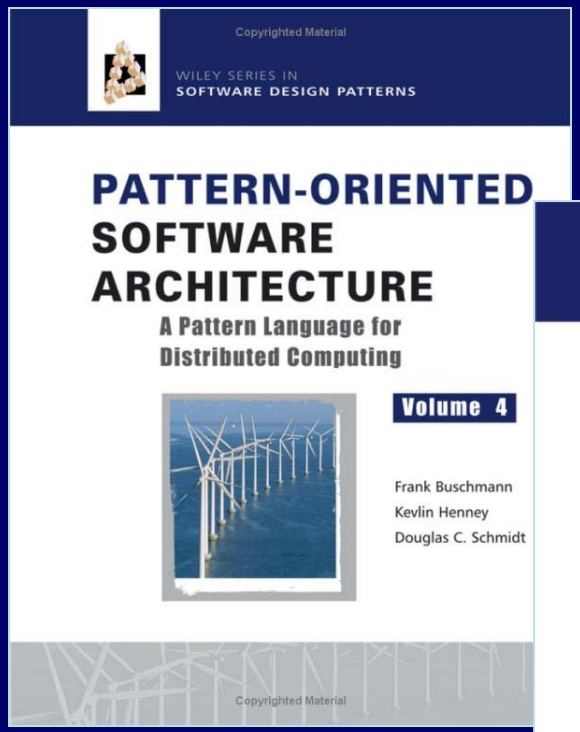
Design Stories

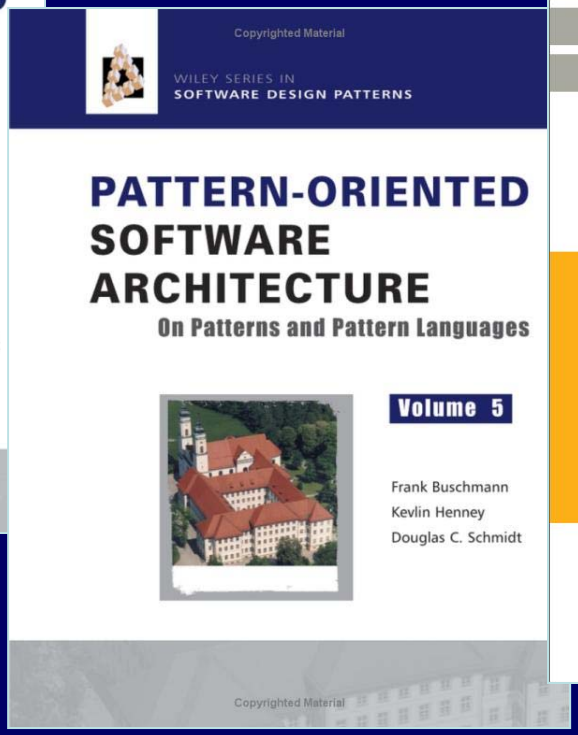
Exploring and Creating Code from a Narrative Perspective

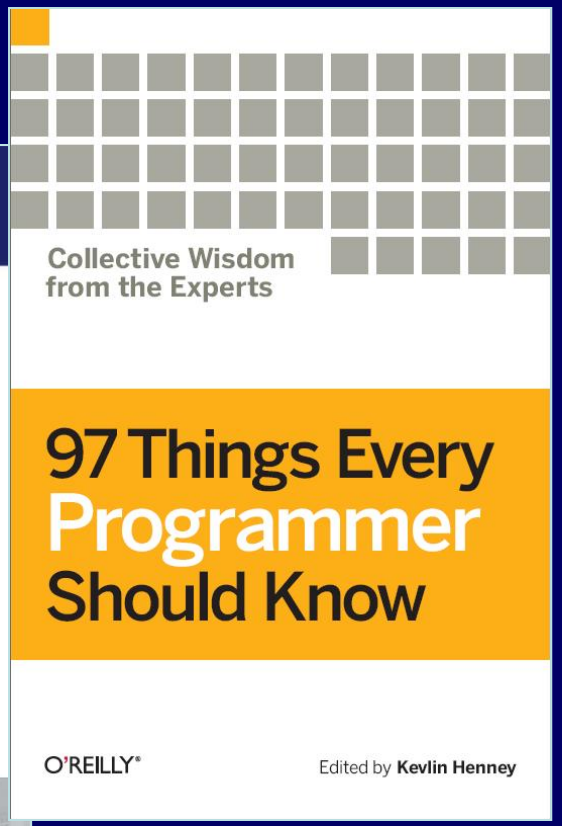
Kevlin Henney

kevlin@curbralan.com

@KevlinHenney







See *http://programmer.97things.oreilly.com*
(also *http://tinyurl.com/97tepsk*)
and follow *@97TEPSK*


55
0
5
10
15

An Agile View of Design

- Look forward
  - Establish a clear vision of the whole
- Approach design in time-driven, incremental slices
  - Quantisation as user stories, use cases, other scenario styles, features, spikes, programming episodes, etc.
- Look for feedback
  - Be responsive and reactive


007
新鐵金剛之量子殺機
QUANTUM OF SOLACE
11月6日 絕境反擊
007.COM

Quantum of Flow (QoF)

quaff, *v*.

To drink deeply; to take a long draught; also, to drink repeatedly in this manner.

quaff, *n*.

An act of quaffing, or the liquor quaffed; a deep draught.

Oxford English Dictionary


GOOD THINGS COME TO THOSE WHO WAIT.
GUINNESS

Slicing Design over Time

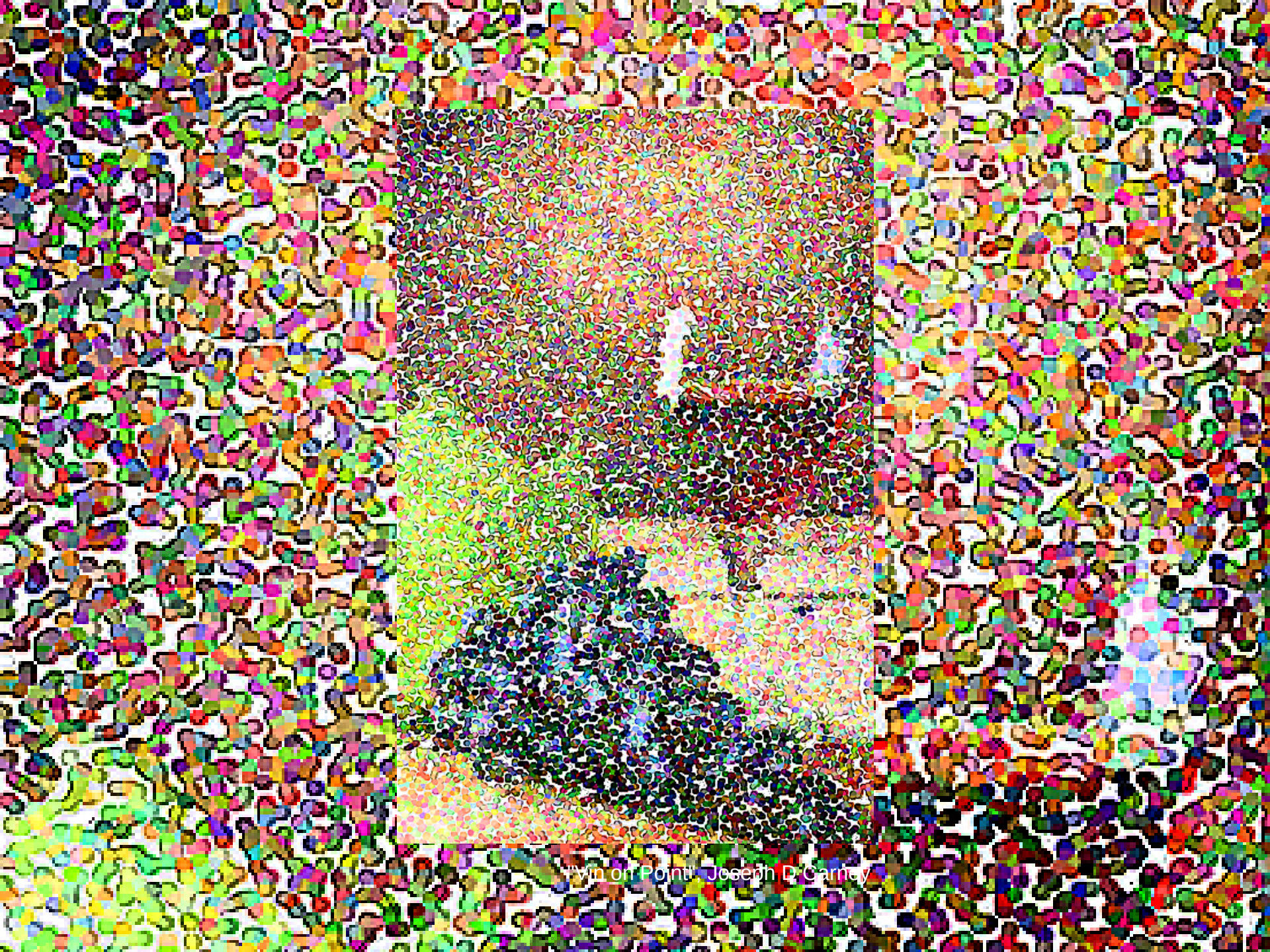
- Structural decomposition is one view of a system's design
  - E.g., a static view of code structure
- Temporal decomposition concerns how code is developed over time
  - Build by adding functionally complete capabilities based around usage goals
  - From user stories to pattern stories

Balancing Value and Risk

- Priority measures importance to a stakeholder, not urgency
  - Value can be context sensitive
- Keep in mind that business risk is something to be managed
  - It does not always manifest itself at the same time or in the same place as value or other measures of priority

Goal-Structured Slicing

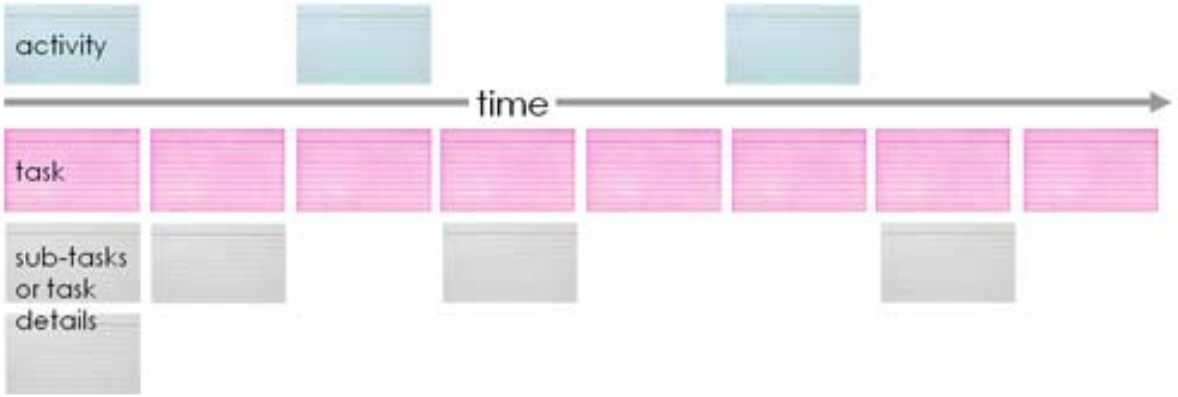
- Development steps in terms of visible, functionally complete slices
  - E.g., use cases, user stories, user story maps, FDD features and other scenario-based techniques — emphasis in each case is different, but all are related
- Each slice is anchored in a goal and works towards an outcome
  - They can be applied recursively

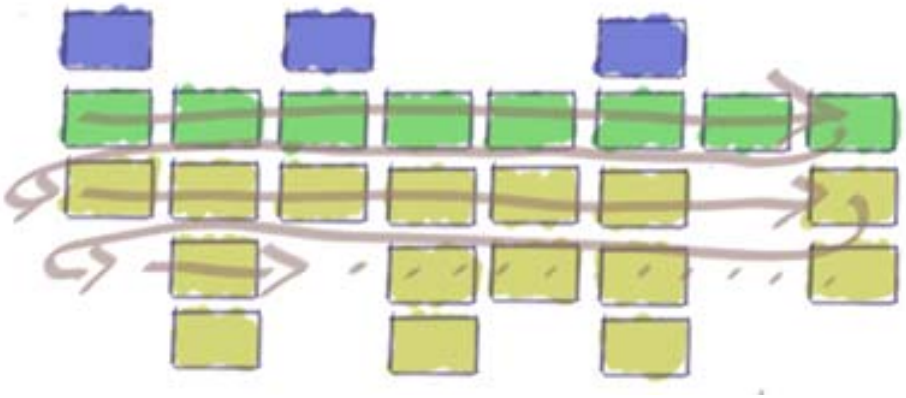

"Vin on Point" Joseph D Carney

The new user story backlog is a map

Jeff Patton

http://www.agileproductdesign.com/blog/the_new_backlog.html





Requirement-Styled Testing

- It seems obvious that tests should relate to requirements in some way
  - Also code-level requirements imposed on one piece of code by another
- But it is another thing to use a requirement-based style for tests
  - Tests should define behaviour, not just prod and poke at it
  - Applies to unit as well as system tests

```
public static boolean isLeapYear(int year)
```

Procedural test structured in terms of the function being tested, but not in terms of its functionality:

```
testIsLeapYear
```

Tests partitioned in terms of the result of the function being tested:

```
testNonLeapYears
testLeapYears
```

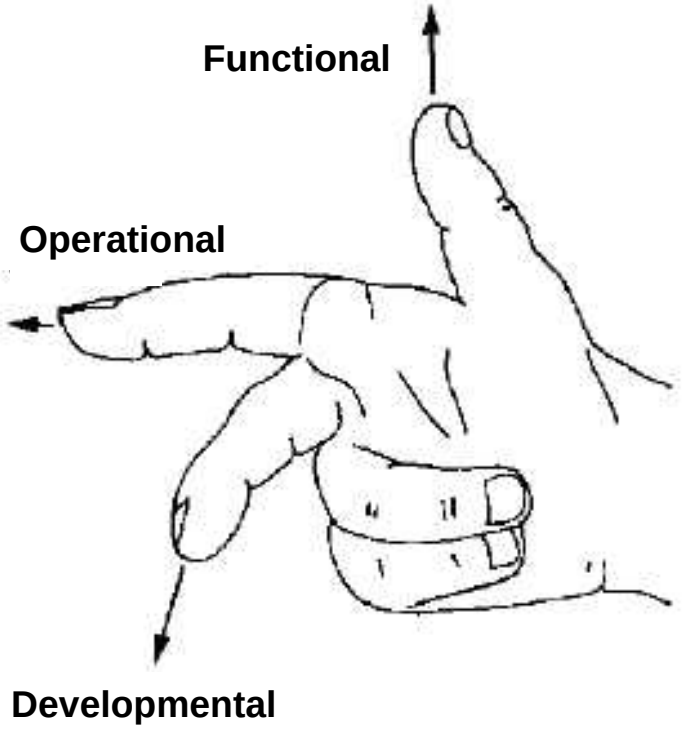
Propositional tests reflecting requirements and partitioned in terms of the problem domain (prefix with *test_that* if *test* is required as a prefix):

```
years_not_divisible_by_4_are_not_leap_years
years_divisible_by_4_but_not_by_100_are_leap_years
years_divisible_by_100_but_not_by_400_are_not_leap_years
years_divisible_by_400_are_leap_years
```

Refactoring (noun): ***a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.***

Refactor (verb): ***to restructure software by applying a series of refactorings without changing the observable behavior of the software.***

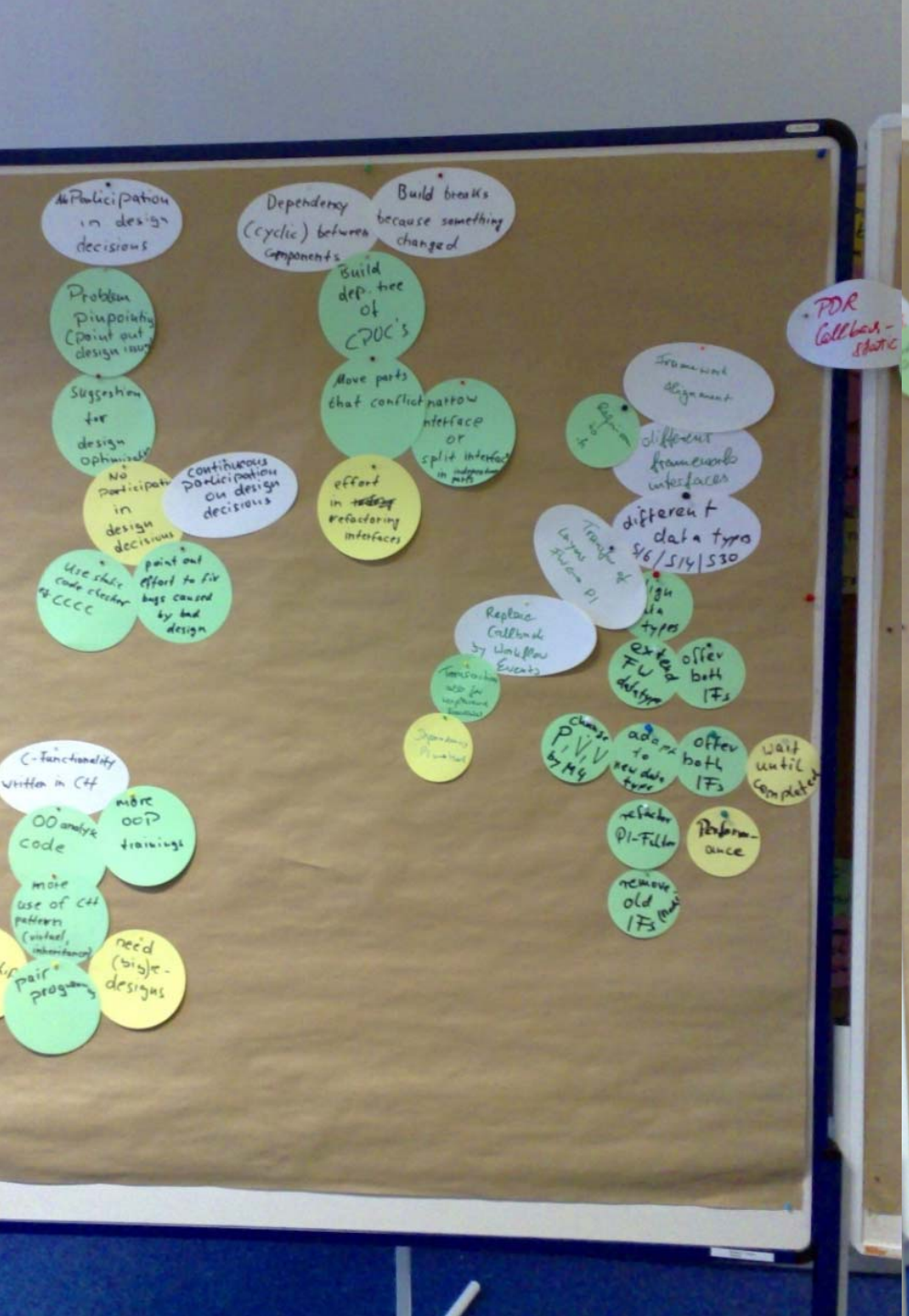
Martin Fowler, ***Refactoring***

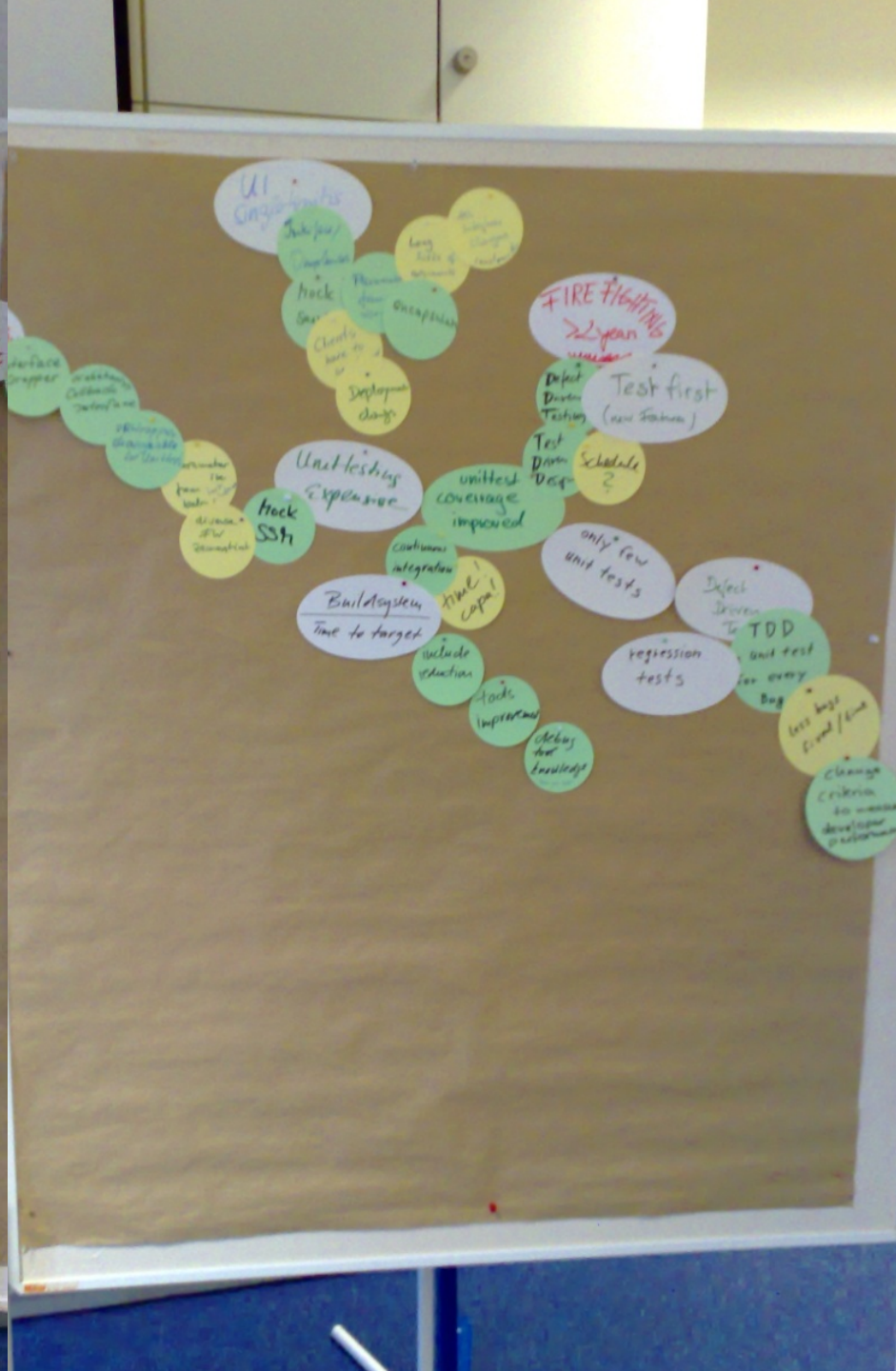

Functional
Operational
Developmental

```
class access_control
{
public:
    bool is_locked(const std::basic_string<char> &key) const
    {
        std::list<std::basic_string<char> >::const_iterator found = std::find(locked.begin(), locked.end(), key);
        return found != locked.end();
    }
    bool lock(const std::basic_string<char> &key)
    {
        std::list<std::basic_string<char> >::iterator found = std::find(locked.begin(), locked.end(), key);
        if(found == locked.end())
        {
            locked.insert(locked.end(), key);
            return true;
        }
        return false;
    }
    bool unlock(const std::basic_string<char> &key)
    {
        std::list<std::basic_string<char> >::iterator found = std::find(locked.begin(), locked.end(), key);
        if(found != locked.end())
        {
            locked.erase(found);
            return true;
        }
        return false;
    }
    ...
private:
    std::list<std::basic_string<char> > locked;
    ...
};
```

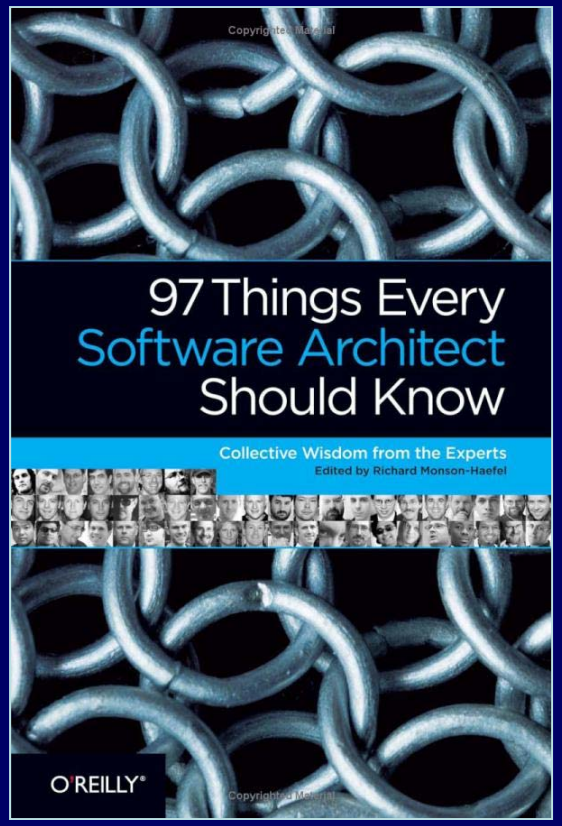
```
class access_control
{
public:
    bool is_locked(const std::string &key) const
    {
        return std::count(locked.begin(), locked.end(), key) != 0;
    }
    bool lock(const std::string &key)
    {
        if(is_locked(key))
        {
            return false;
        }
        else
        {
            locked.push_back(key);
            return true;
        }
    }
    bool unlock(const std::string &key)
    {
        const std::size_t old_size = locked.size();
        locked.remove(key);
        return locked.size() != old_size;
    }
    ...
private:
    std::list<std::string> locked;
    ...
};
```

```
class access_control
{
public:
    bool is_locked(const std::string &key) const
    {
        return locked.count(key) != 0;
    }
    bool lock(const std::string &key)
    {
        return locked.insert(key).second;
    }
    bool unlock(const std::string &key)
    {
        return locked.erase(key);
    }
    ...
private:
    std::set<std::string> locked;
    ...
};
```


No Participation in design decisions
Dependency (cyclic) between components
Build breaks because something changed
Problem Pinpointing (Point out design issue)
Build dep. tree of CPOC's
PDR Callback - static
Suggestion for design optimizations
Move parts that conflict
narrow interface or split interface
Framework alignment
different frameworks interfaces
No participation in design decisions
Continuous participation on design decisions
effort in refactoring interfaces
different data types
Use static code checker of CCCC
point out effort to fix bugs caused by bad design
Replace Callback by Workflow Events
extend FW datatypes
offer both IFs
C-functionality written in C++
OO analysis code
more OOP trainings
adapt to new data types
offer both IFs
wait until completed
refactor PI-Fillers
Performance
more use of C++ pattern (virtual, inheritance)
remove old IFs
need (big)-designs
pair programming


UI unit tests
FIRE FIGHTING
Test first
Unit testing expensive
unittest coverage improved
Mock SSH
Build system
Time to target
only few unit tests
TDD
regression tests

Using Uncertainty as a Driver



Confronted with two options, most people think that the most important thing to do is make a choice between them. In design (software or otherwise) it is not. The presence of two options is an indicator that you need to consider uncertainty in the design. Use the uncertainty as a driver to determine where you can defer commitment to details and where you can partition and abstract to reduce the significance of design decisions.

Kevlin Henney
"Use Uncertainty as a Driver"

No house should ever be on a hill or
on anything. It should be of the hill.
Belonging to it. Hill and house
should live together each the happier
for the other.

Frank Lloyd Wright

TABLE 1

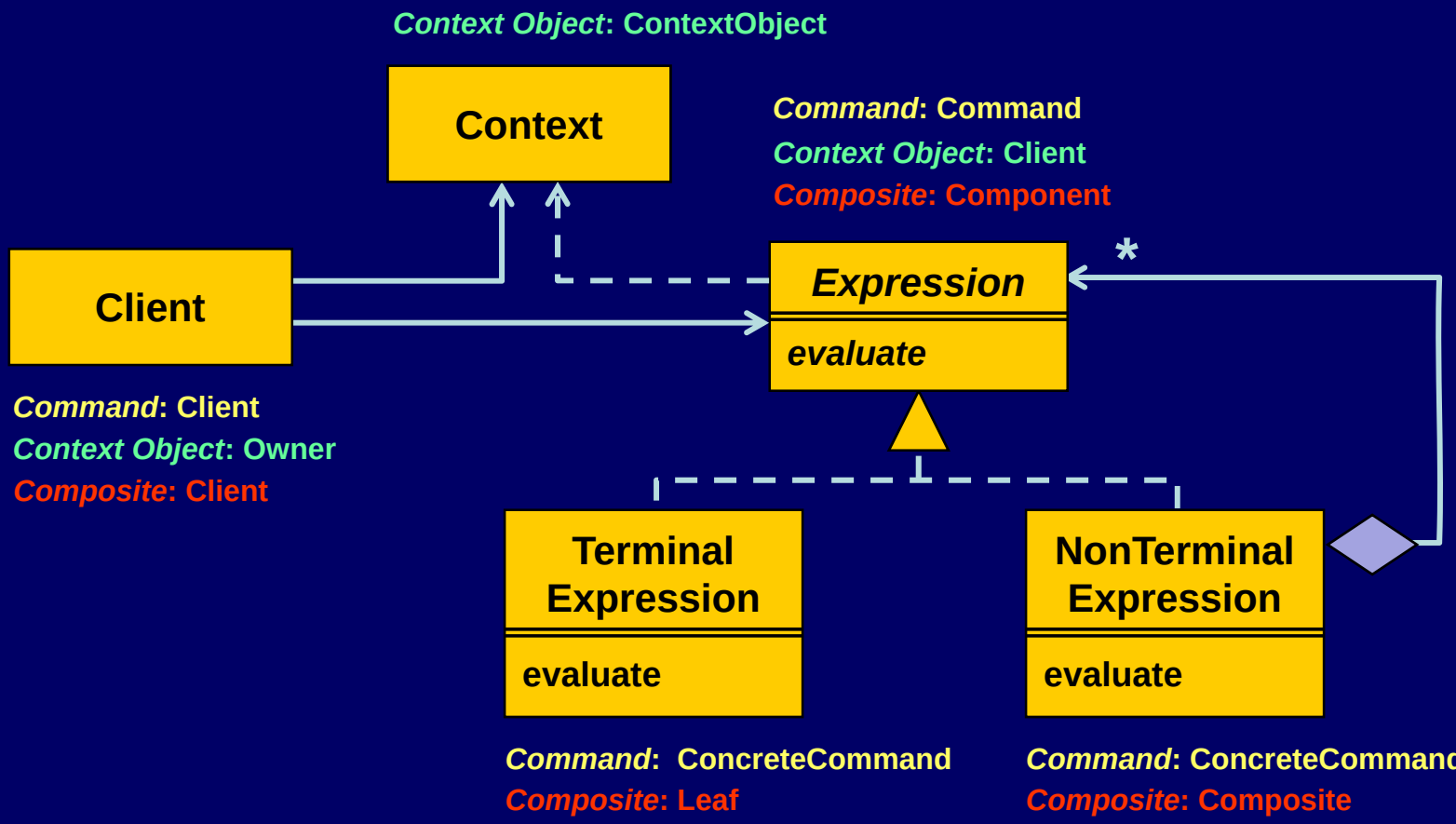
Skill Level / Mental Function	NOVICE	COMPETENT	PROFICIENT	EXPERT	MASTER
Recollection	Non-situational	Situational	Situational	Situational	Situational
Recognition	Decomposed	Decomposed	Holistic	Holistic	Holistic
Decision	Analytical	Analytical	Analytical	Intuitive	Intuitive
Awareness	Monitoring	Monitoring	Monitoring	Monitoring	Absorbed


PATTERN
SHOP
Caution
Uneven Floor

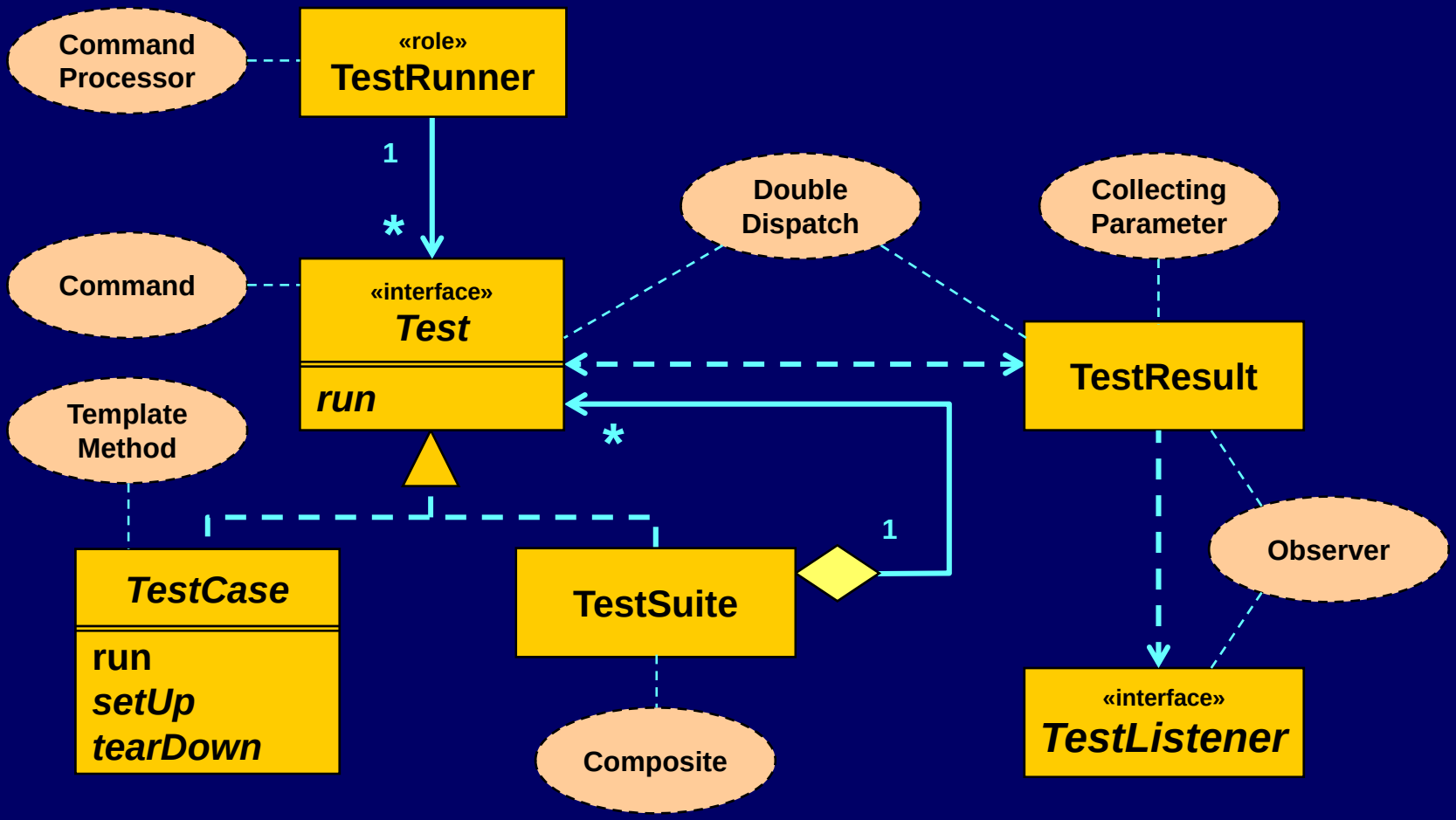
Patterns

- Patterns name and reason about recurring design decisions
  - Decisions may be implicit or explicit, conscious or not
  - The naming of a pattern contributes to design vocabulary
  - Patterns described in terms of context, problem forces, solution structure and consequences

Inside the Interpreter Pattern



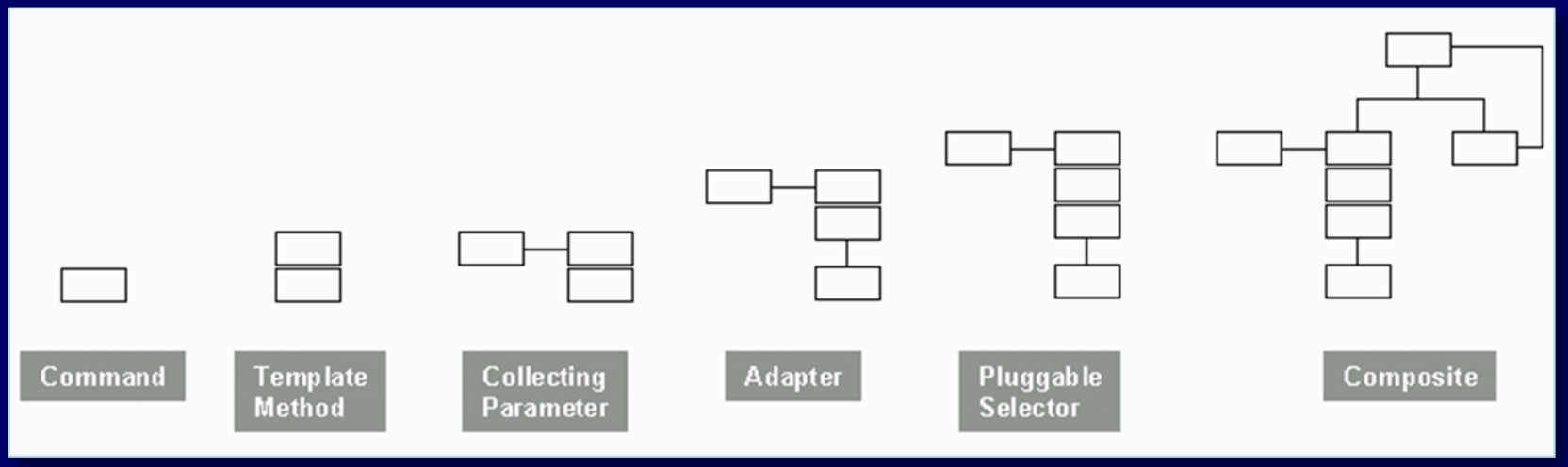
Pattern Usage in Classic JUnit

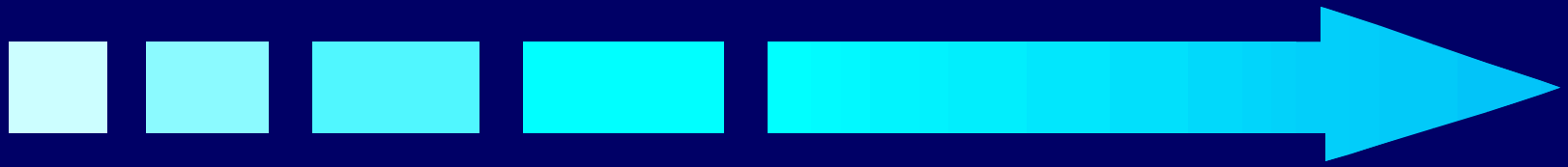


Pattern Stories

- A pattern story brings out sequence of patterns in a design example
  - Capture conceptual narrative behind a given piece of design, whether a system in production or an illustrative example
  - Forces and consequences played out in order, each decision illustrated concretely

JUnit Storyboard

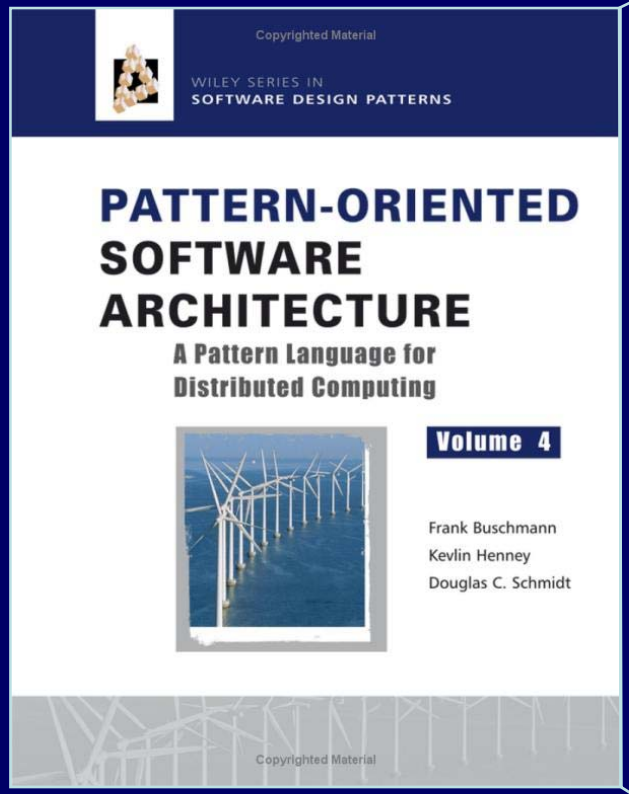




History rarely happens in the right order or at the right time, but the job of a historian is to make it appear as if it did.

James Burke

POSA4 Warehouse Story



Pattern Sequences

- A pattern sequence captures the underlying narrative behind a story
  - A sequence can be described and applied independent of a pattern story
  - Pattern sequences focus on incremental development
- Pattern compounds are examples of named, short sequences
  - E.g., MVC, Pluggable Factory

Interpreter's Sequence

- The Interpreter pattern can be seen as a pattern compound
  - A recurring set of overlapping and interacting roles
- It can also be seen as a sequence of pattern application
  - I.e., ⟨*Command*, *Context Object, Composite*⟩

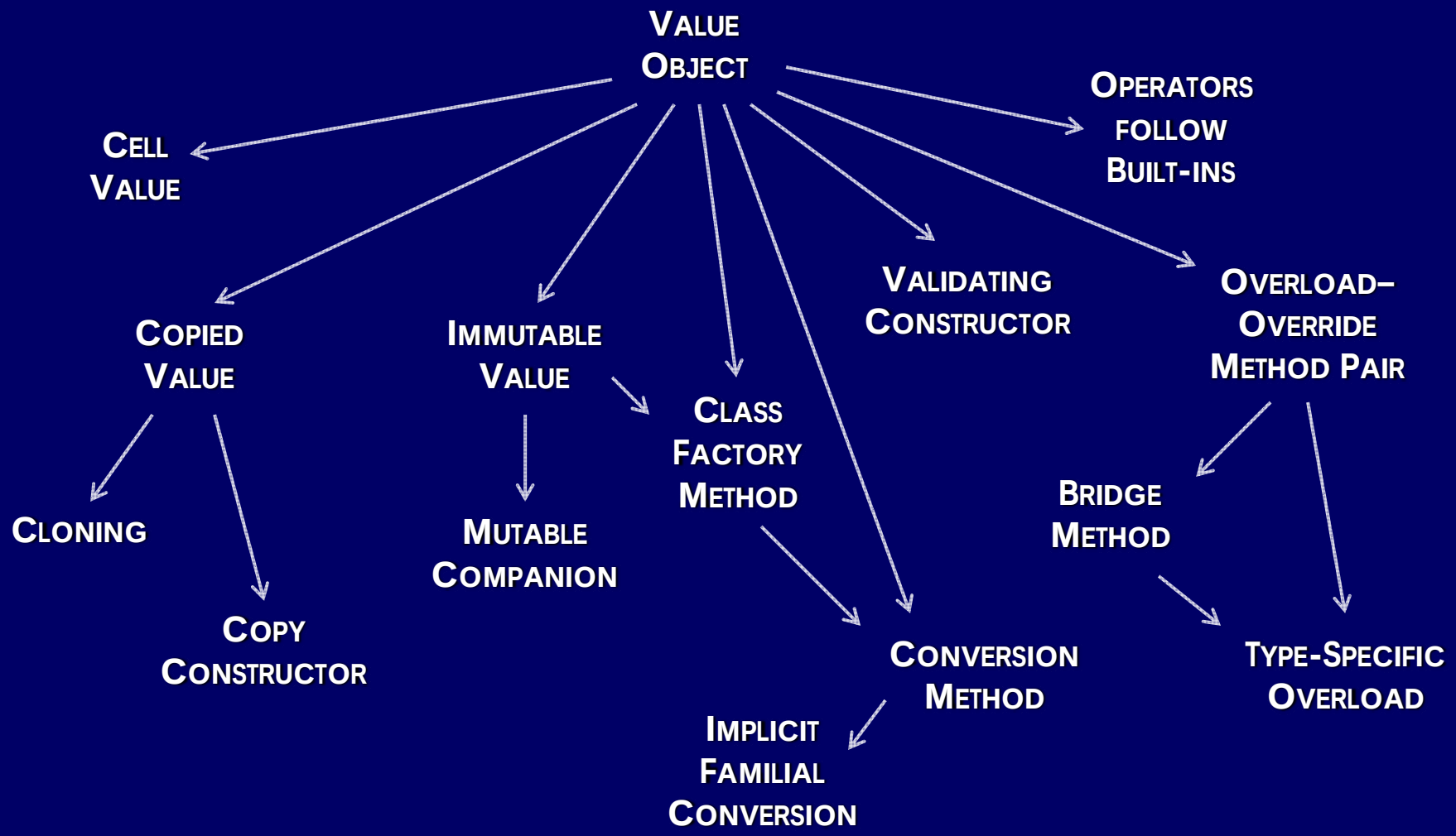
JUnit Storyboard Distilled

- JUnit storyboard can be summarised as a pattern sequence
  - I.e., ⟨*Command*, *Template Method, Collecting Parameter, Class Adapter, Pluggable Selector, Composite*⟩
- A summary of the sequence does not show how roles interact
  - E.g., what classes play what roles in Composite

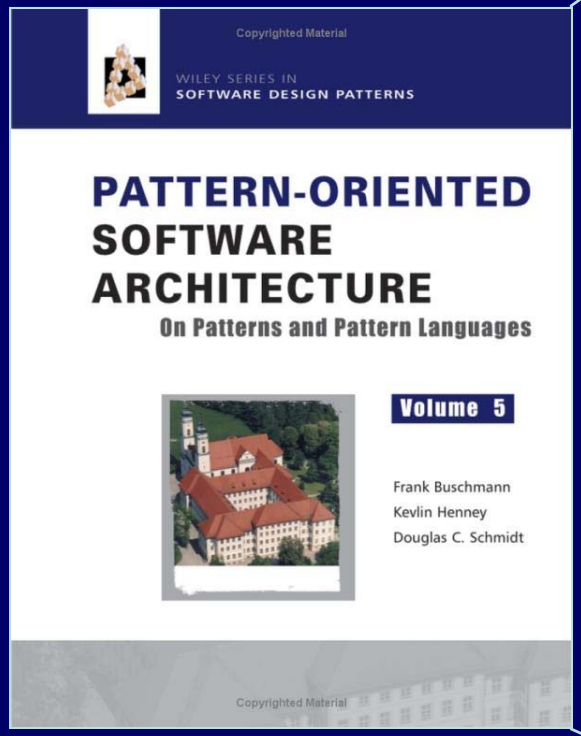
Pattern Languages

- A pattern language connects many patterns together
  - Captures connections and possibilities between patterns, including options, alternatives and necessary steps
- There may be many possible sequences through a language
  - A lone pattern sequence can be considered a narrow pattern language

Patterns of Value



POSA5 Request Handling



Patterns Form Vocabulary, Sequences Illustrate Grammar 283

portion from the grammar of our example pattern language for request handling, derived from the pattern sequences presented above:

∅ →° (COMMAND → EXPLICIT INTERFACE →°
(MEMENTO →° COMPOSITE →° COMMAND PROCESSOR →
COLLECTIONS FOR STATES → STRATEGY → NULL OBJECT)
| (COMPOSITE →° MEMENTO))

A BNF-derived notation [EBNF96], as used for specifying the syntax of programm
grammar of a
alternative:

COMMAND
lowed by
followed
POSITE, w
which if
which m
NULL OBJ
followed
MENTO.

Graphical no
pattern langu
which visuall
design [CzEi0
language for
root *concept*
combinations
as whether t
tional. In bor
confused with
concepts via

Another graph
'railroad' nota
since the 197
in particular

284 A Network of Patterns and More

The portion of our pattern language for request handling outlined above could be represented as follows using the 'railroad notation':

The preferences of pattern language authors or the demands of their target audience determine the specific expression of a grammar that works best—whether a list of pattern sequences, formal or semi-formal prose, or a graphical form of describing grammar rules, and whether interwoven with the pattern descriptions or separate. For example, the pattern language for distributed computing from POSA4 expresses grammar rules in prose, interwoven with the pattern descriptions [POSA4]. This option has been chosen by most pattern languages in the software area, from design-centric pattern languages [VSW02] [Fow02] [HoWo03] [VKZ04] to pattern languages for development process and organization, and project and people management [Ker95] [Cun96] [CoHa04].

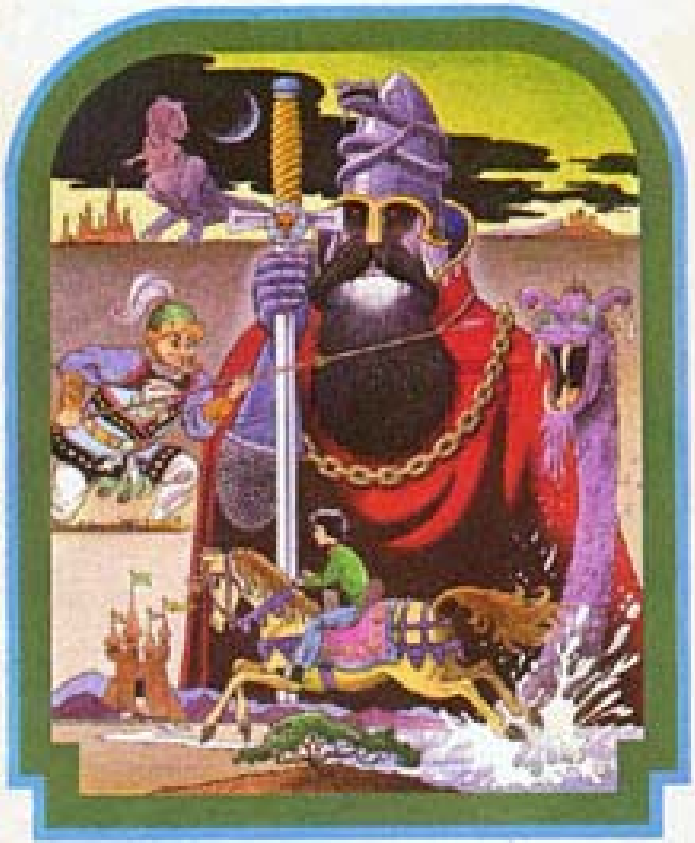
Regardless of which grammar form is chosen, however, it is important that documented pattern languages actually offer guidance on the issue of meaningful paths through a language. Otherwise, it is hard to avoid the selection of ill-formed pattern sequences that create fundamentally broken software. The set of sensible sequences through a language is part of the language and not something accidental or separate. Thus making the grammars of pattern languages more explicit is one method for supporting their appropriate use. However, we must also recognize some practical limitations in this endeavor: the grammar for

CHOOSE YOUR OWN ADVENTURE™ · 1

YOU'RE THE STAR OF THE STORY!
CHOOSE FROM 40 POSSIBLE ENDINGS

THE CAVE OF TIME

BY EDWARD PACKARD



ILLUSTRATED BY PAUL GRANGER

Interactive Pattern Stories

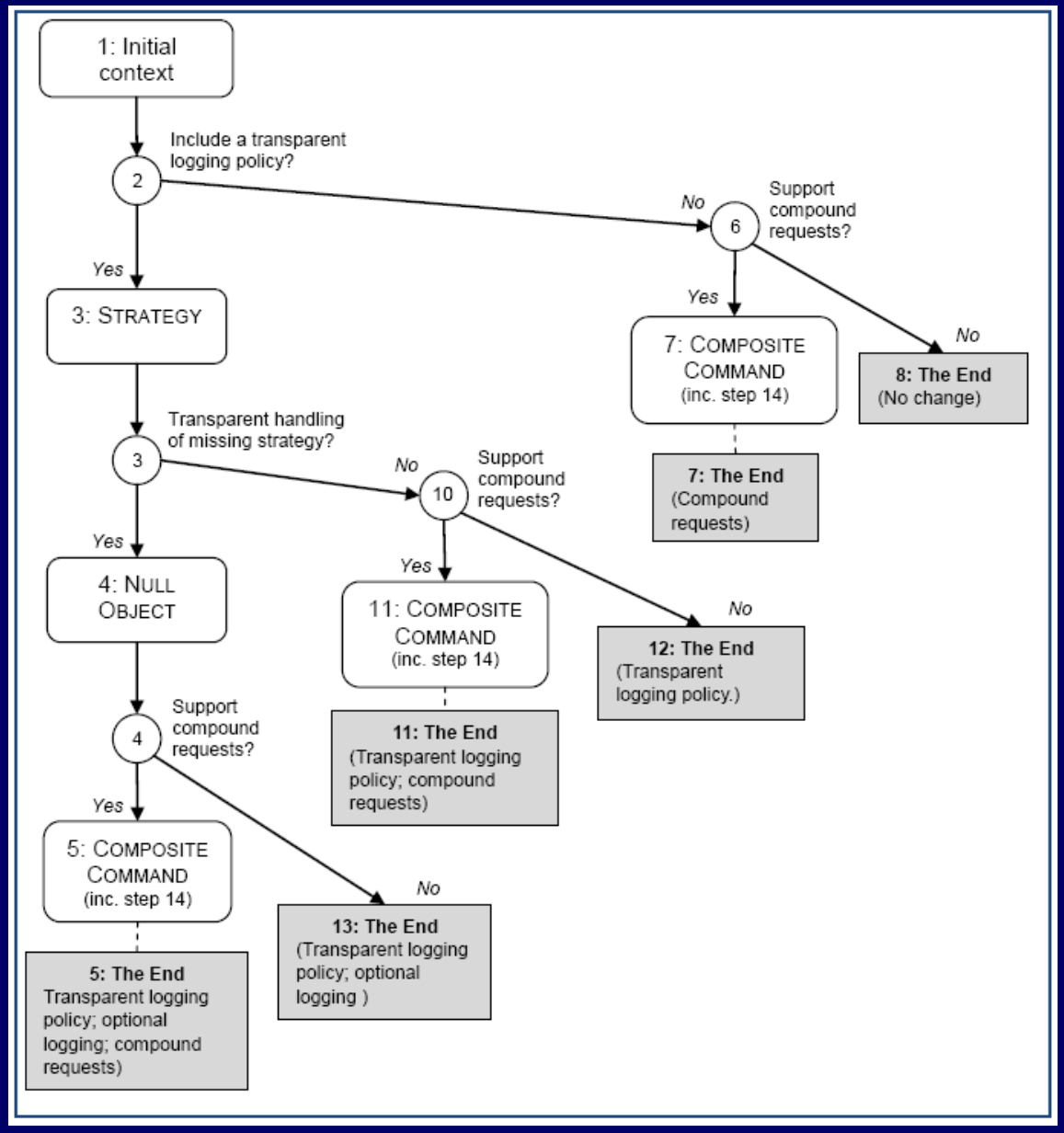
2

You now realise that the framework needs a logging facility for requests, and wonder how logging functionality can be parameterized so that users of the framework can choose how they wish to handle logging, rather than the logging facility being hard-wired.

If you wish to use inheritance to support variations in housekeeping functionality, turn to 7.

Otherwise if you prefer the use of delegation, turn to 3.

James Siddle
"Choose Your Own Architecture" – Interactive Pattern Storytelling


1: Initial context
Include a transparent logging policy?
2
No
6
Support compound requests?
Yes
3: STRATEGY
Yes
7: COMPOSITE COMMAND (inc. step 14)
No
8: The End (No change)
Transparent handling of missing strategy?
3
No
10
Support compound requests?
7: The End (Compound requests)
Yes
4: NULL OBJECT
Yes
11: COMPOSITE COMMAND (inc. step 14)
No
12: The End (Transparent logging policy.)
Support compound requests?
4
11: The End (Transparent logging policy; compound requests)
Yes
5: COMPOSITE COMMAND (inc. step 14)
No
13: The End (Transparent logging policy; optional logging)
5: The End Transparent logging policy; optional logging; compound requests)

Like snowflakes, the human pattern is never cast twice. We are uncommonly and marvelously intricate in thought and action, our problems are most complex and, too often, silently borne.

Alice Childress