a way to track data structure evolution, cheaply and automatically

Nick Mitchell
nickm@us.ibm.com

IBM TJ Watson Research Center
Hawthorne, New York, USA

December 4, 2003

our three constraints

(framework-intensive applications are fun!)

- as automated as possible
  - no shallow patterns identify the bug
    (not dominant type or allocation site or biggest data structure, not even diff'd over time)
  - application-level memory management
    (caches, pools, lazy pools, lazy or asynchronous deallocation policies)
  - sandwiching effects
    (the framework is the driver, your application is just along for the ride)
- scale to gigantic heaps
  - e.g. 40 million objects on a laptop, analyzed in a few minutes
- impose minimal perturbation
  - time and space perturbation on the server must be in the noise

the common datatypes don't help diagnose problems with structure evolution

	live instances
java/lang/String	230025
com/ibm/servlet/util/HashtableEntry	92825
java/util/Hashtable$Entry	59727
org/apache/xerces/dom/TextImpl	15627
org/apache/xerces/dom/AttrImpl	11278
org/apache/xalan/xpath/xml/StringToStringTable	11204
...	...
org/apache/xerces/dom/DocumentImpl	52

the big data structures are... big, not bugs

	# constituents
com/ibm/servlet/DynamicClassLoader	82882
com/ibm/servlet/DynamicClassLoader	73537
com/.../XSLTransform	71628
com/.../PropertiesFactory	66957
elements of Finalizer queue	39886
org/apache/xalan/xslt/TemplateList	28969
owned by native code	18829
...	...

if we wait long enough, then the leaking data structure will float to the top; otherwise, noise effects dominate

our three constraints

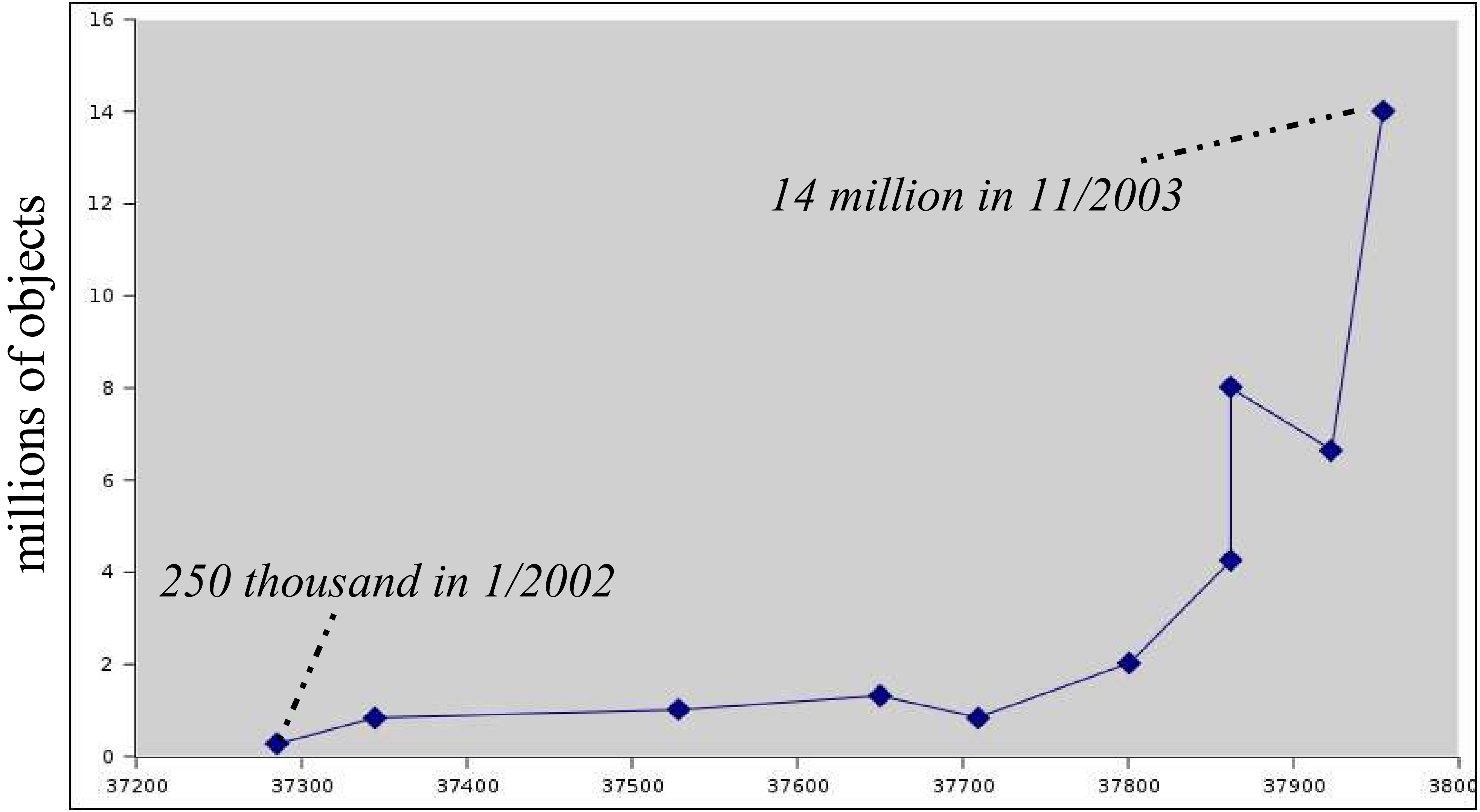
(framework-intensive applications are fun!)

- as automated as possible
  - no shallow patterns identify the bug
    (not dominant type or allocation site or biggest data structure, not even diff'd over time)
  - application-level memory management
    (caches, pools, lazy pools, lazy or asynchronous deallocation policies)
  - sandwiching effects
    (the framework is the driver, your application is just along for the ride)

- scale to gigantic heaps
  - e.g. 40 million objects on a laptop, analyzed in a few minutes

- impose minimal perturbation
  - time and space perturbation on the server must be in the noise

object reference graphs are getting very large

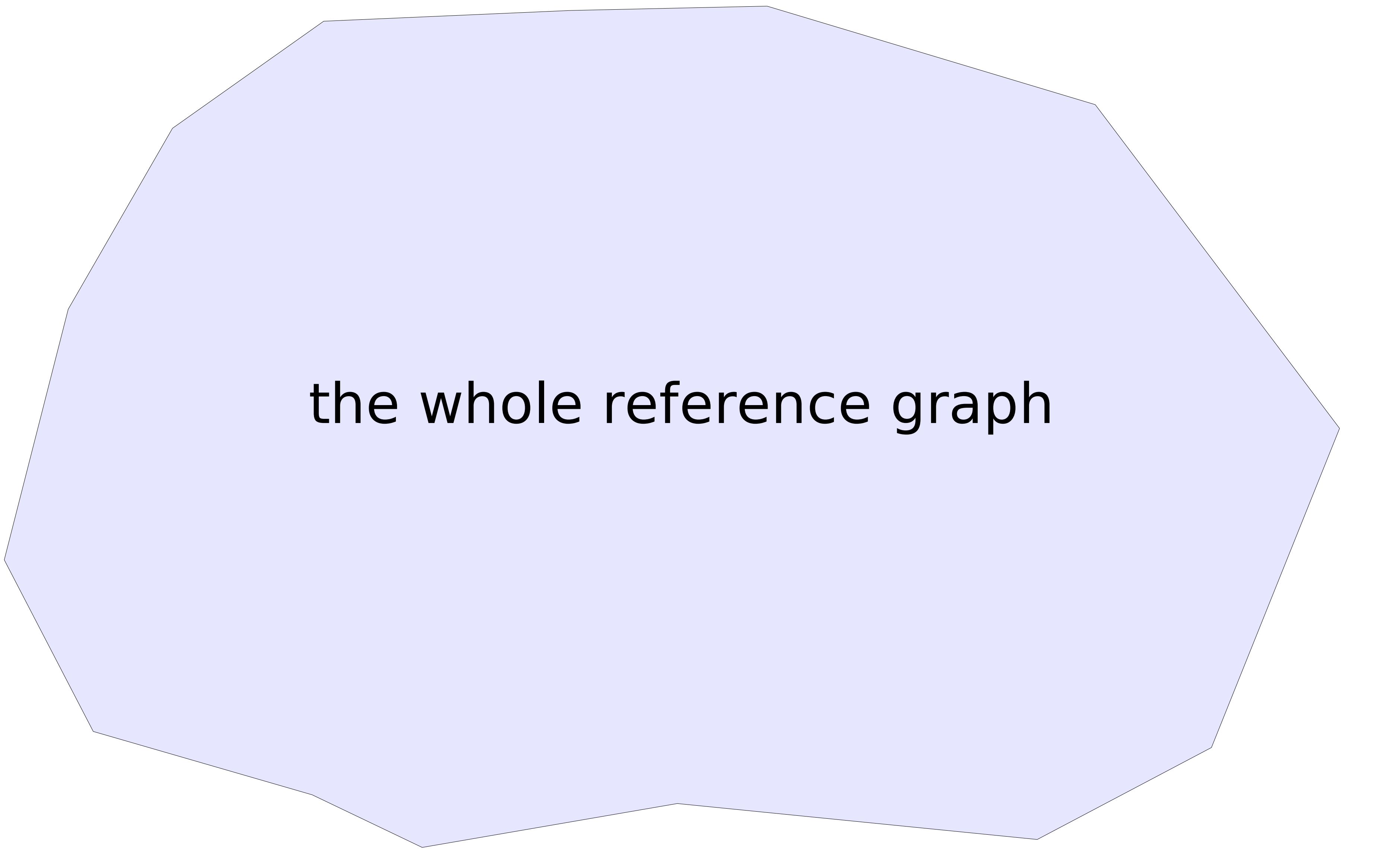


our three constraints

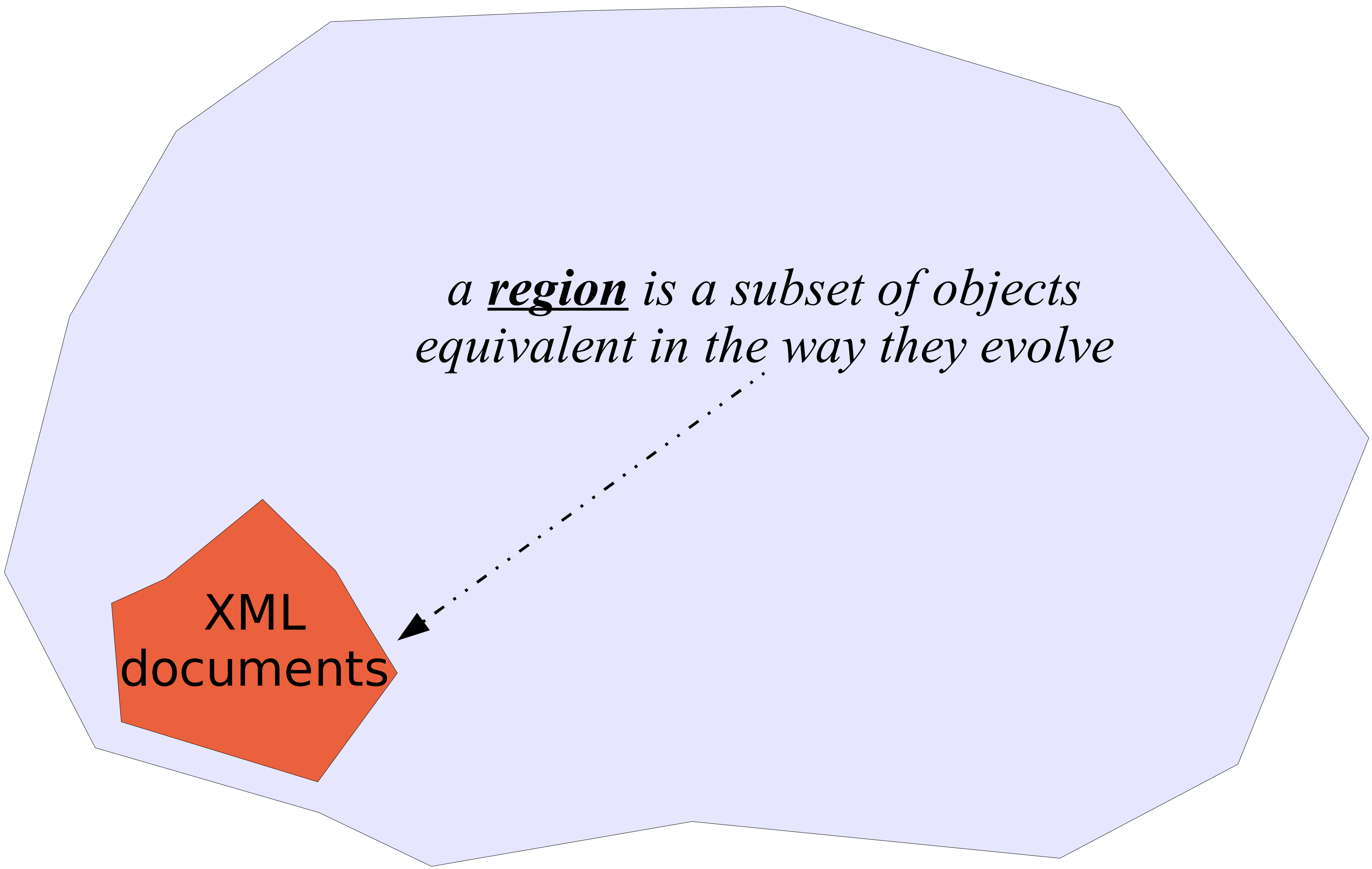
(framework-intensive applications are fun!)

- as automated as possible
  - no shallow patterns identify the bug
    (not dominant type or allocation site or biggest data structure, not even diff'd over time)
  - application-level memory management
    (caches, pools, lazy pools, lazy or asynchronous deallocation policies)
  - sandwiching effects
    (the framework is the driver, your application is just along for the ride)
- scale to gigantic heaps
  - e.g. 40 million objects on a laptop, analyzed in a few minutes
- impose minimal perturbation
  - time and space perturbation on the server must be in the noise

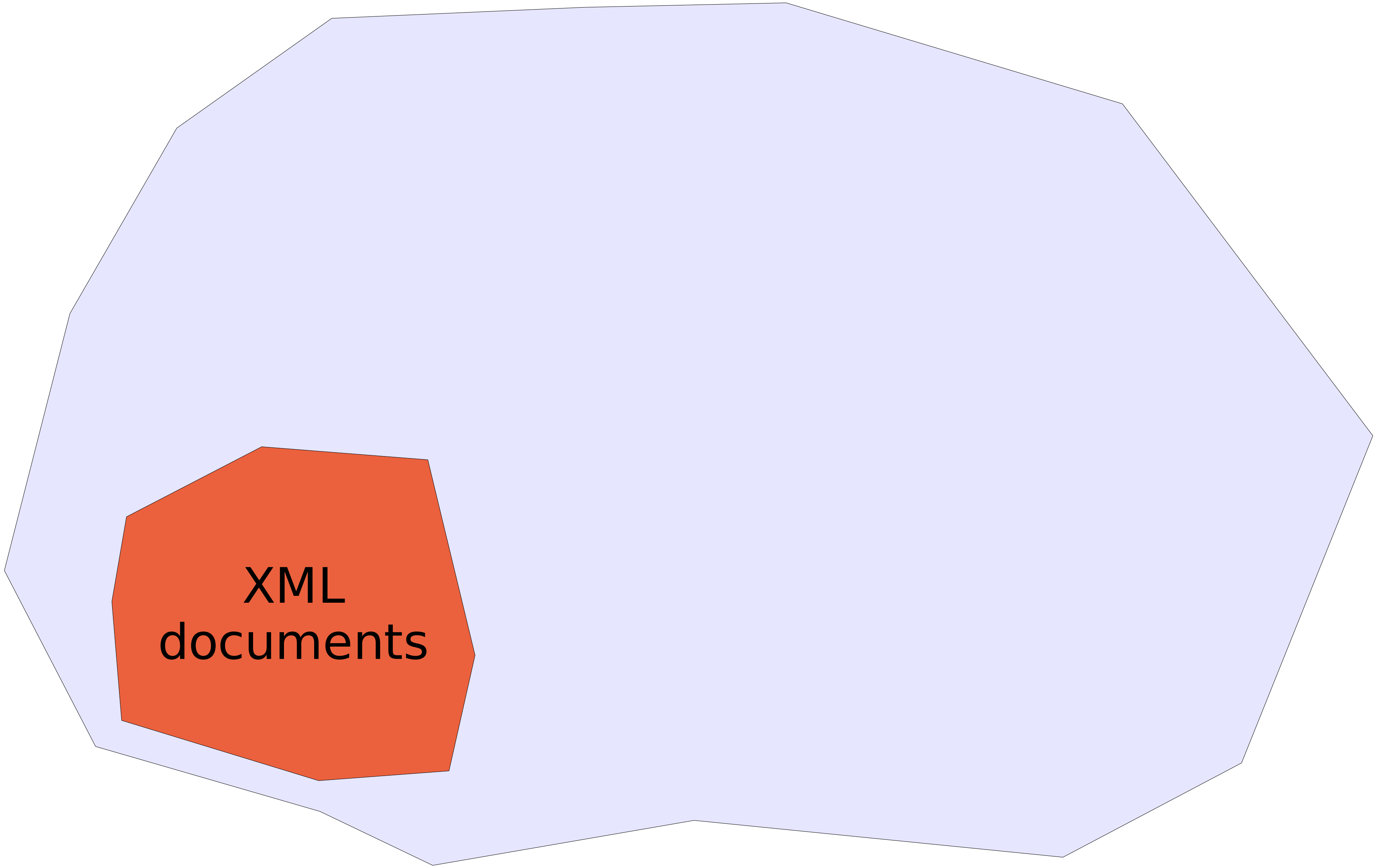
categories of evolution



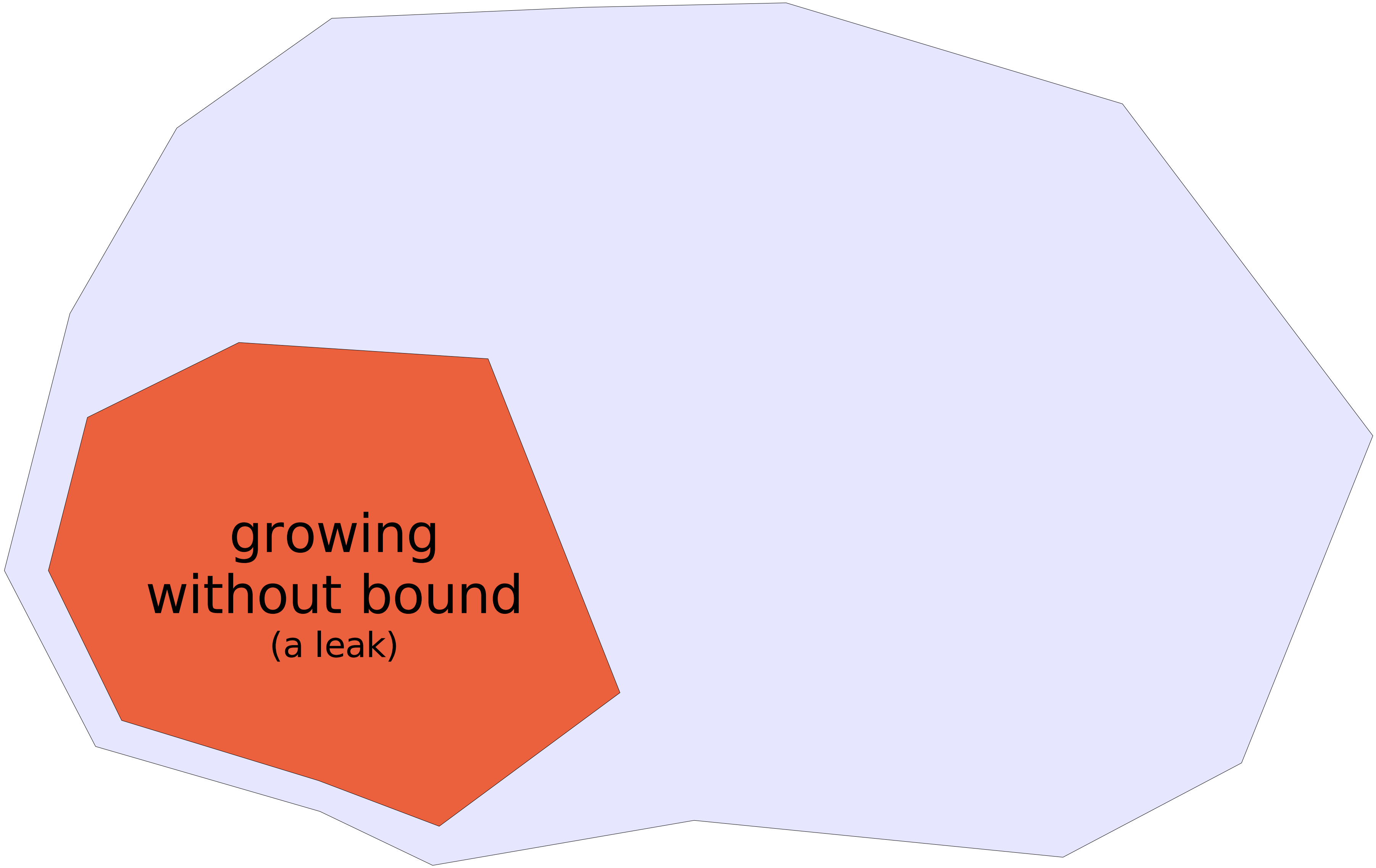
categories of **region** evolution



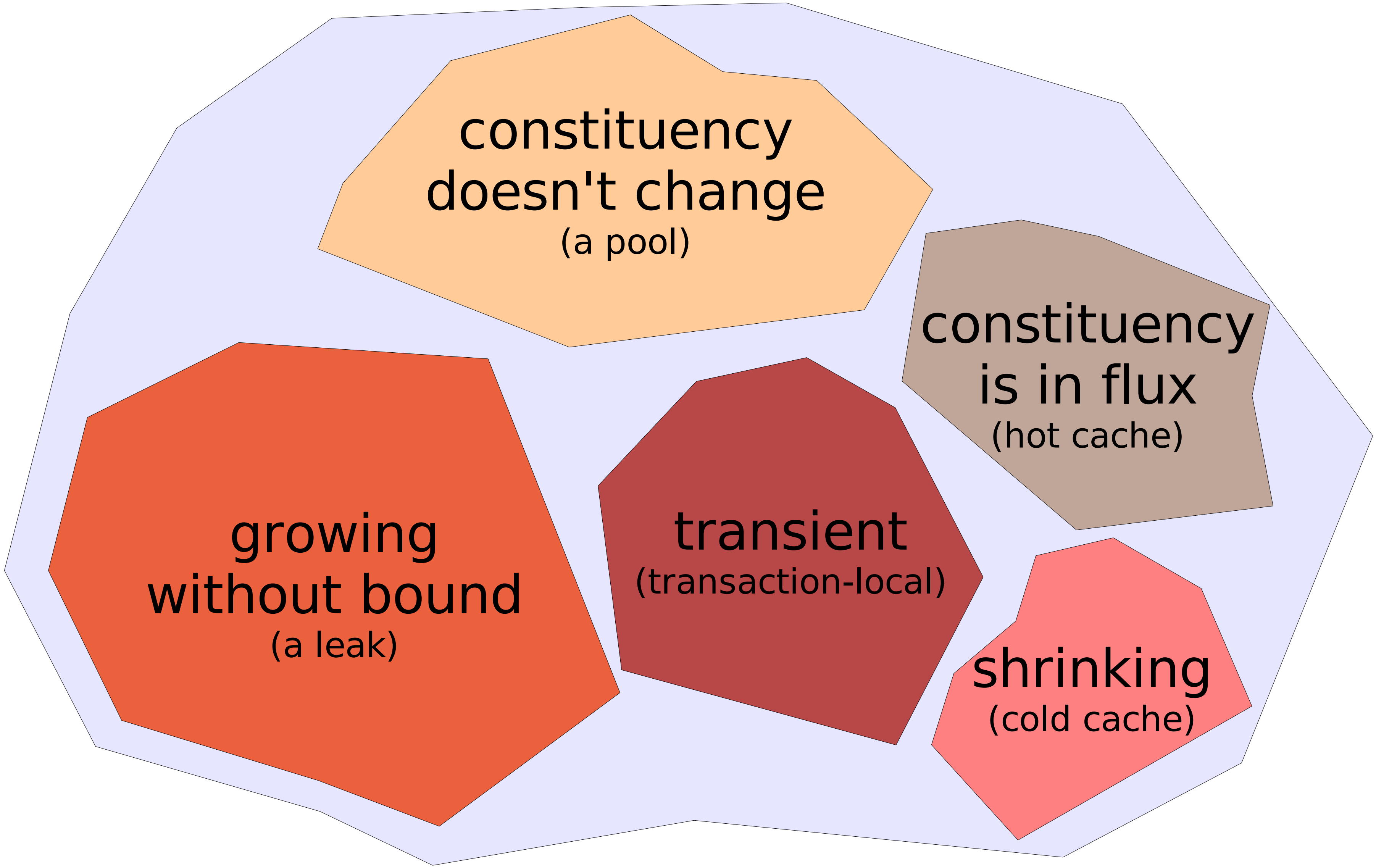
categories of region evolution



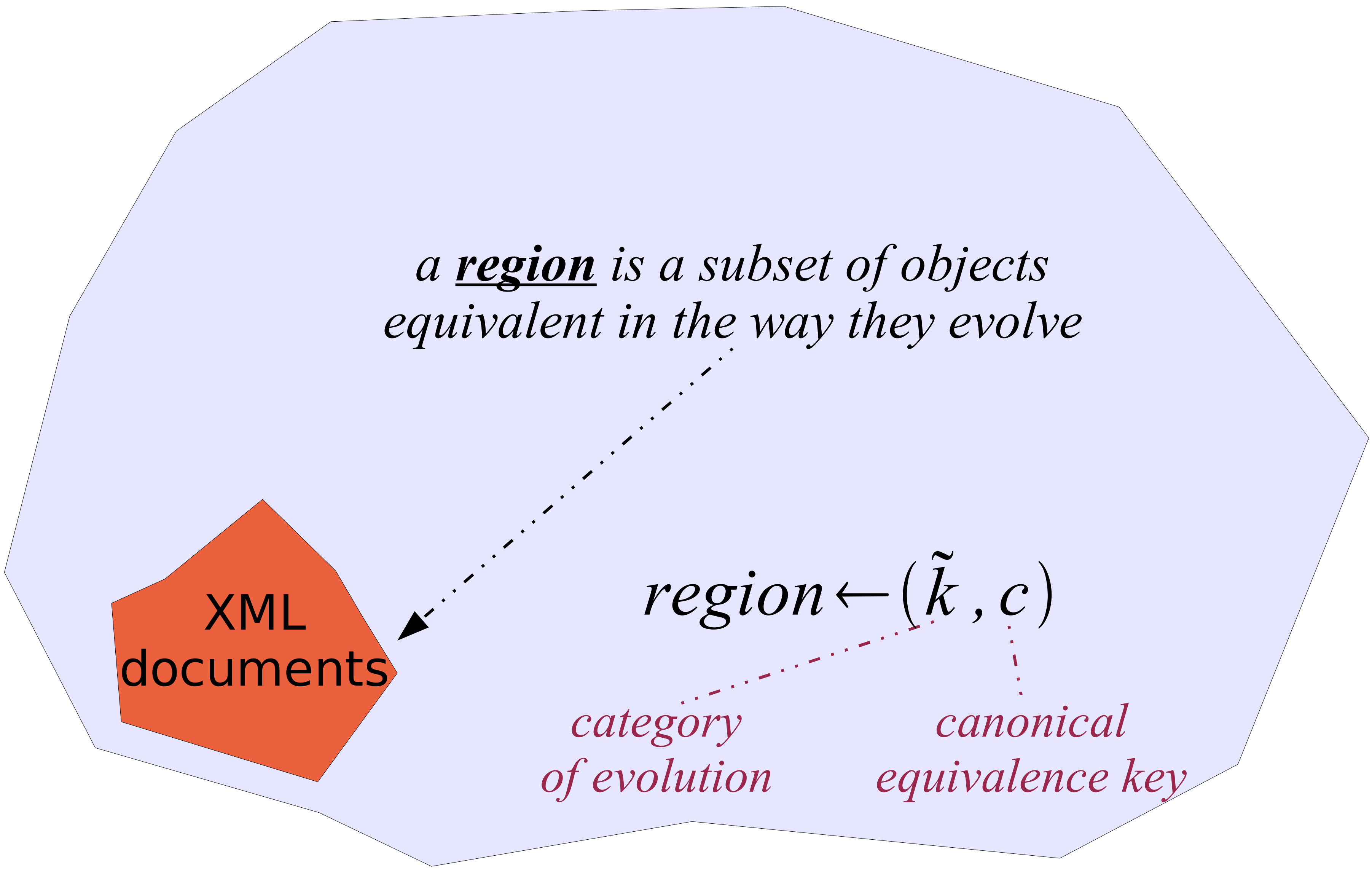
categories of region evolution



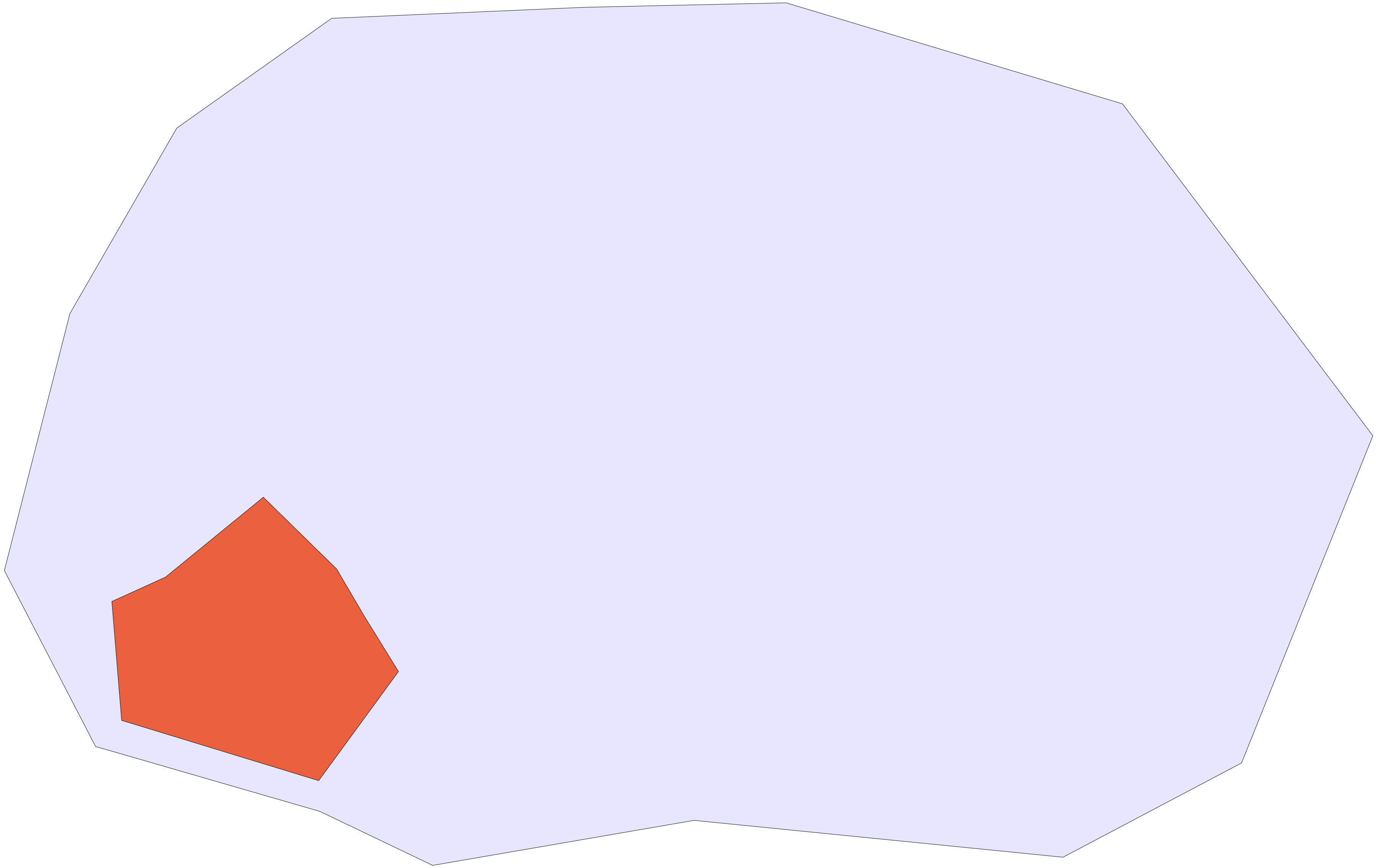
categories of region evolution



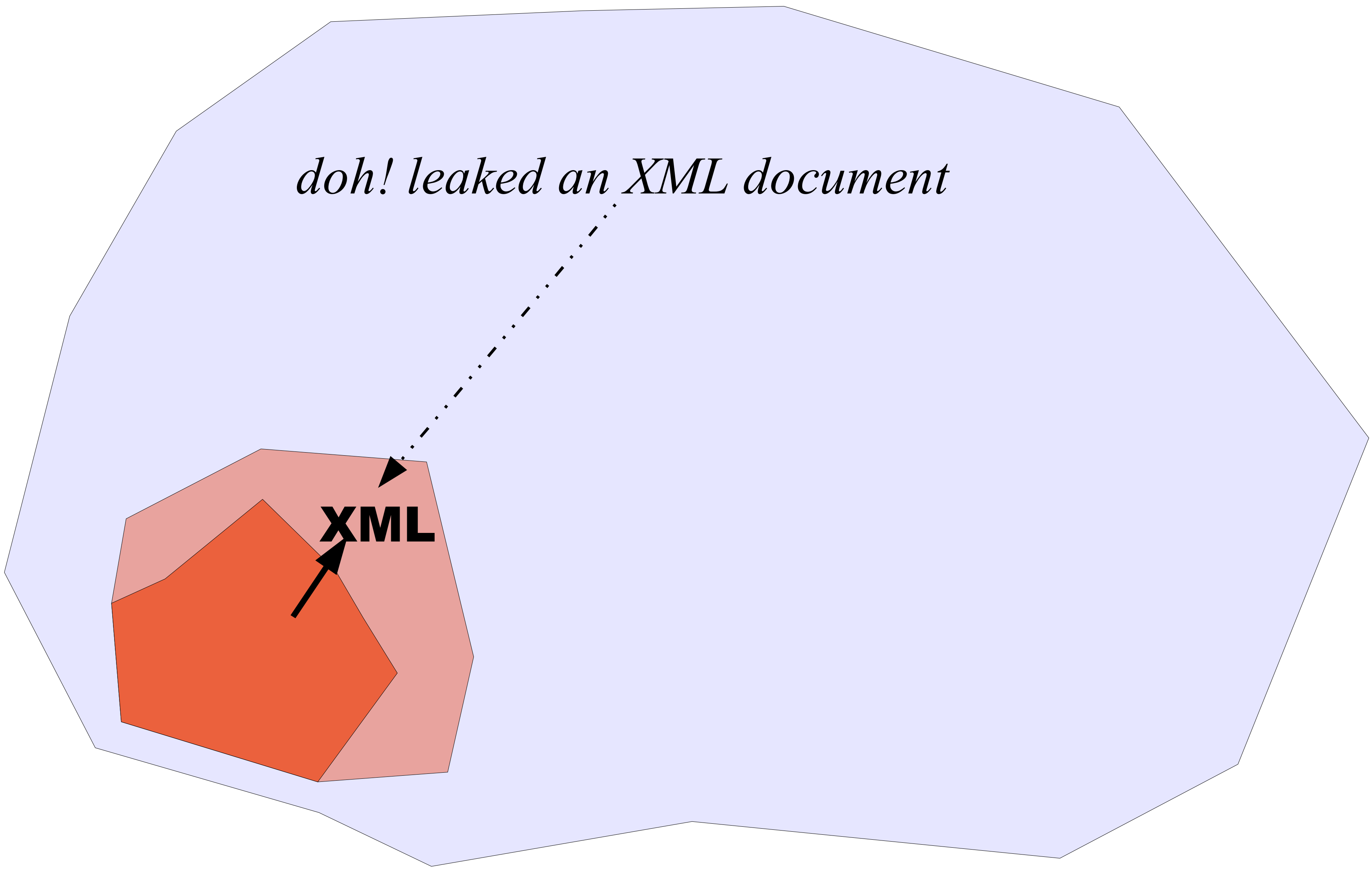
regions as equivalence clases



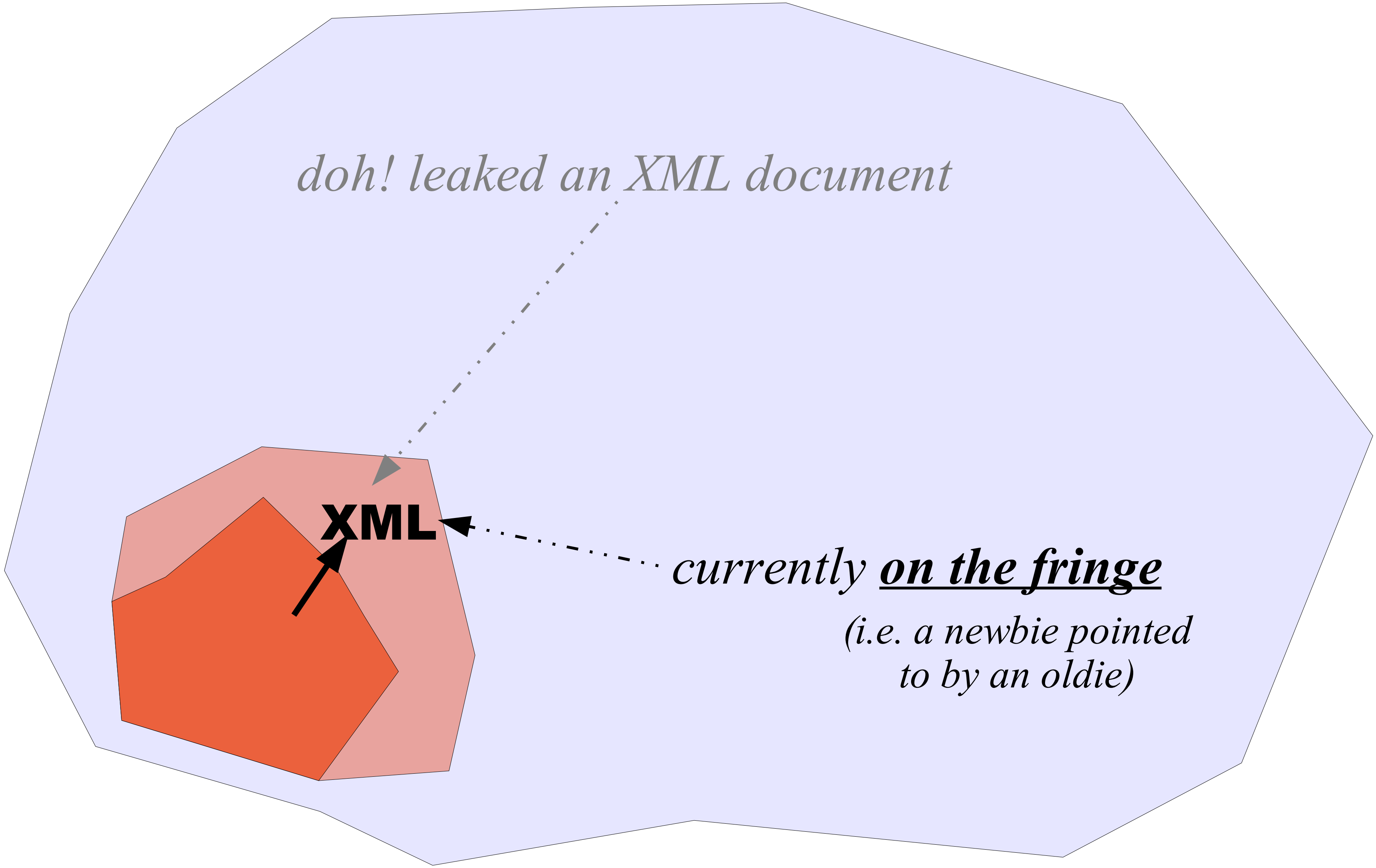
when a region grows...



when a region grows...

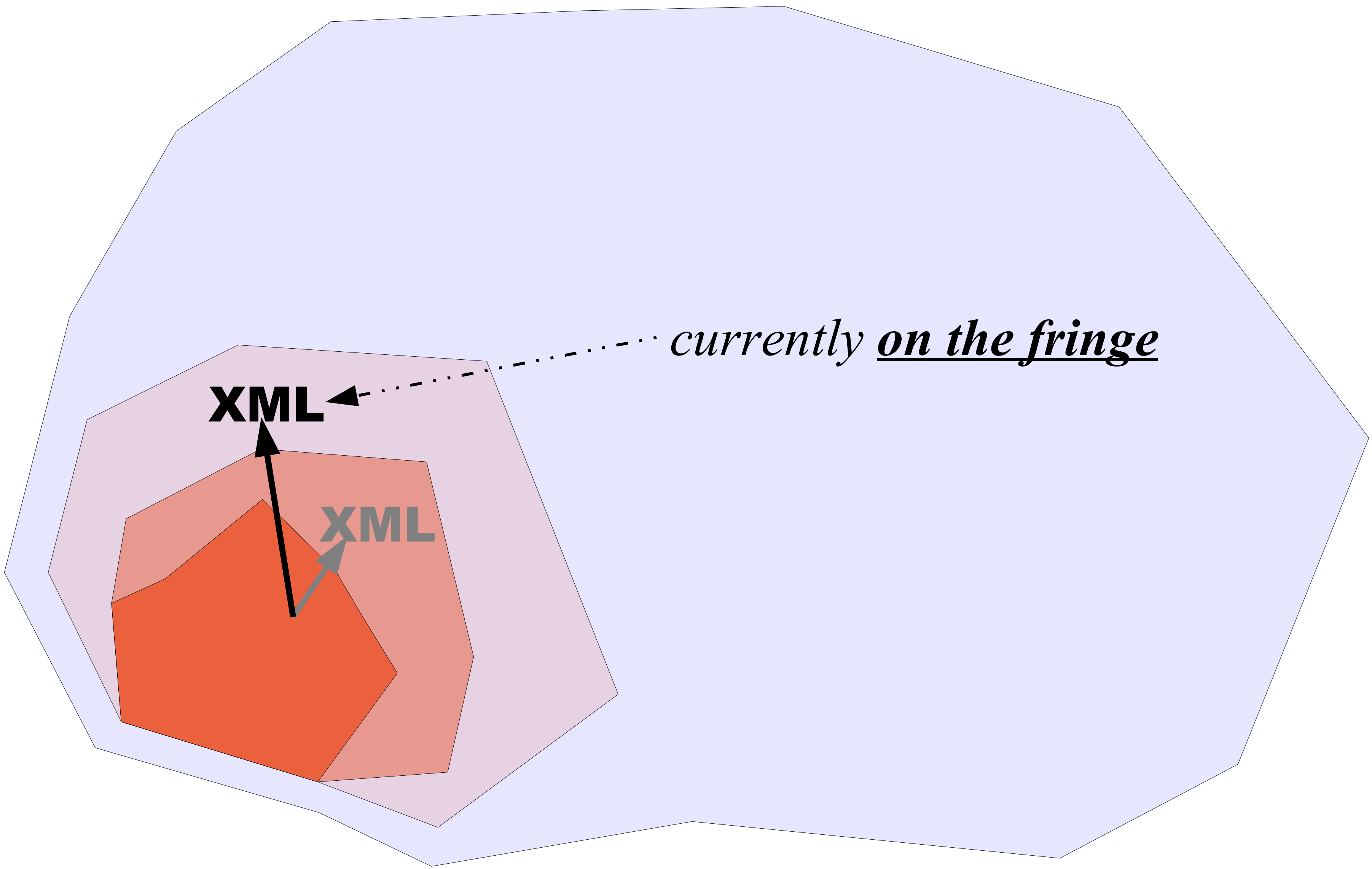


when a region grows...



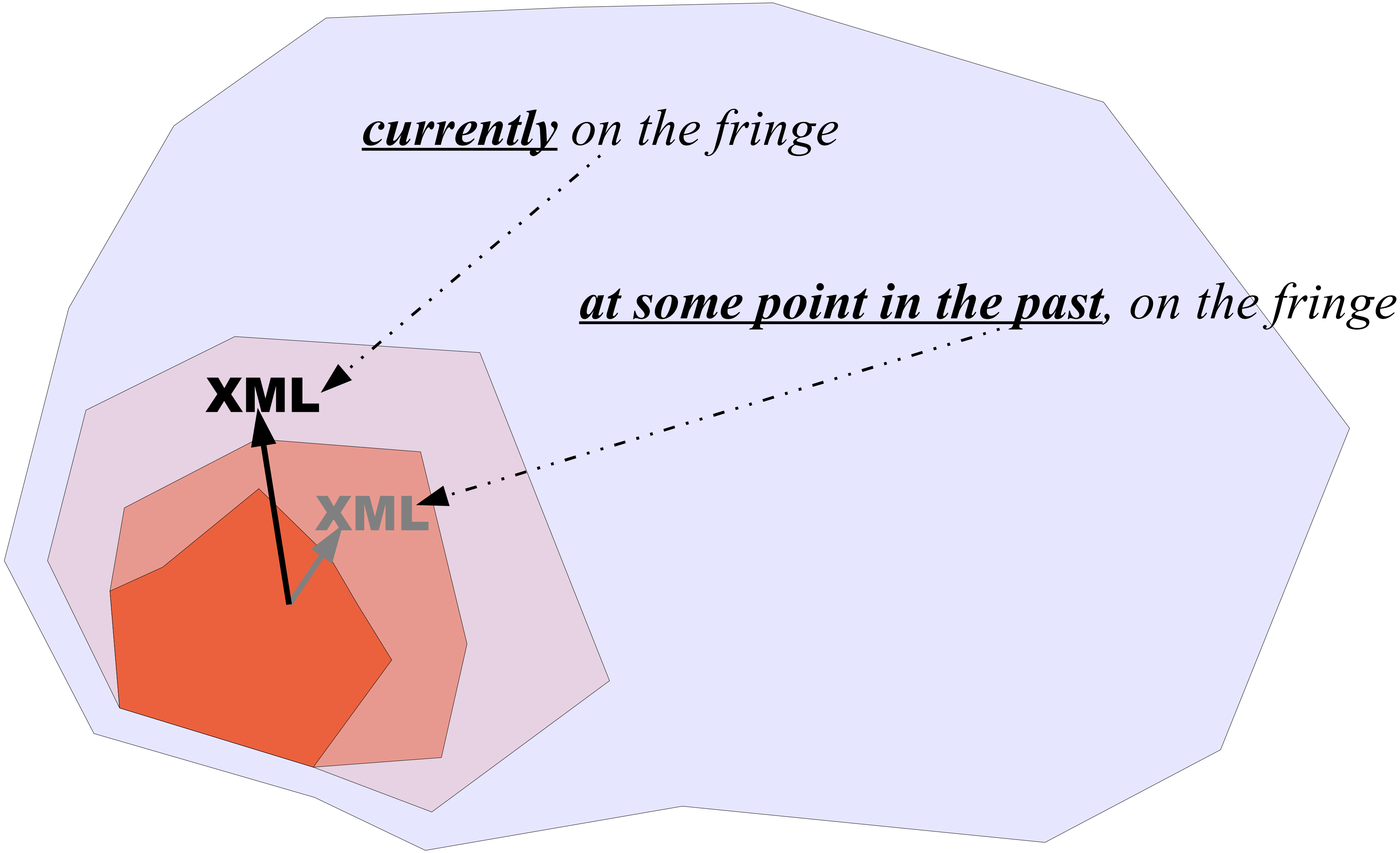
when a region grows...

we can observe a **fringe**

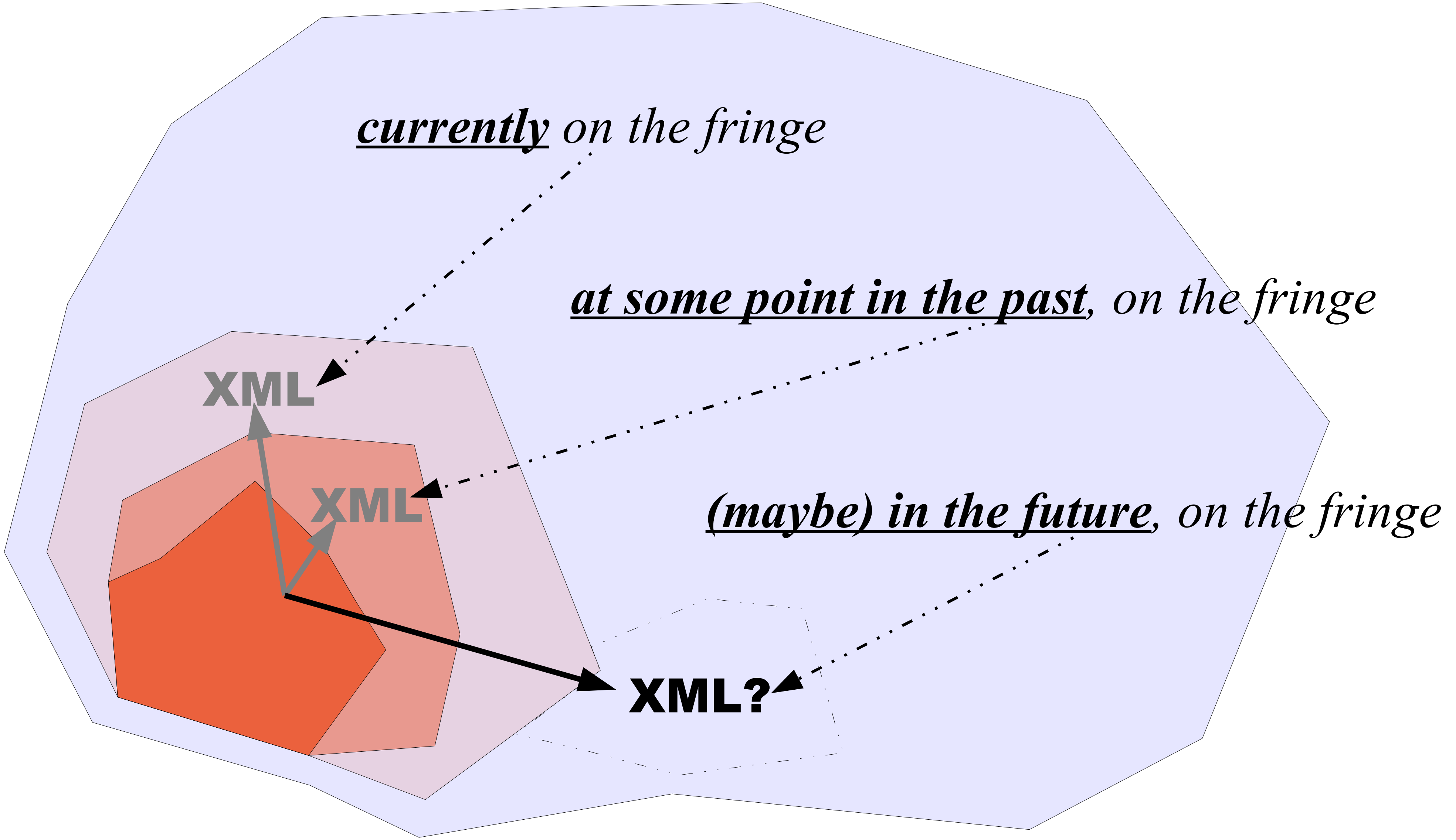


for each region, we can also

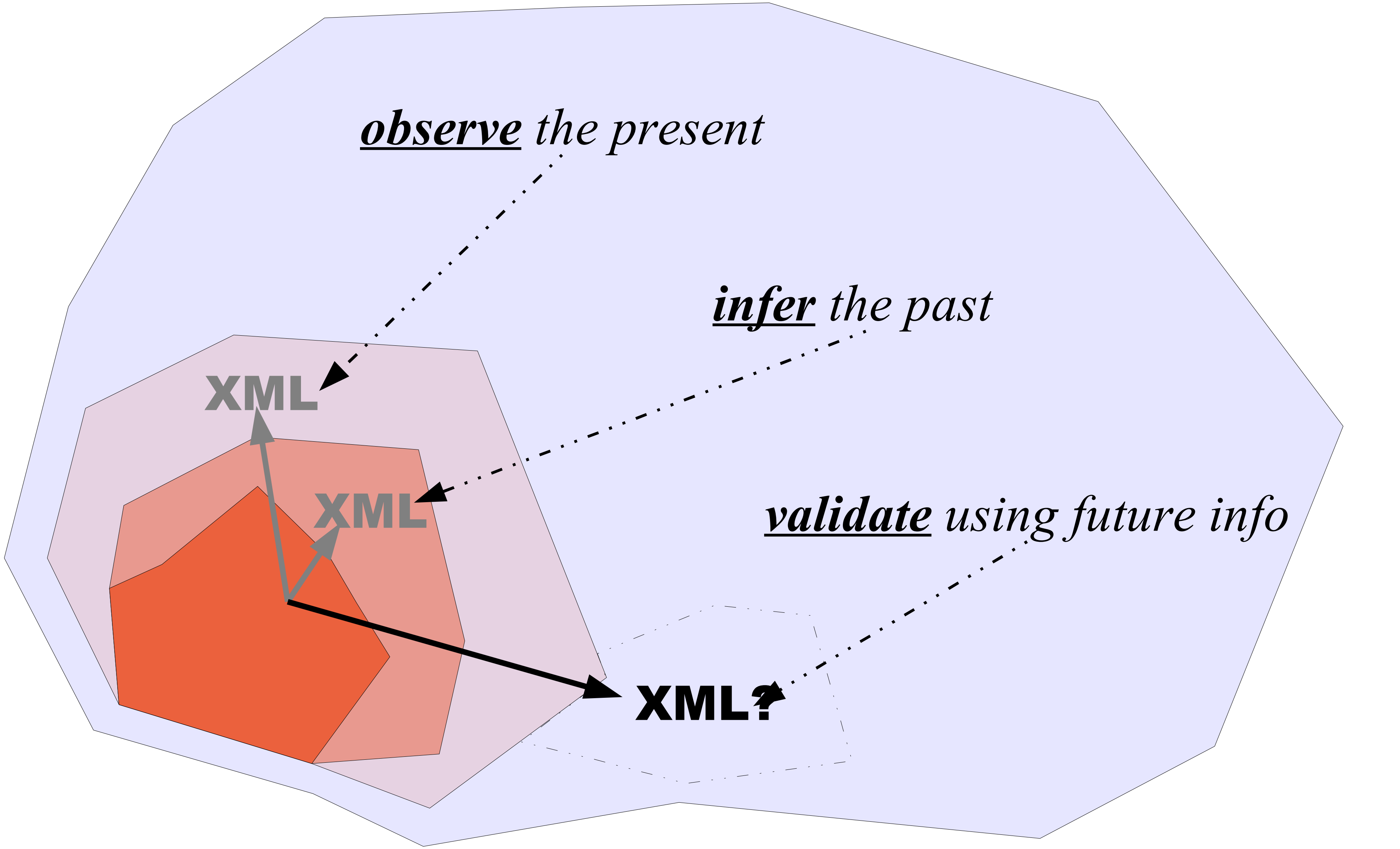
infer a **historic fringe**



finally, verify that the region evolves as expected



three tasks of evolution analysis



a way to diagnose heap evolution, in production

- **collect** a few heap snapshots
- **observe** what *is* on the fringe
  - yields a set of “seed” region keys
- **infer** what *was* on the fringe
  - yields regions populated based on region key equality
- **validate by adaptive tracing**
  - generate a set of *change detectors* that monitor violations or confirmations of a region's category of evolution
  - periodically execute a detector to refine category, set of change detectors, and quantification of how evolution is progressing

how do we implement all that?

- what's a region key?
  - a tuple of features that summarize that region's evolution
  - each object gets a key, set of canonical keys is the set of regions
- can we avoid presenting, tracking every region?
  - yup! a mixture model reduces from millions of regions to a handful
- what's a region change detector?
  - a short path traversal of the program's running heap that sees if additions, removals, or internal relinking of a region has occured
- can we avoid modeling every region?
  - use the historic fringe to identify and model only the subsets of the reference graph likely to evolve in ways the analysis cares about

how do we implement all that?

- what's a region key?
  - a tuple of features that summarize that region's evolution
  - each object gets a key, set of canonical keys is the set of regions
- can we avoid presenting, tracking every region?
  - yup! a mixture model reduces from millions of regions to a handful
- what's a region change detector?
  - a short path traversal of the program's running heap that sees if additions, removals, or internal relinking of a region has occured
- can we avoid modeling every region?
  - use the historic fringe to identify and model only the subsets of the reference graph likely to evolve in ways the analysis cares about

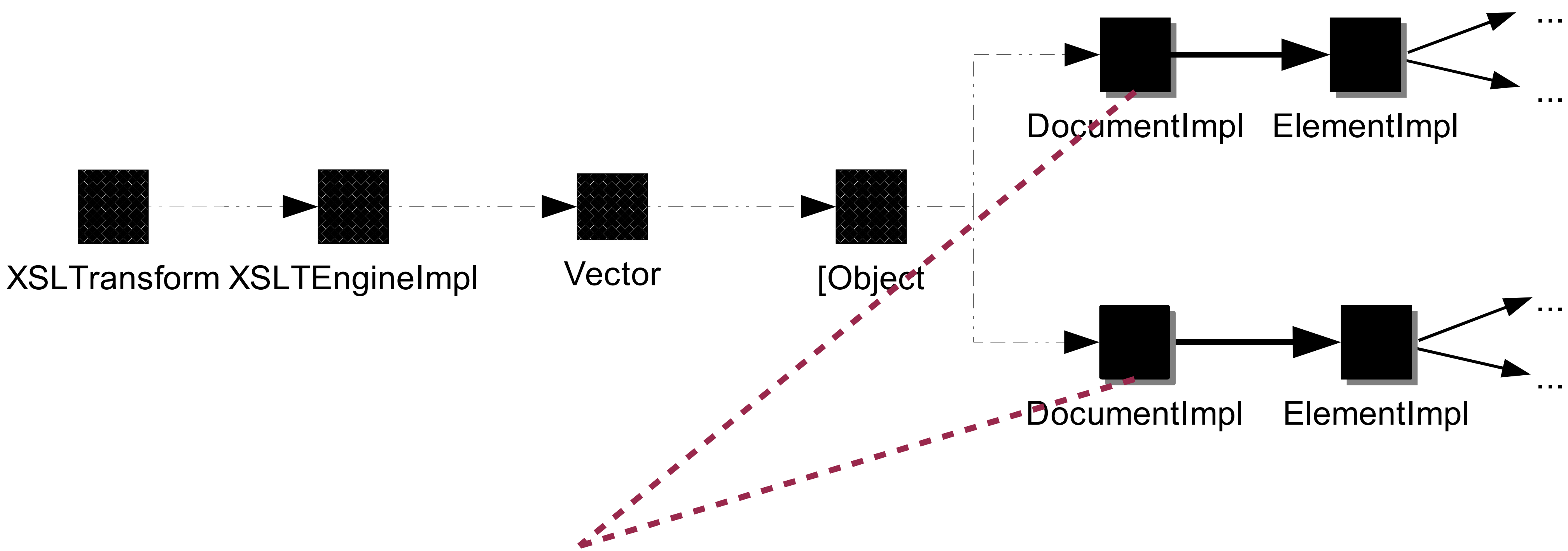
how do we implement all that?

- what's a region key?
  - a tuple of features that summarize that region's evolution
  - each object gets a key, set of canonical keys is the set of regions
- can we avoid presenting, tracking every region?
  - yup! a mixture model reduces from millions of regions to a handful
- what's a region change detector?
  - a short path traversal of the program's running heap that sees if additions, removals, or internal relinking of a region has occured
- can we avoid modeling every region?
  - use the historic fringe to identify and model only the subsets of the reference graph likely to evolve in ways the analysis cares about

how do we implement all that?

- what's a region key?
  - a tuple of features that summarize that region's evolution
  - each object gets a key, set of canonical keys is the set of regions
- can we avoid presenting, tracking every region?
  - yup! a mixture model reduces from millions of regions to a handful
- what's a region change detector?
  - a short path traversal of the program's running heap that sees if additions, removals, or internal relinking of a region has occured
- can we avoid modeling every region?
  - use the historic fringe to identify and model only the subsets of the reference graph likely to evolve in ways the analysis cares about

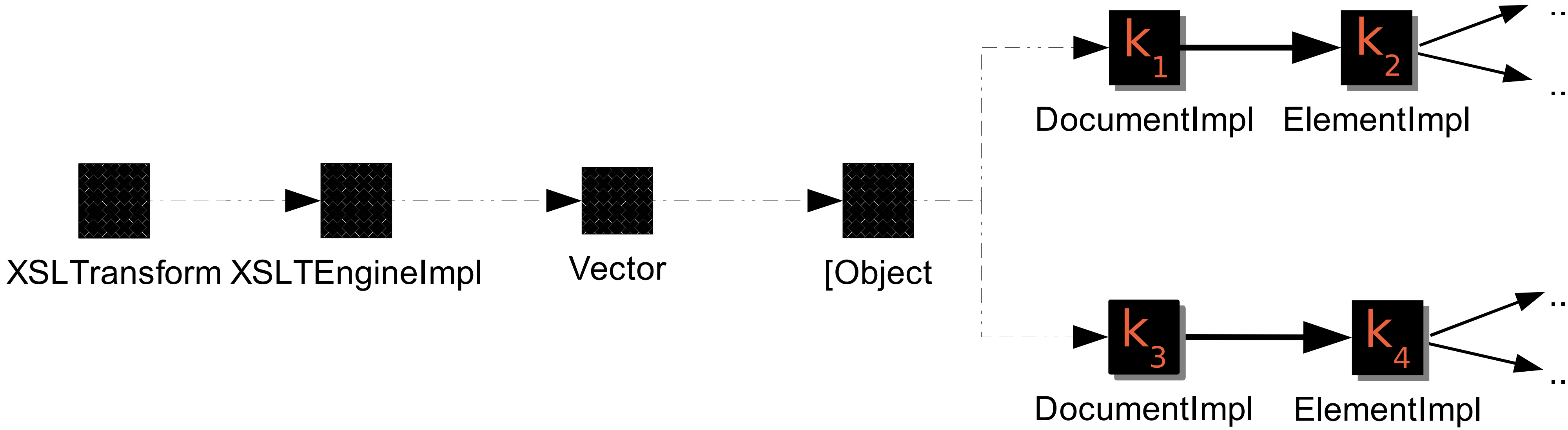
when are two objects in the same region?



recall that we're leaking XML documents

(and so these two dudes are on the historic fringe)

when are two objects in the same region?

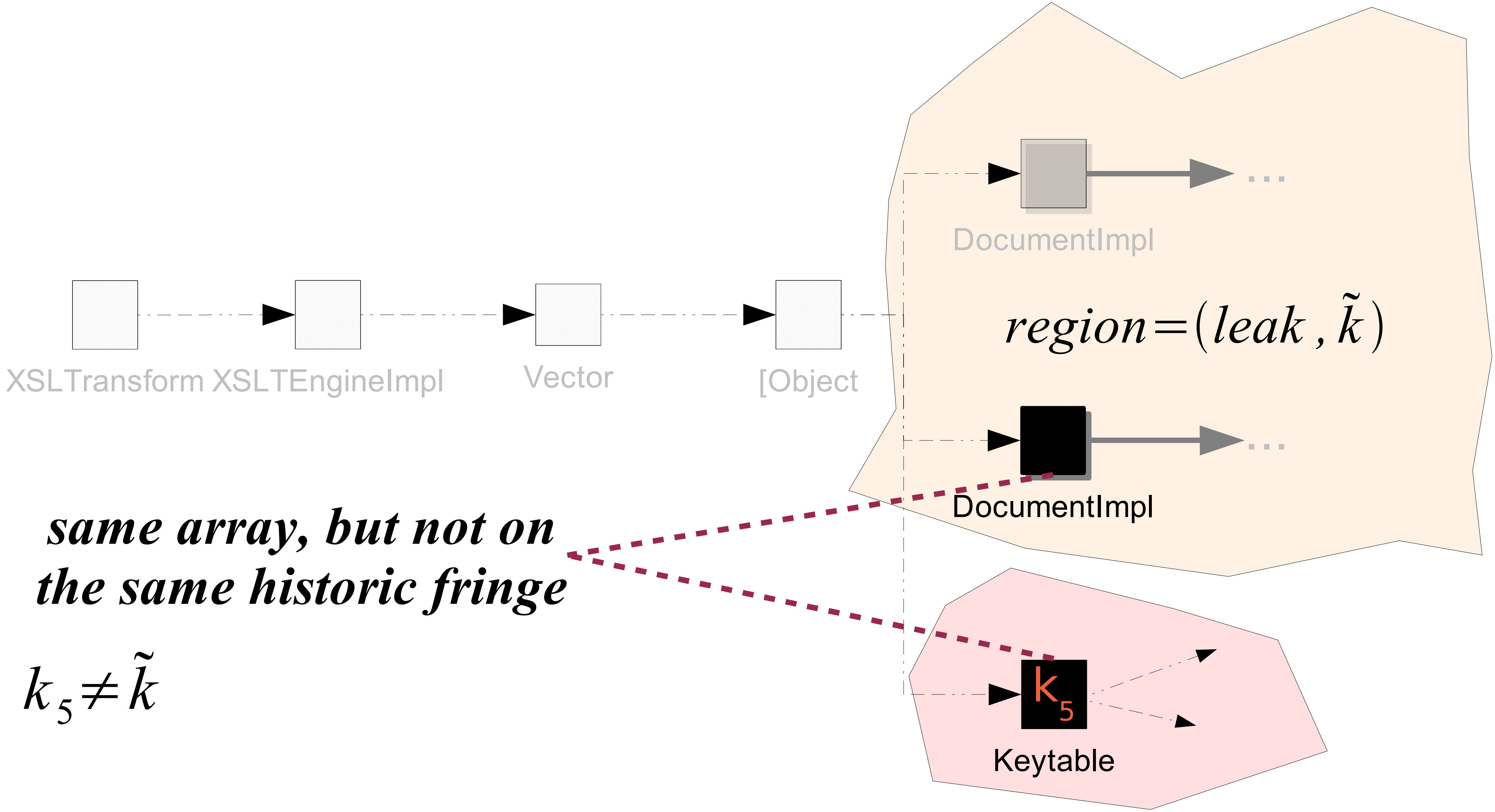


all objects "below" the historic fringe have equal region keys

(each DocumentImpl is a proxy for its dominated evolution)

$$k_1 = k_2 = k_3 = k_4 = \tilde{k}$$

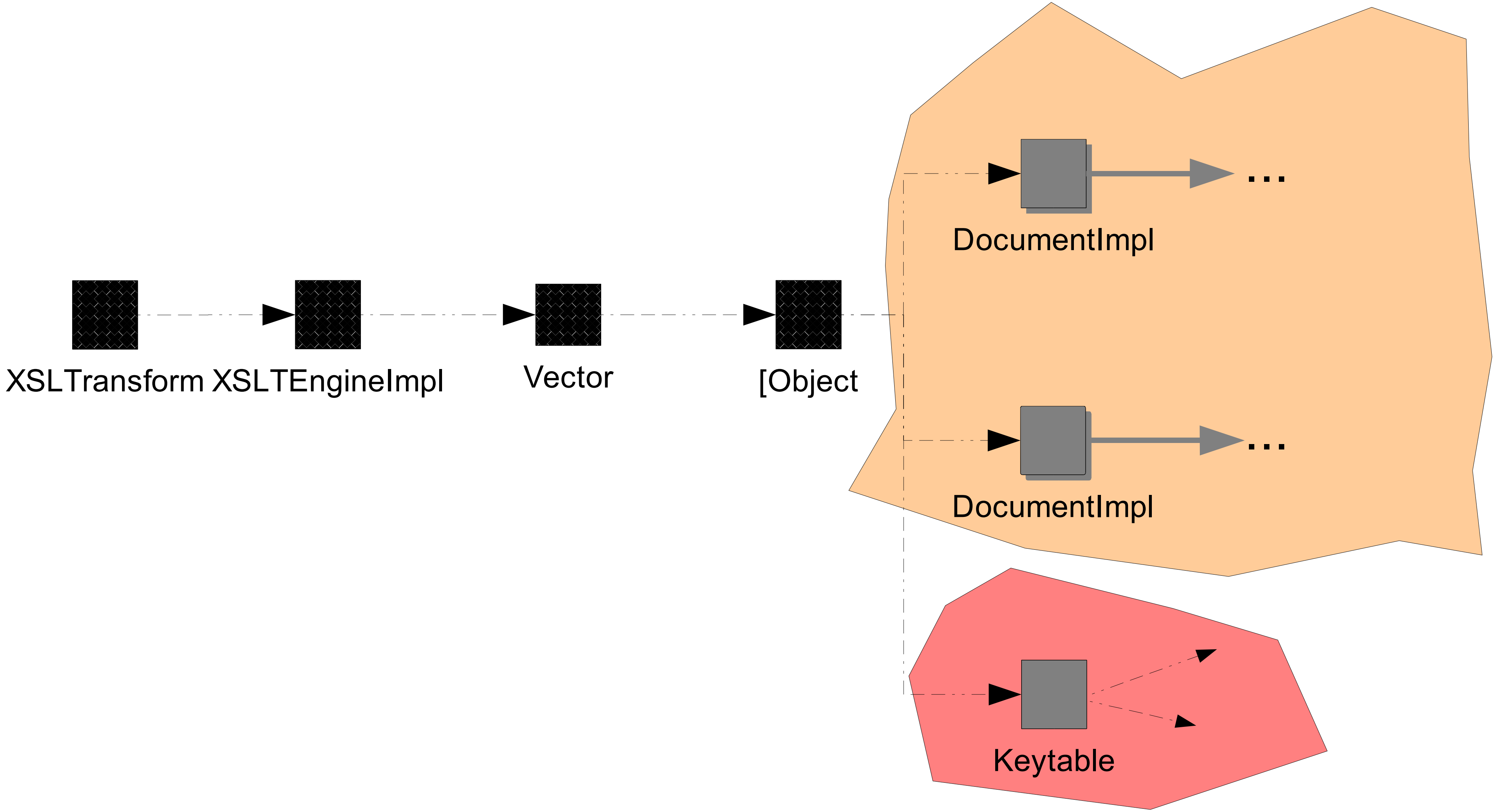
these two objects have different keys



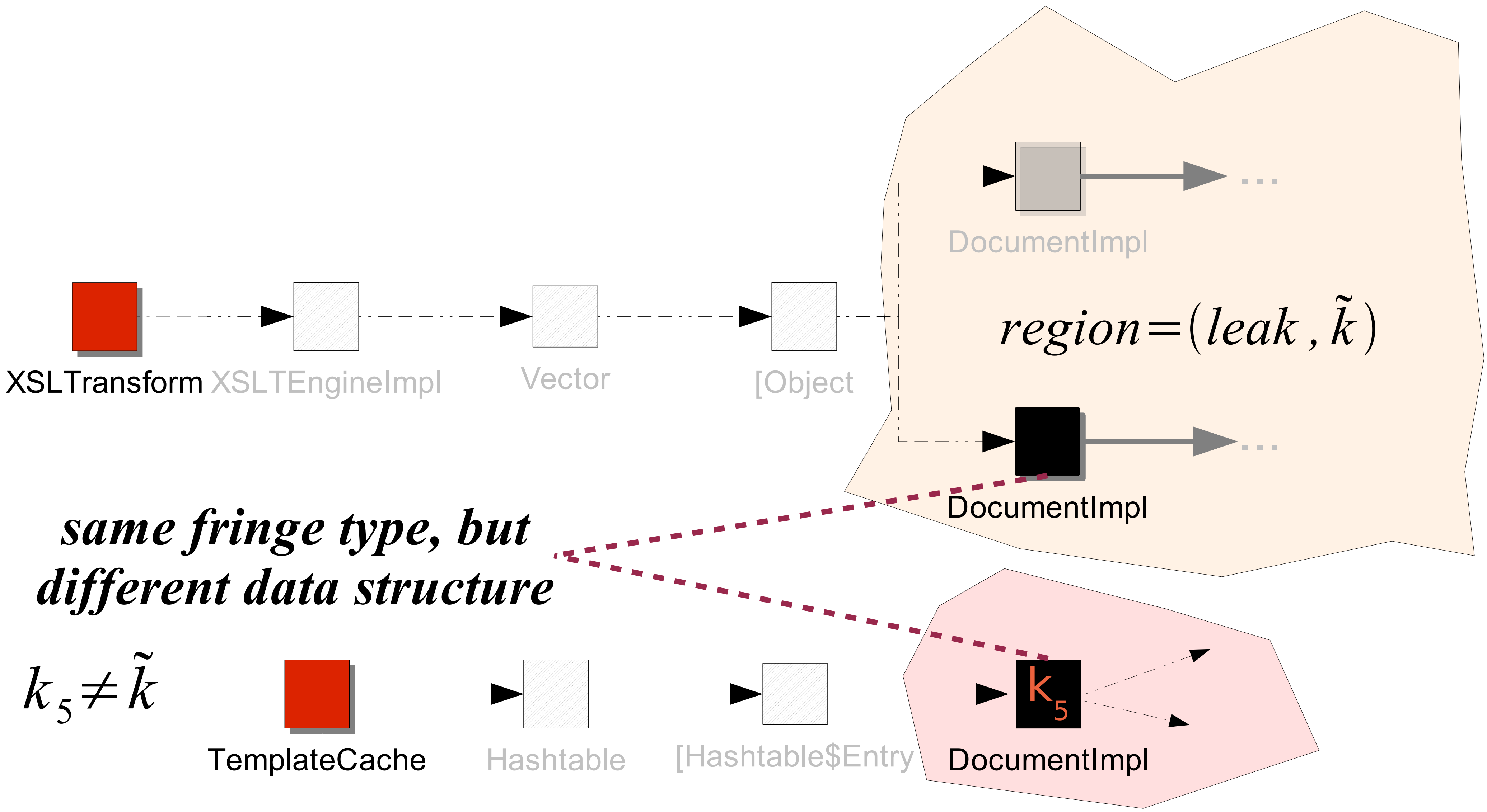
same array, but not on the same historic fringe

$$k_5 \neq \tilde{k}$$

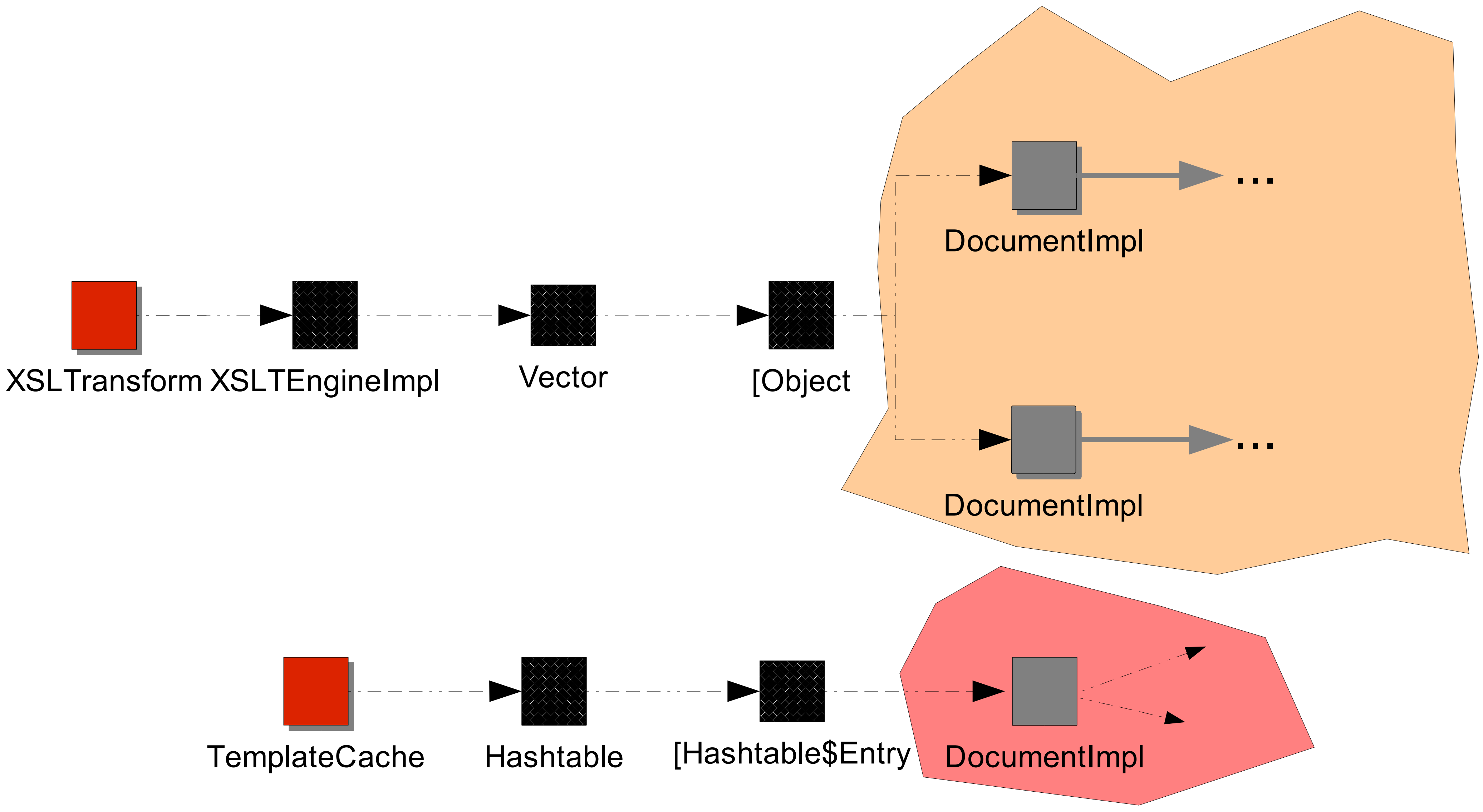
feature #1:
historic fringe datatype



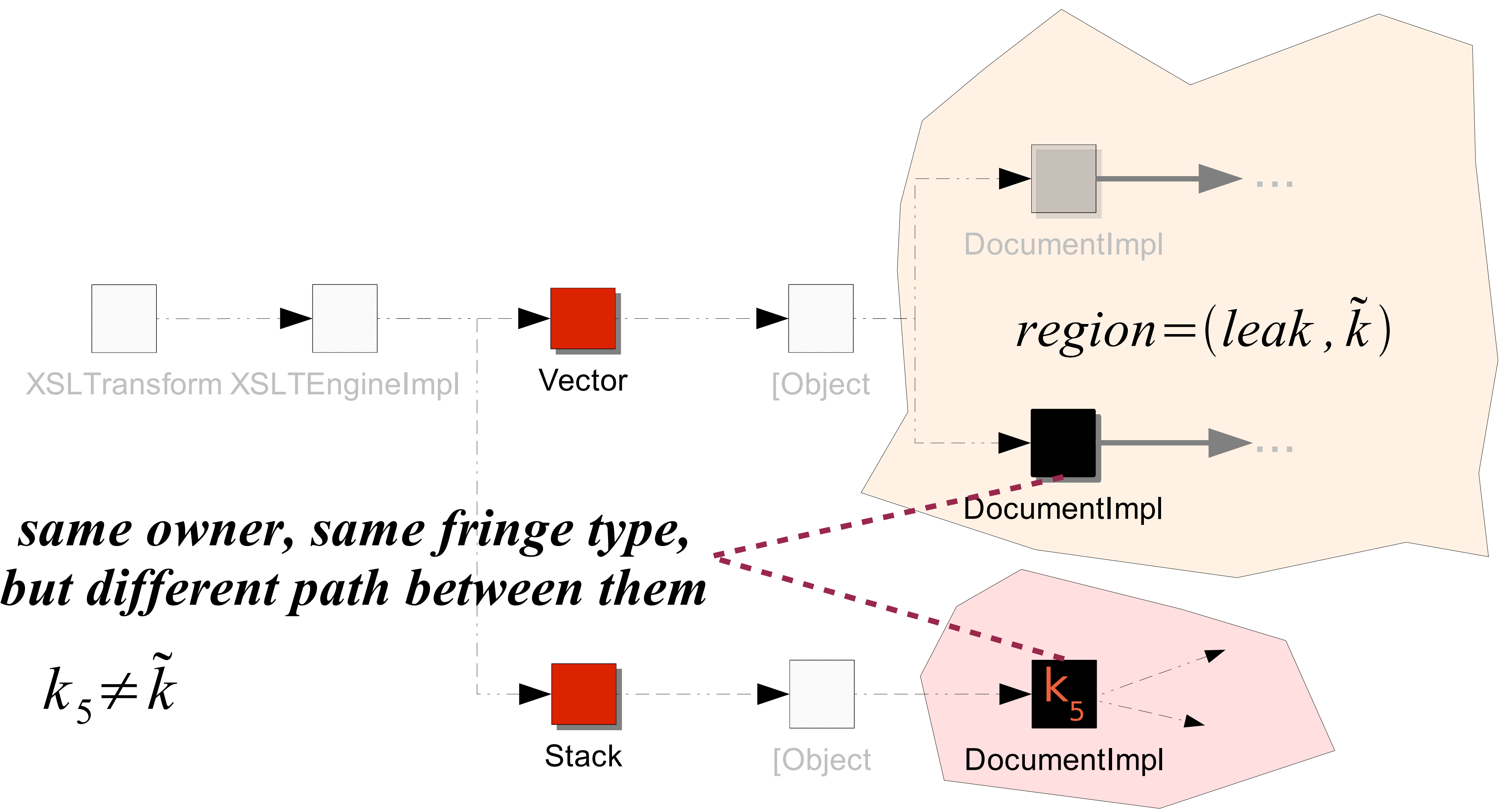
these two objects also have different keys



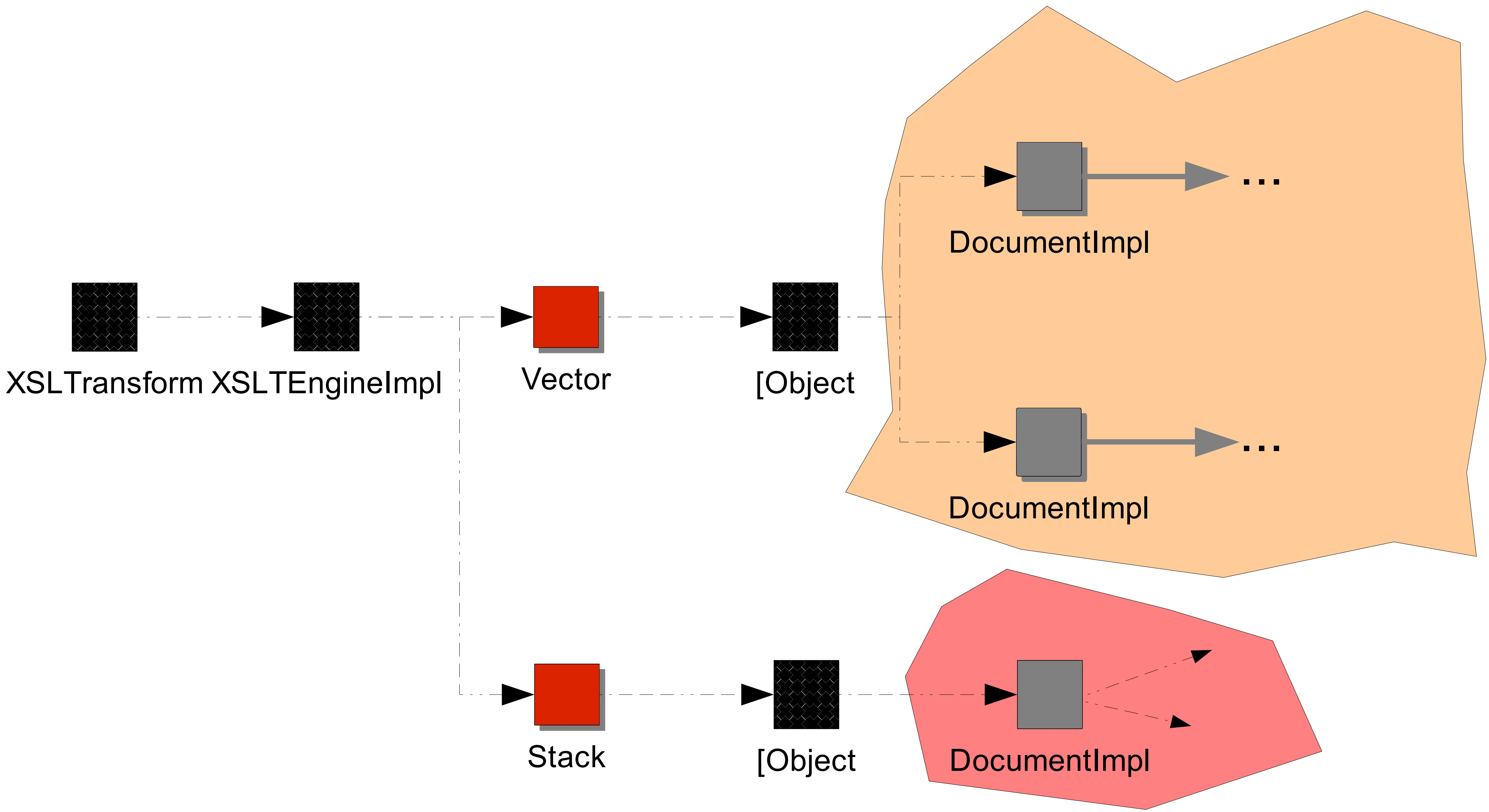
feature #2:
root data structure



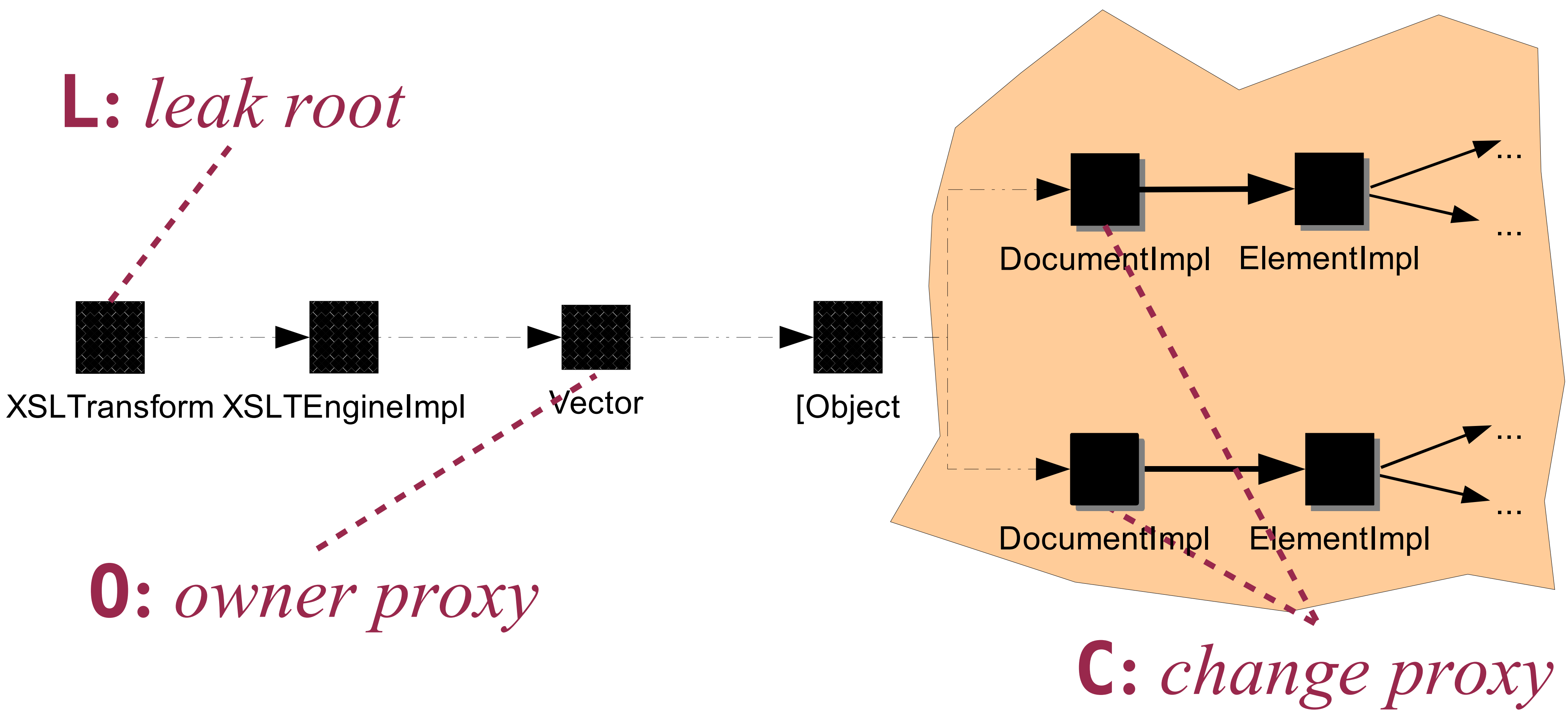
these two objects also have different keys



feature #3:
owning container



to each object, a region key tuple



$$\tilde{k} = [L, O, typeof(C)]$$

how do we implement all that?

- what's a region key?
  - a tuple of features that summarize that region's evolution
  - each object gets a key, set of canonical keys is the set of regions
- can we avoid presenting, tracking every region?
  - yup! a mixture model reduces from millions of regions to a handful
- what's a region change detector?
  - a short path traversal of the program's running heap that sees if additions, removals, or internal relinking of a region has occured
- can we avoid modeling every region?
  - use the historic fringe to identify and model only the subsets of the reference graph likely to evolve in ways the analysis cares about

how do we avoid presenting and tracking every region?

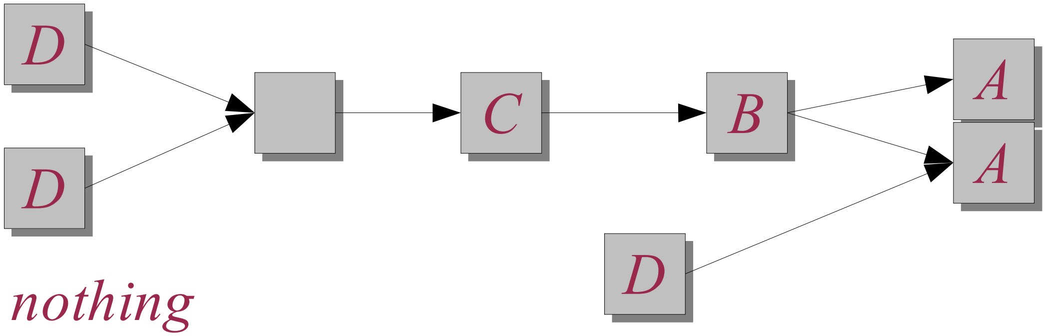
Define leak root metric, **`LRM=B∘M∘G`**, such that each leaking region has one **`o`** with **`LRM(o)>0`**, and few **`o`**'s have **`LRM(o)>`**Θ.

- **`B:`** eight binary rules to rule out impossible
  - (be Sherleak Holmes!)
  - narrow from a **million** to a **hundred**
- **`M:`** mixture model to rank the remaining
  - narrow from a **hundred** to **tens**
- **`G:`** global fixpoint to ensure uniqueness
  - narrow from **tens** to a **handful** of highly-ranked leak roots

B: ruling out the impossible

(using structural information)

	# objects	fraction of objects remaining			
		-A	-A-B	-A-B-C	-A-B-C-D
phone company	267,956	0.67	0.59	0.09	0.06
IDE	350,136	0.61	0.55	0.09	0.07
brokerage1	838,912	0.65	0.62	0.07	0.03
brokerage2	1,015,112	0.71	0.70	0.02	0.01
credit bureau	1,320,953	0.60	0.56	0.11	0.08



A. objects pointing to nothing aren't very interesting

D. objects which don't uniquely own anything also aren't interesting

B. arrays themselves don't leak
(but their dominating containers might)

C. ibid for objects not at the head of a single-entry region

B: ruling out the impossible

(using temporal information)

			# objects remaining			
	# objects	-structural	-E	-E-F	-E-F-G	all told
phone company	267,956	16,346	73	73	72	29
IDE	350,136	25,653	99	99	29	10
brokerage1	838,912	26,291	97	82	81	67
brokerage2	1,015,112	12,020	102	102	64	17
credit bureau	1,320,953	160,900	579	519	518	242

E. *ignore structures that contain only old or only new objects*
(e.g. an already-primed pool)

F. *structures that contain only new arrays are borinng*
(theres nothing new in those arrays)

G. *ignore structures with no overlap in datatypes over time*

H. *structures that contain no objects on the fringe are safe to ignore*

LRM=B∘M∘G, for example

(before applying the bug fixes)

	# constituents	size rank	**LRM(o)**
com/.../EventNotifier	377276	1	0.895
com/.../FormProperties	270	157	0.658
com/.../XslTemplateCollection	32	841	0.463
com/.../VerifySignonScenario	18	1050	0.420

of the highest-ranked candidate roots,
the top two indeed leak

(from 1,015,112 live objects)

LRM=B∘M∘G, for example

(after applying the bug fixes)

	# constituents	size rank	**LRM(o)**
com/websphere/AlarmThread	399	130	0.322
com/.../ContextModel	837	86	0.266
com/websphere/PoolManager	391	134	0.260
com/websphere/PoolEpm	385	137	0.254

after fixing the leak,
there are no stand-out candidates

(from 779,540 live objects)

how do we implement all that?

- what's a region key?
  - a tuple of features that summarize that region's evolution
  - each object gets a key, set of canonical keys is the set of regions
- can we avoid presenting, tracking every region?
  - yup! a mixture model reduces from millions of regions to a handful
- what's a region change detector?
  - a short path traversal of the program's running heap that sees if additions, removals, or internal relinking of a region has occured
- can we avoid modeling every region?
  - use the historic fringe to identify and model only the subsets of the reference graph likely to evolve in ways the analysis cares about

detecting evolution cheaply

- a region evolves when elements are
  - added to
  - removed from
  - relinked within
- track evolution with **region change detectors**

detecting evolution cheaply

- a region evolves when elements are
  - added to
  - removed from
  - relinked within
- track evolution with **region change detectors**
- a detector is a tuple `[R,H,T,B,P,M]`
  - `R:` region to detect changes in
  - `H,T`: the head and tail of a short, bounded-size traversal
  - `B`: a sample bias
  - `P`: a match precondition
  - `M`: a mutator, updates the set of existing detectors

leakbot in action

just after initial analysis

Leakbot Display Options

rank of leak	root	owner-proxy	change-proxy	# leakages	trend	tick
0.867	simpleleaker class object	Vector object	by type:Boolean	93	grower	grower
0.867	simpleleaker class object	Vector object	by type:Integer	93	grower	grower
0.838	simpleleaker class object	Vector object	by type:Character	84	grower	grower
0.57	simpleleaker class object	LinkedList$Entry object	by type:LinkedList$Entry	1	not enough info...	not enough info...
0.57	simpleleaker class object	LinkedList$Entry object	by type:LinkedList$Entry	1	not enough info...	not enough info...

Publishing region descriptors to JVM.
Enabling JVM-side region change detection.
Enabling JVM-side region change detection.

over time...

about one minute later

Leakbot Display Options

rank of leak	root	owner-proxy	change-proxy	# leakages	trend	tick
21.484	simpleleaker class object	Vector object	by type:Integer	7838	grower	grower
1.558	simpleleaker class object	Vector object	by type:Boolean	593	alternater	flatliner
1.114	simpleleaker class object	LinkedList$Entry object	by type:LinkedList$Entry	4	grower	grower
1.114	simpleleaker class object	LinkedList$Entry object	by type:LinkedList$Entry	4	grower	grower
0.687	simpleleaker class object	Vector object	by type:Character	190	grower	grower

Tracking grower in jinsight.leaky.AdditionTemplate@a01711; size estimate=4.
Tracking grower in jinsight.leaky.AdditionTemplate@1ba4b54; size estimate=4.
Tracking grower in jinsight.leaky.AdditionTemplate@1ba4b54; size estimate=4.

a non-leaking region

and another few minutes...

Leakbot Display Options

rank of leak	root	owner-proxy	change-proxy	# leakages	trend	tick
27.815	simpleleaker class object	Vector object	by type:Integer	10193	grower	grower
2.365	simpleleaker class object	LinkedList$Entry object	by type:LinkedList$Entry	9	grower	grower
2.365	simpleleaker class object	LinkedList$Entry object	by type:LinkedList$Entry	9	grower	grower
1.193	simpleleaker class object	Vector object	by type:Character	359	grower	grower
0.917	simpleleaker class object	Vector object	by type:Boolean	593	flatliner	flatliner

Tracking grower in jinsight.leaky.AdditionTemplate@1ba4b54; size estimate=9.
Tracking grower in jinsight.leaky.AdditionTemplate@a01711; size estimate=9.
Tracking grower in jinsight.leaky.AdditionTemplate@a01711; size estimate=9.

is downgraded

final stuff

- analysis handles 40 million objects with 600M
- adaptive, online tracing slows app down only 2%
- can identify very slow leaks in a few minutes
- implemented as a JVMPI agent (written in C++) and an analyzer (written in Java)
- going into WebSphere and Rational Studio

final stuff

- analysis handles 40 million objects with 600M
- adaptive, online tracing slows app down only 2%
- can identify very slow leaks in a few minutes
- implemented as a JVMPI agent (written in C++) and an analyzer (written in Java)
- going into WebSphere and Rational Studio

- thanks to the team! **Bowen Alpern, Glenn Ammons, Vas Bala, Herb Derby, Todd Mummert, Darrell Reimer, Gary Sevitsky, Edith Schonberg, Harini Srinivasan, Kavitha Srinivas**
  - JIT/BCI interface for efficient bytecode-level probing (going into J9)
  - rules-based validation system (going into Rational Studio)
  - automated performance analysis (ongoing)

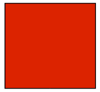
factoring out objects via heap differencing is insufficient

	“new” live instances
java/lang/String	9444
org/apache/xerces/dom/TextImpl	6810
org/apache/xerces/dom/AttrImpl	5290
java/util/Hashtable$Entry	3244
org/apache/xerces/dom/NamedNodeMapImpl	2713
org/apache/xerces/dom/ElementImpl	2123
...	...
org/apache/xerces/dom/DocumentImpl	27

an atom of a leak

(every leaking operation leaks **lots** of these objects)

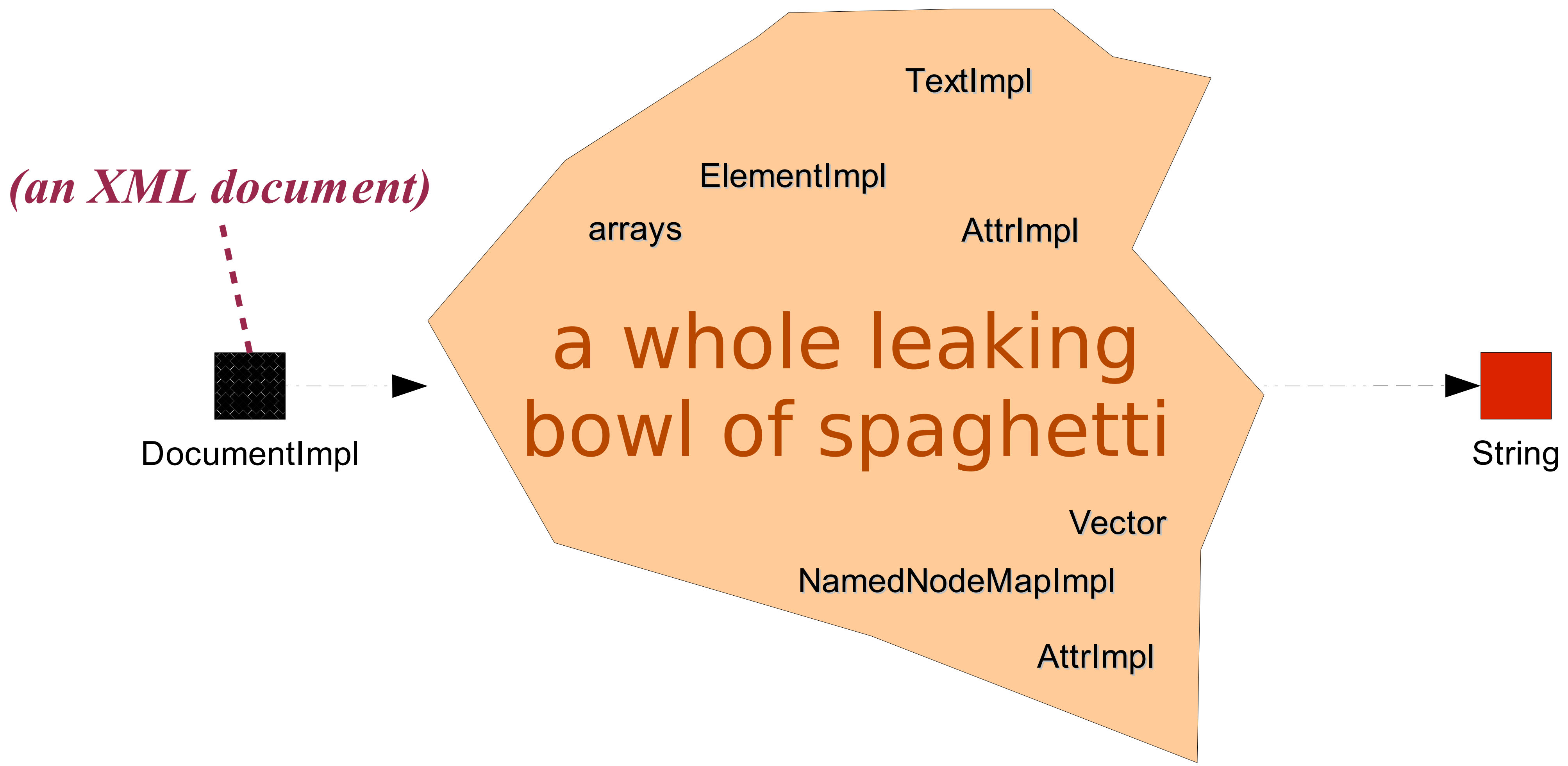
you're leaking Strings



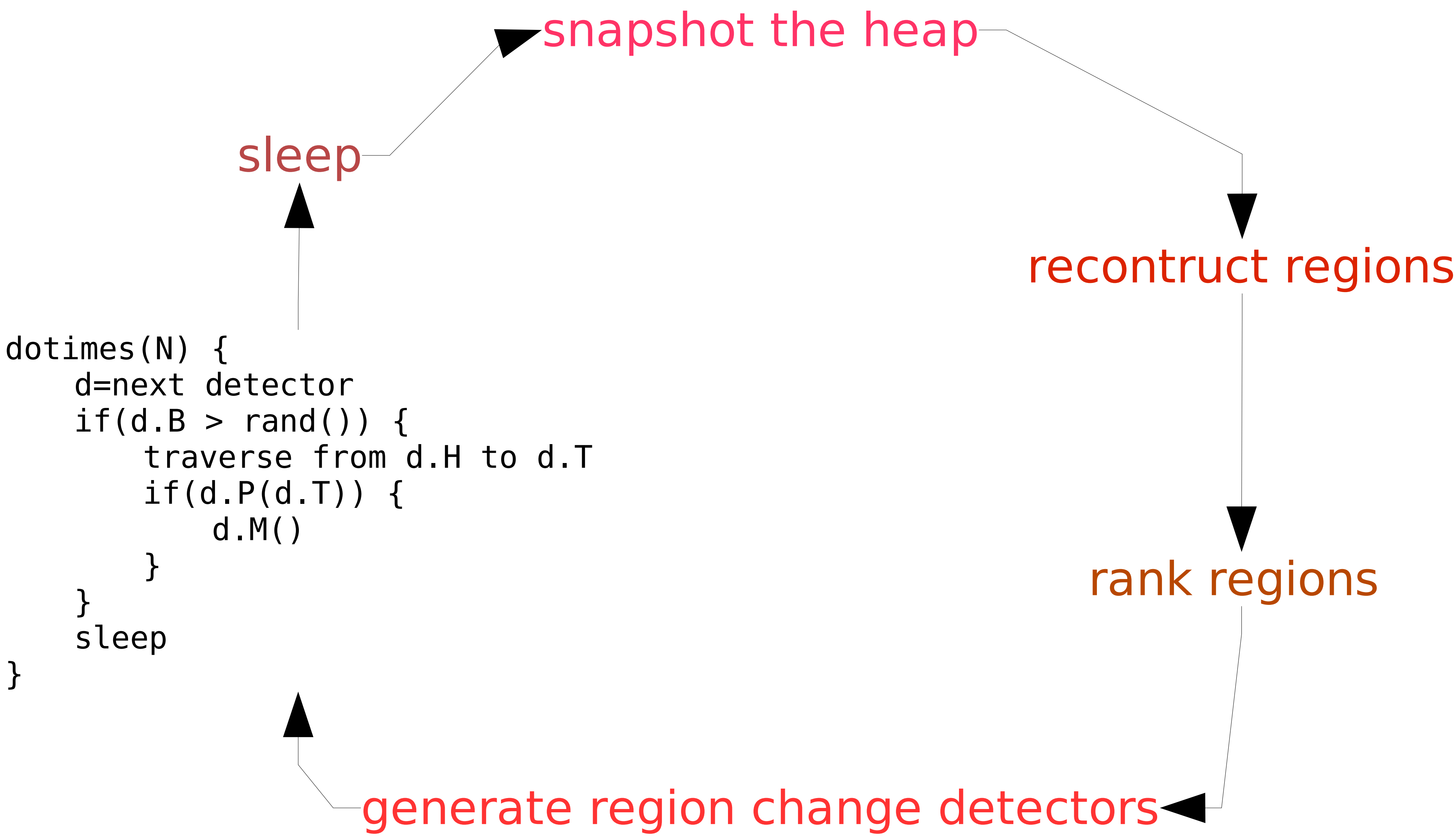
String

a bowl leaks

(every leaking operation leaks **one** of these **data structures**)



leakbot and its loops




```
dotimes(N) {
    d=next detector
    if(d.B > rand()) {
        traverse from d.H to d.T
        if(d.P(d.T)) {
            d.M()
        }
    }
    sleep
}
```

Strategies for Dissecting Leaks

(and some problems with each)

- histogram by datatype
  - Strings are in every data structure
- histogram by allocation site
  - Strings are allocated everywhere
  - expensive (c.f. HPROF's **5-10x** slowdown)
- visualize reference graph
  - an application doesn't just leak objects, it leaks entire (and entirely ugly) data structures
  - c.f. Jinsight, JProbe, Purify

Summary of the LeakBot Technique

- structure live objects into **Co-evolving Regions**
  - portions of data structures which change in similar ways
- **rank regions** according to likelihood of problem
  - only present to user those regions likely to leak, the suspects
  - e.g. of Schwab's 1M live objects, leakbot identifies <u>three</u> suspects
- **track evolution** of regions as program runs
  - treat structuring and ranking as initial estimates
    - e.g. we might have caught a pool being populated – it'll eventually plateau
  - from them, derive a scheme for very lightweight probing
  - verify whether initial estimates correct, and update ranking

offline mode

online mode

M: the mixture model

- no single property is entirely indicative
- instead, use gated mixure of them all

	instances	newer	on-stack	on-fringe	type overlap
EventNotifier	377,276	34%	0	44	33%
ThreadDiscriminator	274,433	2%	455	52	13%
FormProperties	270	97%	50	3	100%
XslTemplateCache	32	90%	0	1	40%
VerifySignonScenario	18	11%	1	1	50%

this application had two leaks