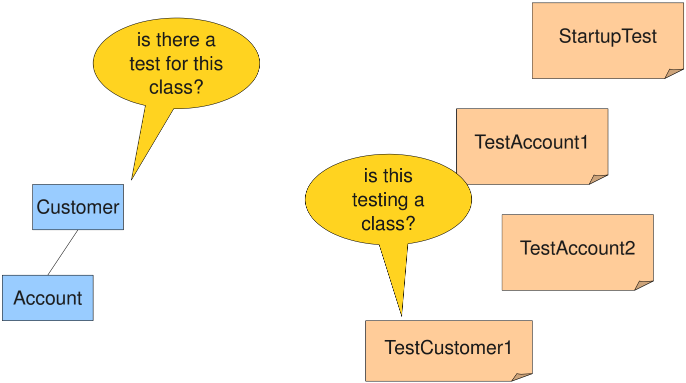
What I didn't specify: are we assuming $m$ and $n$ consistent?

If yes: *weak* undoability

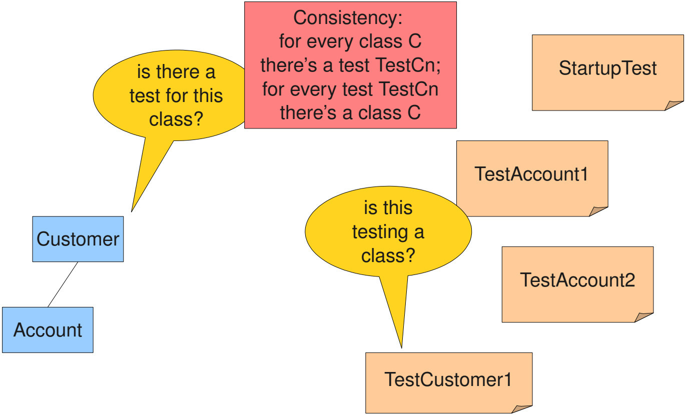
If no: *strong* undoability, aka history ignorance

Unfortunately, either is too much to expect...

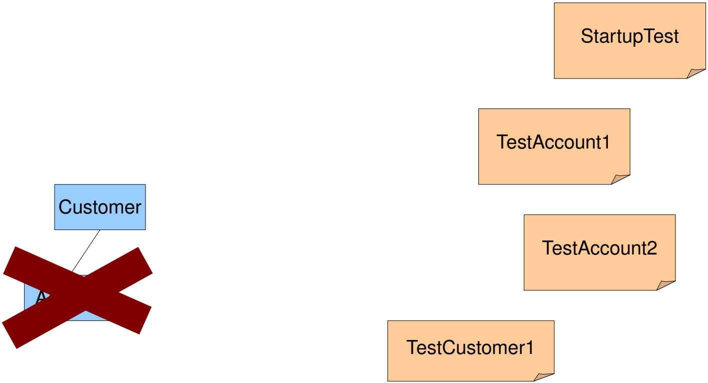
# Informal example: Java-JUnit again



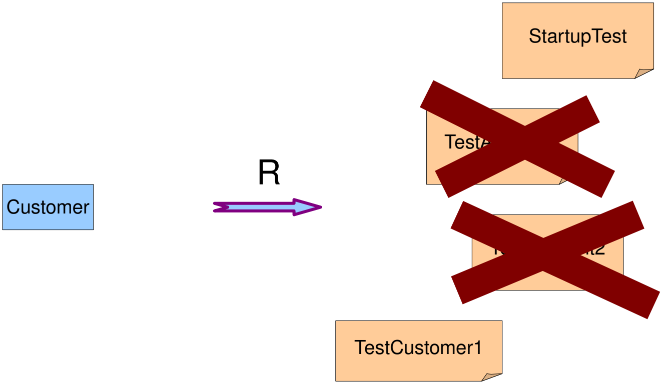
# A possible consistency relation



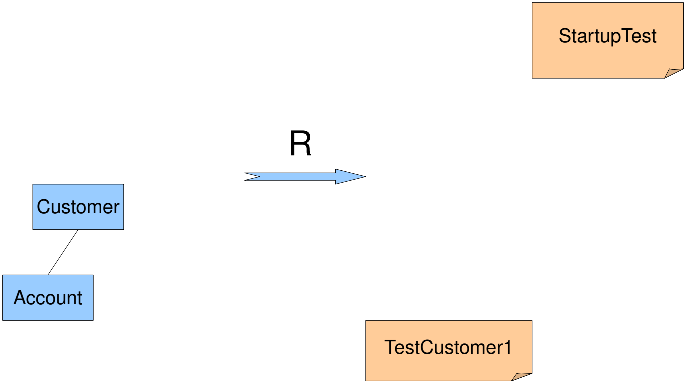
# Newbie deletes class Account...



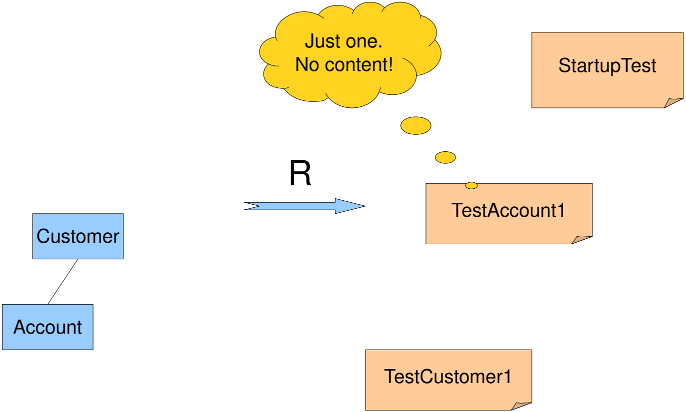
# ...and propagates the change



# Then realises her mistake!



# And propagates...



# Formalising “consistency-relevant information”

- $m \sim_F m' \Leftrightarrow \forall n \in N. \overrightarrow{R}(m,n) = \overrightarrow{R}(m',n)$

  *no* differences between $m$ and $m'$ remain visible on the $N$ side

  (their consistency-relevant information is the same already)

- $m \sim_B m' \Leftrightarrow \forall n \in N. \overleftarrow{R}(m,n) = \overleftarrow{R}(m',n)$

  *all* differences between $m$ and $m'$ are visible on the $N$ side

  (overwriting their consistency-relevant information makes them the same)

Crucial Fact

$m = m'$ iff $m \sim_F m'$ and $m \sim_B m'$

Hence, if we pick any transversals for $\sim_F$ and $\sim_B$, we can use them as coordinates for $M$.

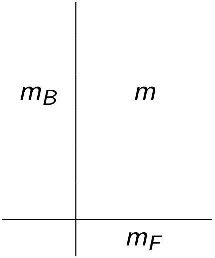
(And dually for $N$ of course.)

# Coordinate grid

Pick a transversal, i.e. $M_F \subseteq M$ including exactly one representative of every $\sim_F$-equivalence class.

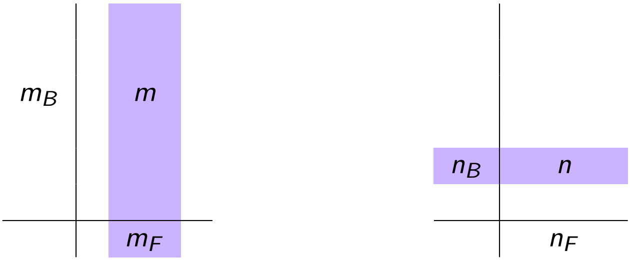
Define $M_B$, $N_F$, $N_B$ similarly.

Represent arbitrary $m$ by the unique $(m_F, m_B)$ s.t. $m \sim_F m_F$ and $m \sim_B m_B$.




Positions can be empty, but no position contains more than one element.

# Consistency depends only on $M_F$ and $N_B$




$R(m, n)$ iff $R(m_F, n_B)$

# Java-JUnit

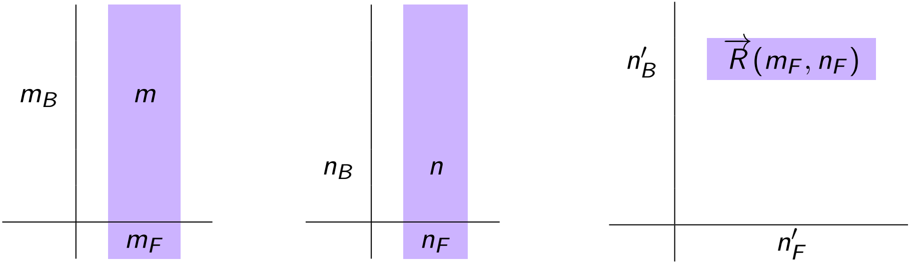
For $m$ a set of Java classes:

- the $\sim_F$-equivalence class is given by the classnames in $m$
- the $\sim_B$-equivalence class is more complicated...

For $n$ a set of JUnit tests:

- the $\sim_B$-equivalence class is given by names appearing in tests `TestName`$X$
- the $\sim_F$-equivalence class is more complicated...

## Result of $\overrightarrow{R}$ depends only on $M_F$ and $N_F$

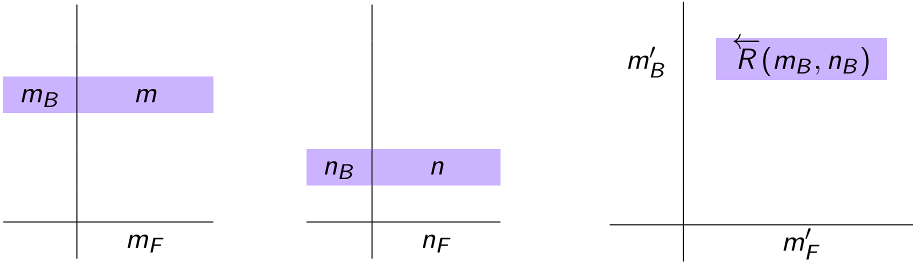



NB $n'_F$ and $n'_B$ are new.

$n'_B$ must be a row that is consistent with the column given by $m_F$ (there might be one, or several)

$n'_F$ might have changed too – need not be $n_F$ (indeed those squares might be empty!)

# Result of $\overleftarrow{R}$ depends only on $M_B$ and $N_B$




NB $m'_F$ and $m'_B$ are new.

$m'_F$ must be a column that is consistent with the row given by $n_B$ (there might be one, or several)

$m'_B$ might have changed too – need not be $m_B$ (indeed those squares might be empty!)

# The following are equivalent

1. $R : M \leftrightarrow N$ is strongly undoable.
2. $M$ and $N$ are full with respect to $R$.
3. For each $m \in M$ and $n \in N$ we have

   $$\overrightarrow{R}(m, n) \sim_F n$$

   that is, $\overrightarrow{R}$ stabilises the coordinate grid columns of $N$, and

   $$\overleftarrow{R}(m, n) \sim_B m$$

   that is, $\overleftarrow{R}$ stabilises the coordinate grid rows of $M$.

# Coordinate representation of models

Informally, those conditions amount to "the information relevant to consistency is independent of the rest of the information".

Lovely when you can get it – which is not often.

A bx language that enforced this would be too inexpressive

*S., Observations relating to the equivalences induced on model sets by bidirectional transformations, EASST 49, 2012*

# Least change, strong and weak

As soon as you have a notion of “small” change, you would like:

> a small change on one side causes only a small change on the other

and/or

> by limiting the amount of change on one side, you can limit your exposure to change on the other

Again, it matters whether or not you assume you are starting from a consistent point.

(Why wouldn’t you? Simultaneously live models – the other side doesn’t stop working on their model because you are working on yours.)

*Cheney, Gibbons, McKinna, S., On principles of Least Change and Least Surprise for bidirectional transformations, JOT 16/1, 2017*

# Problems

- each side has information not present on the other: lenses are not enough
- in practice, information relevant to consistency is interdependent with the rest: can't insist on strong undoability
- "the right" metric depends strongly on your perspective
- and you may not want least change wrt it anyway
- and btw, computing metric-least consistency restoration is NP-hard

etc., etc.

# Part 2 summary

It's fun to write down properties you'd like of your bx...

... but you can't have them (all).

Different bx languages are good at different things.

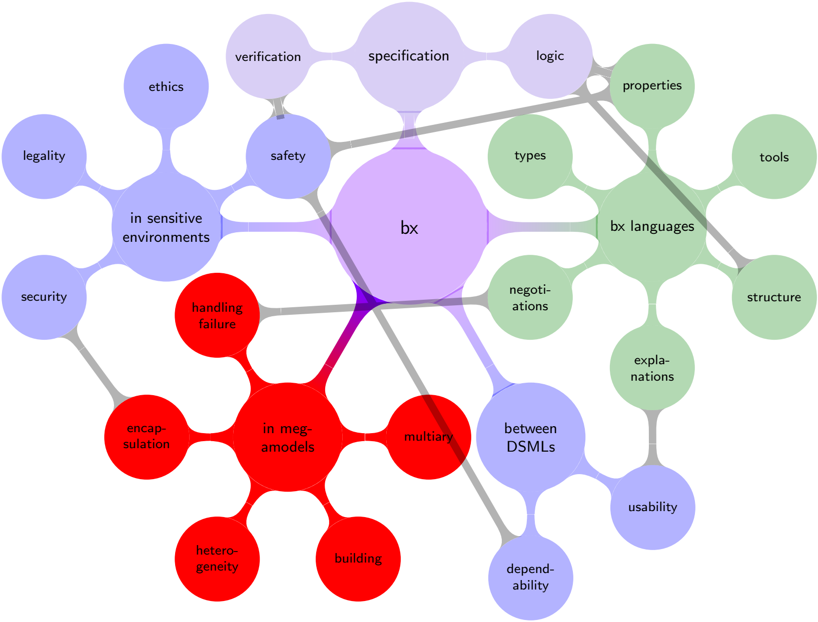
So we need them to be able to interoperate.

In a setting with lots of models!

That may be related by bx in different languages!

# Part 3
# Heterogeneity

# Bx in the large: problems to work on



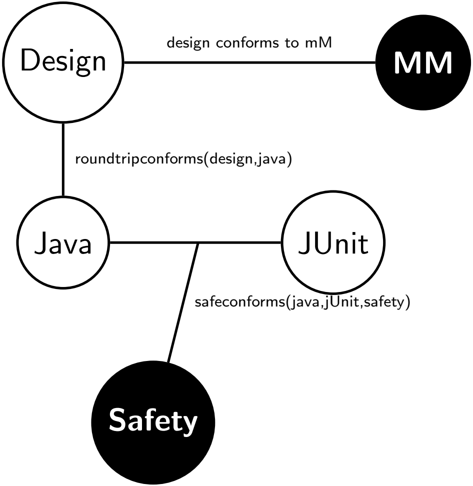
# Bx in the large

Going beyond just two models...

... in the bx community we'd like to be able to restore consistency in collections of models, when any of them changes.

*S., Maintaining Consistency in Networks of Models: Bidirectional Transformations in the Large, SoSyM 19(1), 2020*

# Example megamodel



# Consistency restoration problem

Given: some models (some authoritative), connected by some binary bx.

Find: a sequence of applications of the binary bx's consistency restoration functions that restores *all* the consistency relations.

# Consistency restoration problem

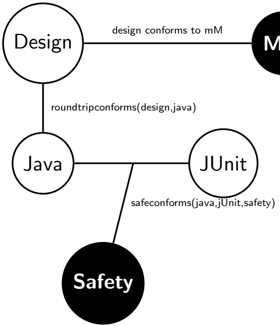
Given: some models (some authoritative), connected by some binary bx.

Find: a sequence of applications of the binary bx's consistency restoration functions that restores *all* the consistency relations.

Bad news: mostly impossible.

# Things that can go wrong

- there may simply be no solution
- it may not be reachable by any sequence of the bx you have
- different sequences may give different solutions




More interestingly, the bx you have may be almost able to do it:

"if only I could apply this one, then fiddle the result a bit, then apply that one..."

# David Wheeler's Law

All problems in computer science can be solved by another level of indirection

# Builders

Each model that should be modified automatically is given a builder

which, on demand, modifies its model to bring it into consistency with a given collection of its neighbours.

The builder might:

- invoke some bx
- in an order of its choosing, maybe repeatedly
- "fix things up" in between or afterwards
- interact with a user
- search
- anything else appropriate

– provided that in the end it either restores consistency, or fails.

# But what invokes the builders?

Observation: restoring consistency in a network of models is very like building a software system from sources.

Therefore: Adapt the pluto build system (Erdweg et al.), allowing

- proven soundness and optimality (in appropriate senses...)
- dynamic dependencies
- early cut-off
- custom stamps to identify when re-checking is needed

*S., Connecting Software Build with Maintaining Consistency between Models: Towards sound, optimal, and flexible building from megamodels, SoSyM 2020*

# Key decision 1: pull, don't push

Observation: pushing all changes through a network is disruptive and unnecessary.

Therefore: instead, select a target, and rebuild only as necessary to bring it up to date.

That is:

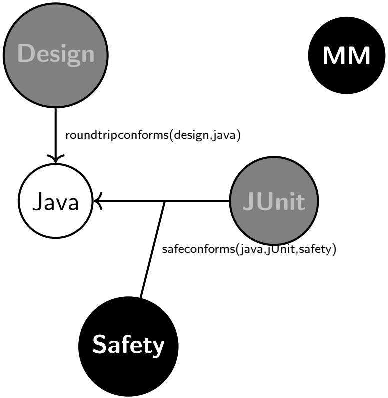
- decide which model you want to work on
- bring it into consistency with everything relevant to it
- including updating those things if necessary
- but ignoring anything you can, e.g., any model that just depends on this one.

# Key decision 2: use an orientation model

Observation: no hope of consistency restoration being independent of which models can be changed, which take priority, etc.

Therefore: provide explicit, inspectable, familiar control over those things.

E.g.



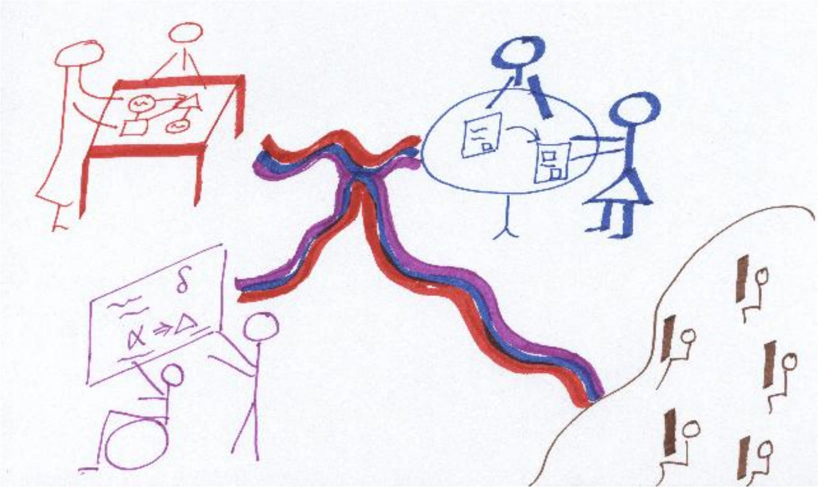
# To use megamodel-pluto

- Design your megamodel; hence
- create an orientation model (just another model: your users can change it, it lives in your CMS).
- For each model you might want to be able to “build”, write a builder, using a skeleton:
- it brings its model into consistency with relevant neighbours, using relevant bx however required.

# Future work: lots, for example

- integrating existing model transformation engines for builders to call on
- generating custom stamps (to check when consistency-relevant information may have changed)
- mechanising proof of correctness (with James McKinna)
- exploring the horizons, e.g. letting builders negotiate with their neighbours
- explaining failure
- practical use!

# Future software engineering: deliverable models



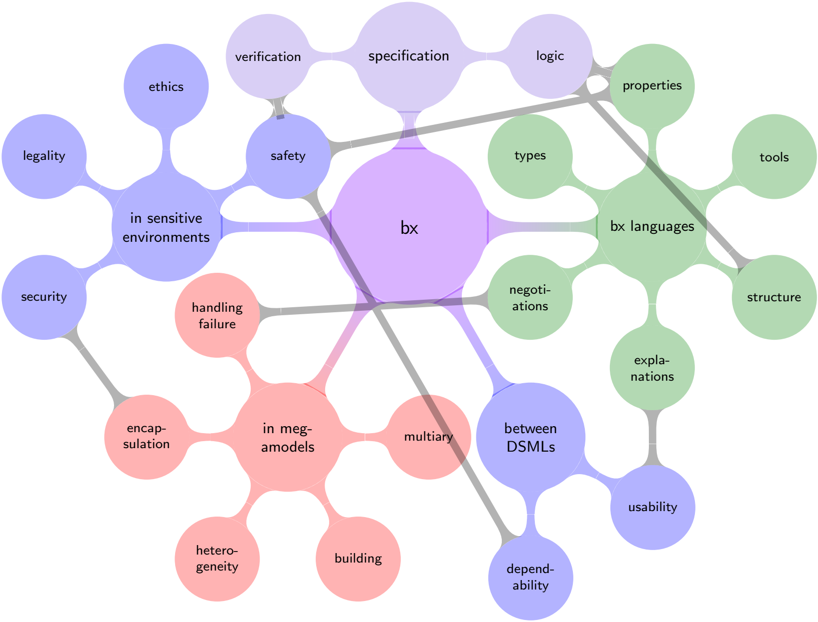
# Conclusions

Bidirectionality is about integrating separated concerns, by maintaining consistency between their representations.

This is difficult...

... but holds out the prospect of a new way to develop software, which may mitigate the software capacity crisis.

# More things to work on...



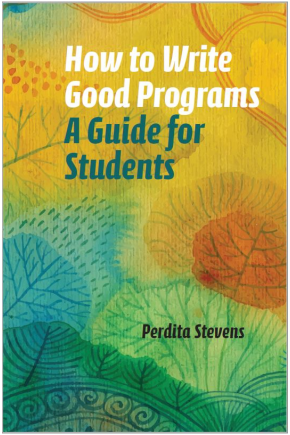
# Questions? and two shameless plugs

MDE Network, starting soon – see my webpage next month

Out this month from CUP

*"provides a wealth of excellent advice tailored to beginning students of programming. It is language-agnostic, well structured, and delivered in an accessible manner. It might as well have the words 'Don't Panic' in large, friendly letters on the cover."*

Prof. Jeremy Gibbons
University of Oxford



# Aside: Eating own dogfood considered harmful

Hardest thing in SE methods and tools research, especially logic/verification/PL research:

paying enough attention to your users.

# Aside: Eating own dogfood considered harmful

Hardest thing in SE methods and tools research, especially logic/verification/PL research:

paying enough attention to your users.

- who are they?
- what are they trying to do?
- what will work for them?
- what will they have to know/remember/understand?
- what could possibly go wrong?

# Aside: Eating own dogfood considered harmful

Hardest thing in SE methods and tools research, especially logic/verification/PL research:

paying enough attention to your users.

- who are they?
- what are they trying to do?
- what will work for them?
- what will they have to know/remember/understand?
- what could possibly go wrong?

Can the weakest 20% of your SE class use your work correctly?

# Aside: Eating own dogfood considered harmful

Hardest thing in SE methods and tools research, especially logic/verification/PL research:

paying enough attention to your users.

- who are they?
- what are they trying to do?
- what will work for them?
- what will they have to know/remember/understand?
- what could possibly go wrong?

Can the weakest 20% of your SE class use your work correctly?

If not, why not? And is it OK?

# You don't always want metric least change

$R$ relates UML model $m$ with test suite $n$, maintaining:

> every class in $m$ stereotyped ⟪persistent⟫ has a test class of the same name in $n$, containing an appropriate (in some specified sense) set of tests for each public operation

(but $n$ may also contain other tests).

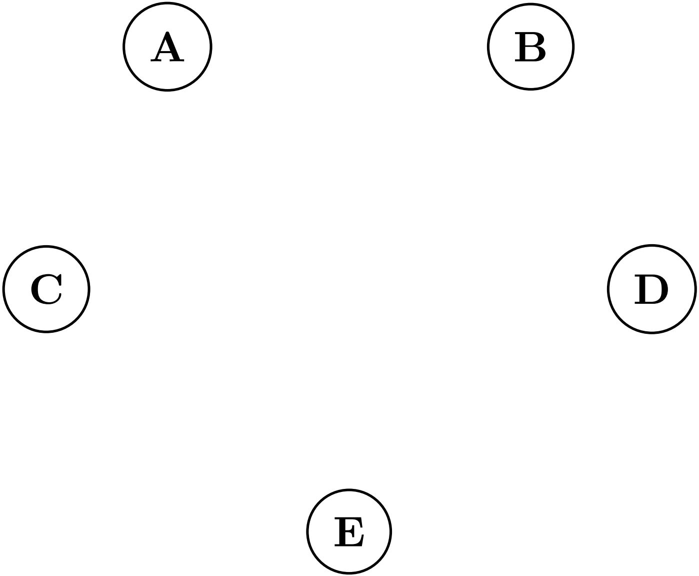
You modify the test class for a ⟪persistent⟫ class `C`, to reflect changes made in the code to the signatures of `C`'s methods, e.g., say `int` has changed to `long` throughout.

$R$ now propagates necessary changes to the model $m$.

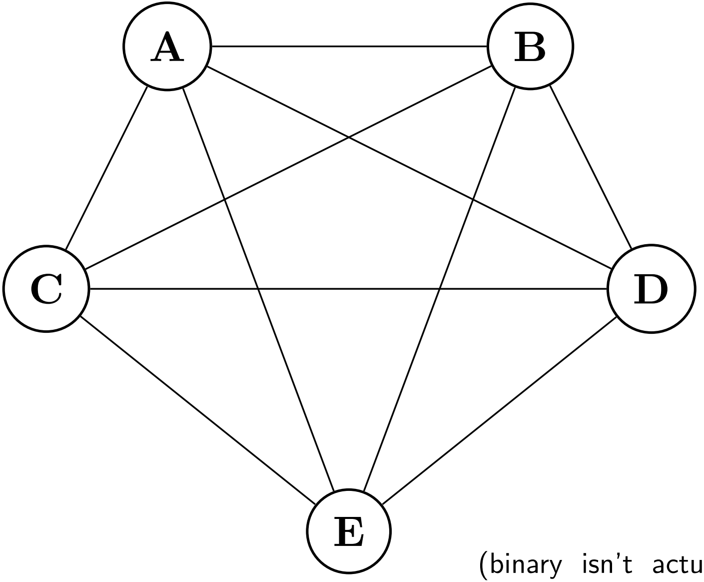
You expect $R$ to perform appropriate changes to the detail of persistent class `C` in the model, changing `int` to `long` in the signatures of its operations.

But instead, $R$ removes the stereotype from `C`! There is no longer be any consistency requirements relating to `C`. Shorter edit distance, but not what you wanted.

## In the abstract: we have some model sets

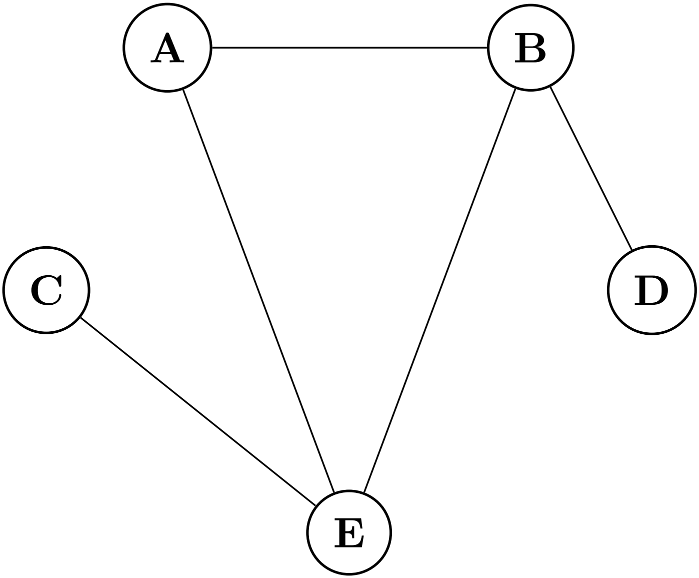


## and some consistency relations

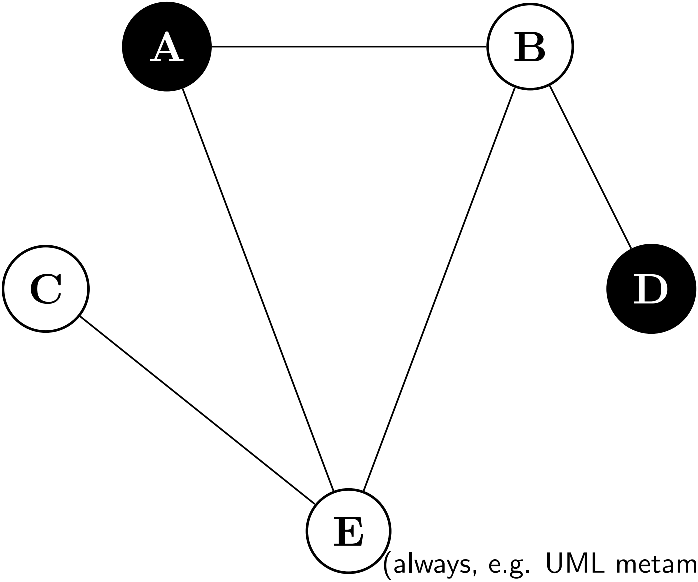



(binary isn't actually important today)

but we needn't draw the universal ones!



## Some models will be authoritative




(always, e.g. UML metamodel, or right now, e.g. model last edited)