Agile Software Development
The Cooperative Game:
An Overview

Alistair Cockburn

http://Alistair.Cockburn.us

Agile Software Development,Cooperative Game
Schedule of the day

I. Programming / Cooperative Games

II. People / Communication / Cooperation

III. Self-Evolving Methodologies

IV. Agile Techniques

V. Named Agile Methodologies

Things you may take away from today:

- **The Cooperative Game vocabulary**
- **Manage the *incompleteness* of communication**
- **Distance hurts. (So DON’T DO IT !)**
- **Take advantage of face-to-face communication.**
- ***Incremental delivery* => Product feedback**
- ***Reflection workshops* => Process feedback**
- **Use of *information radiators***
- **Relevance of *amicability* and *convection currents***
- **Three levels of skill (*shu, ha, ri)***
- **Why methodologies need *tuning* to fit their ecosystem**
- **How to tune yours to your project and your people**
- ***Timeboxing / increments* as core technique**
- ***Burn-down* charts for visibility, *iceberg list* for scheduling**
- ***Concurrent development* saves time**
- **Optimize the rules to fit project-specific bottenecks**
- **All agile methodologies use *empiriical process*, *cooperative game* concepts**
- **How to mix in Scrum, consider other named agile methodologies**
- **How to look for agile methodologies in your environment**

Agile Software Development,Cooperative Game
Schedule of the day

I. Programming / Cooperative Games

II. People / Communication / Cooperation

III. Self-Evolving Methodologies

IV. Agile Techniques

V. Named Agile Methodologies

Cockburn 1998: Making software consists *only* of making ideas concrete in an economic context:

People inventing and communicating, solving a problem they don't yet understand *(which keeps changing),*

Creating a solution they don't really understand *(and which keeps changing),*

Expressing ideas in restricted languages they don't really understand, *(and which keep changing)*

To an interpreter unforgiving of error.

Resources are limited,
and every choice has economic consequences.

It is a cooperative game of invention and communication !

Software development is a *Cooperative Game of Invention and Communication.*

To understand *team* software development:

- Understand goal-directed cooperative *games*
- Understand *people communicating*
- Understand *people inventing*
- Understand *people cooperating*

Notes on Cooperative Game: Two goals:

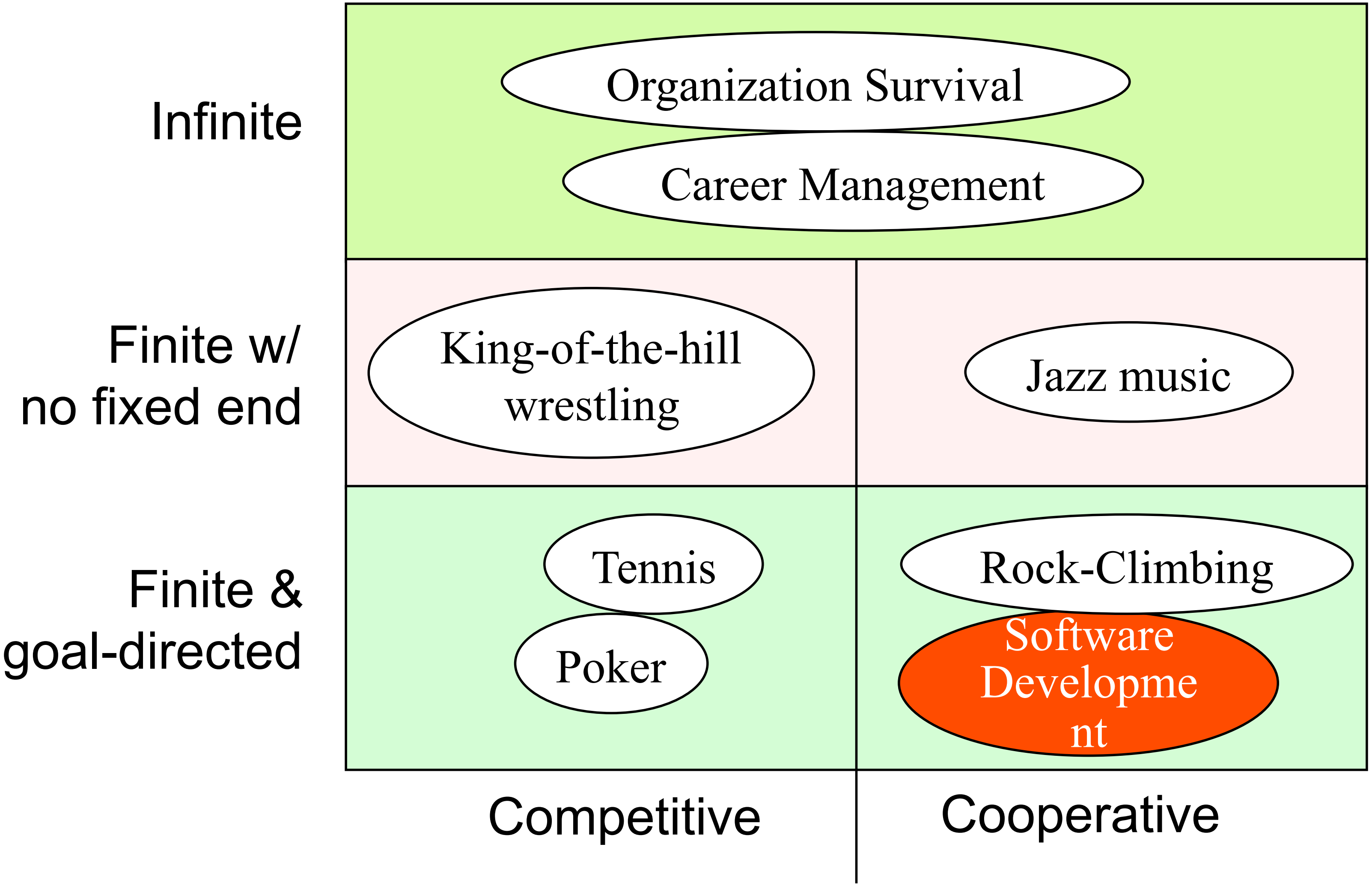
Primary Goal → *Deliver this software*

Secondary Goal → *Set up for the next game*

Two conflicting games in one

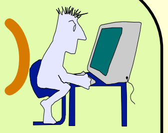
Net result: Not repeatable !

Games, finite/infinite or cooperative/competitive, consist of better/worse '*moves*'



	Competitive	Cooperative
Infinite	Organization Survival Career Management	
Finite w/ no fixed end	King-of-the-hill wrestling	Jazz music
Finite & goal-directed	Tennis Poker	Rock-Climbing Software Development

Musashi 1685: swordfighting (or gaming?) -discuss how this relates to programming-



* The field of martial arts is particularly rife with flambouyant showmanship, with commercial popularization and profiteering.

* Other schools become theatrical, dressing up and showing off to make a living.

* If your sword misses the opponent, leave it there for the moment, until the opponent strikes again, whereupon strike from below

* You should observe reflectively, with overall awareness of the large picture as well as precise attention to small details...

* The views of each school, and the logic of each path, are realized different, according to the individual person, depending on the mentality...

* One can win with the long sword, and one can win with the short sword.

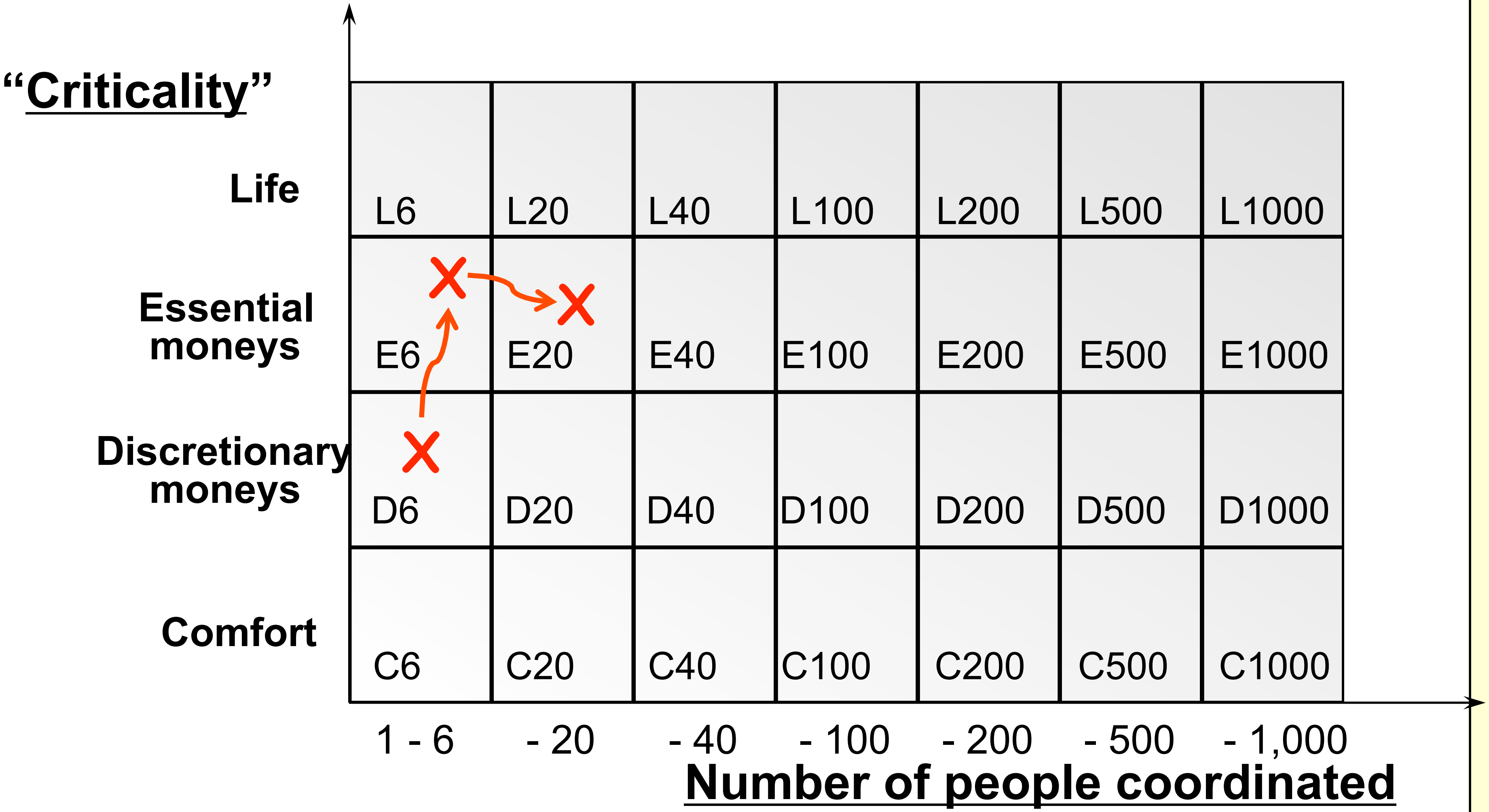
* Where you hold your sword depends on your relationship to the opponent, on the place, and must conform to the situation;

* wherever you hold it, the idea is to hold it so that it will be easy to kill the opponent. This must be understood.

* Whatever guard you adopt, do not think of it as being on guard; think of it as part of the act of killing.

* In my school, no consideration is given to anything unreasonable; the heart of the matter is to use the power of knowledge of martial arts to gain victory any way you can.

Every game run uses different strategies -- Set up each project's suitably or suffer



"Criticality"

"Criticality"	1 - 6	- 20	- 40	- 100	- 200	- 500	- 1,000
Life	L6	L20	L40	L100	L200	L500	L1000
Essential moneys	E6	E20	E40	E100	E200	E500	E1000
Discretionary moneys	D6	D20	D40	D100	D200	D500	D1000
Comfort	C6	C20	C40	C100	C200	C500	C1000

Number of people coordinated

Naur 1986: The primary result of programming is the theory held by the programmers

1. **Theory**: The knowledge a person must have to do certain things intelligently, explain, answer queries, argue about them...

2. The programmer must **Build a Theory** of how certain affairs of the world will be handled by a program; **Explain** how the affairs of the world are mapped into the program and documentation; **Respond to demands for modifications**, perceiving the similarity of the new demand with the facilities built.
   - This knowledge transcends that possible in documentation.

3. This theory is the mental possession of a programmer; the notion of programmer as an easily replaceable component in program production has to be abandoned.

Naur 1986: Modifying a program depends on the new programmers building the same theory !

4. Problems of **program modification** arise from assuming that programming consists of text production, instead of theory building.

5. The **decay of a program** from modifications made by programmers without proper grasp of the underlying theory becomes understandable. The need for direct participation of persons who possess the appropriate insight becomes evident. For a program to retain its quality it is mandatory that each modification is firmly grounded in its theory.

6. The conclusion seems inescapable that at least with certain kinds of large programs, the continued adaption, modification, and correction, is essentially dependent on a certain kind of knowledge possessed by a group of programmers who are closely and continuously connected with them.

Naur 1986: *Programming as Theory Building (Case Studies)*

CASE 1

A compiler extension ... Group B’s solutions were found by group A to make no use of the facilities that were inherent in the existing compiler and were discussed at length in its documentation; based instead on patches that effectively destroyed its power and simplicity.

Group A members were able to spot these instantly and could propose simple, effective solutions, framed within the existing structure.

This is an example of how the program text and additional documentation is insufficient in conveying to even the highly motivated group B the deeper insight into the design, that theory which is immediately present to the members of group A.

CASE 2

A program of 200,000 lines ... The installation programmers have been closely concerned with the system as a full time occupation over a period of several years, from the time the system was under design. When diagnosing a fault they rely almost exclusively on their ready knowledge of the system and the annotated program text, and are unable to conceive of any kind of additional documentation that would be useful to them.

Other groups, who received documentation and guidance from the producer's staff, regularly encounter difficulties that upon consultation with the installation programmers are traced to inadequate understanding of the documentation, but which can be cleared up easily by the installation programmers.

Naur 1986: There can be no right method; Design Simply.

7. On the Theory Building View, which techniques to use and in what order **must remain entirely a matter for the programmer to decide**, taking into account the actual problem to be solved.

8. It is often stated that programs should be designed to include a lot of flexibility, so as to be readily adaptable to changing circumstances. However, flexibility can in general only be achieved at a substantial cost. Each item of it has to be designed, including what circumstances it has to cover and by what kind of parameters it should be controlled. Then it has to be implemented, tested, and described. This cost is incurred in **achieving a program feature whose usefulness depends entirely on future events**.

Phil Armour's 2003 "Laws of Software Process" match *Theory Building, Cooperative Gaming*:

Process only allows us to do things we already know how to do.

The only processes we can use on the current project were defined on previous projects. ... Which were different from this one.

We can only define software processes at two levels: too vague and too confining

Software process rules should be stated in terms of two levels: a general statement of the rule, and a specific detailed example

The last knowledge to be implemented into an executable software system will be the knowledge of how to implement knowledge into an executable software system.

Agile Software Development, Cooperative Game
Schedule of the day

I. Programming / Cooperative Games

II. People / Communication / Cooperation

III. Self-Evolving Methodologies

IV. Agile Techniques

V. Named Agile Methodologies

COMMUNICATION (e.g. this talk) is a sequence of touching into shared experience

Linked sequences of shared experience
becomes a shared experience !

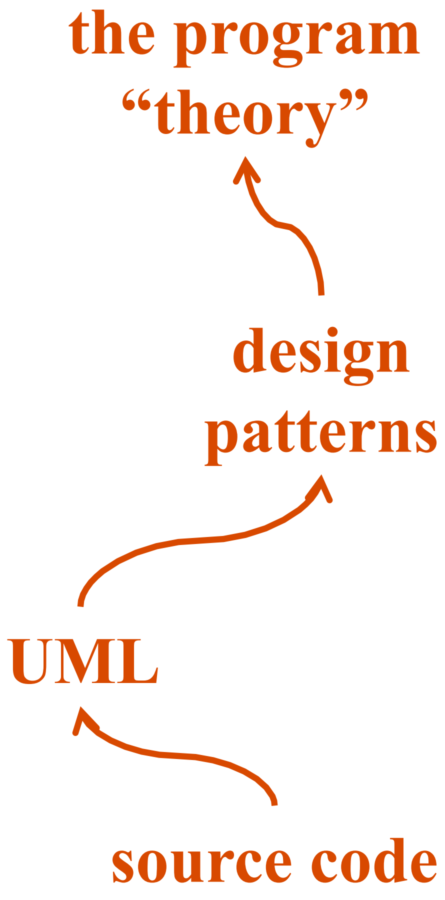
- Project colleagues have rich shared experiences, a shortcut vocabulary

Implications for documentation:

- *can never* fully specify requirements
- *can never* fully document design
- must assume reader's experiences

  more => can write less
  less => must write more.

Our task is to manage the
incompleteness of communications !!!



the program "theory"

design patterns

UML

source code

COMMUNICATION:

Perfect communication is *impossible*

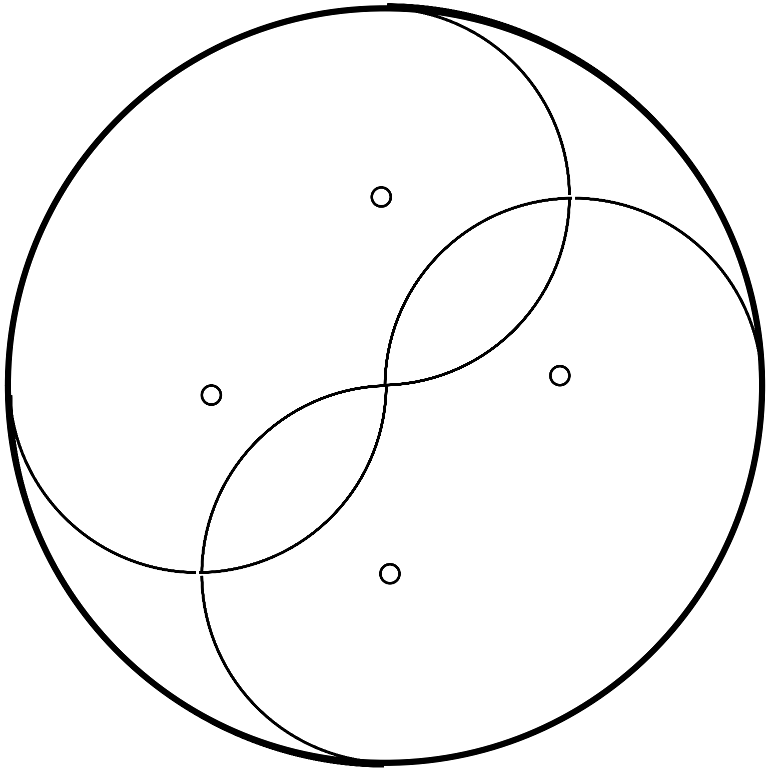
You try to communicate what you "know"

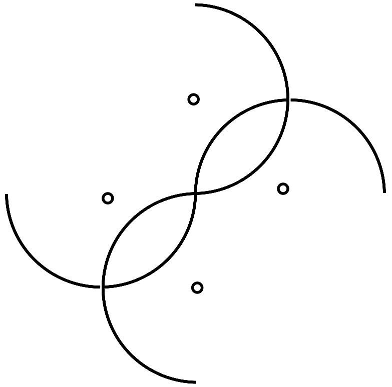
- What you "know" depends on your *individualized* parsing of the world around you;
- You don't know what it is you do know;
- You neither know the thing you are trying to communicate nor what you are actually communicating;
- Your listener *sees* only a part of what you are *saying;*
- What your listener learns depends on his/her internal state (plus everything around you).

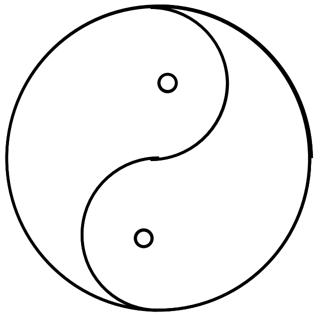
(p.s. How is it we communicate at all?)

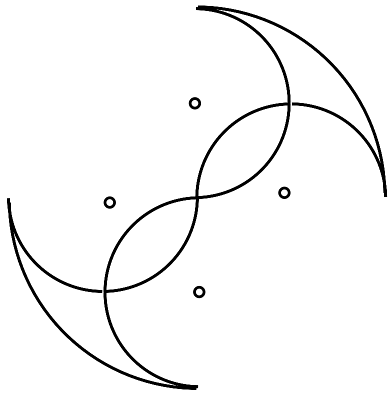
Our perceptions lie to us; our memory lies to us; we all parse experience differently.

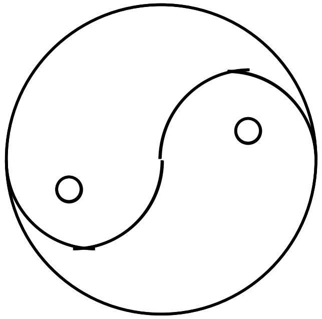
This figure is built from which of the figures on the right?









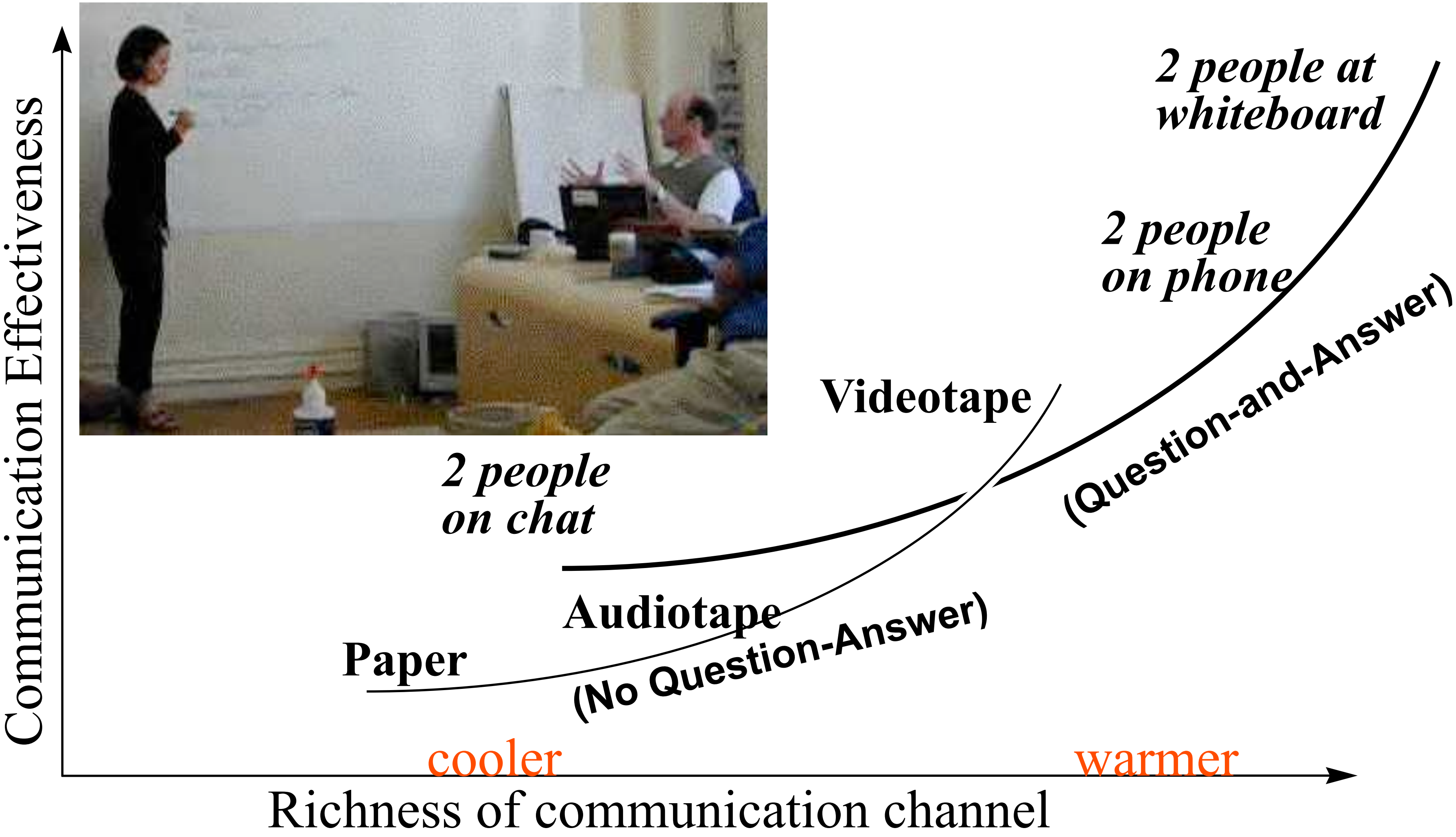


(wine bottle)

You don't need *perfect* communication: You need "good enough to proceed"

- You don't know what you know, nor the thing you are trying to communicate nor what you are communicating.
- Your listener *sees* only a part of what you are *saying*.
  - Your body communicates subvocally directly to the listener's emotions
- What your listeners learn depends on their mental state.
  - You only 'take on' what you are ready to
- "Equivocality" is the ambiguity in communication.
  - It will never be zero (no matter how hard you try)
  - Action and feedback reduces it to acceptable levels.
  - That is *Good Enough*

Face-to-face allows vocal, subvocal, gestural information to flow, with fast feedback



Experience *incommunicability* and elementary Agile techniques for yourselves

Divide each into Specifiers and Artists.

The Specifiers will ask the Artists
to draw a drawing for them.

Draw a Drawing
1st round -- 10 minutes

1. Specifiers and Artists move to opposite ends of the room.
   (this corresponds to distributed virtual teams)

2. Specifiers WRITE instructions to their Artists on what to do (no drawings allowed). One Specifier carries messages back and forth, can watch but NOT speak at Artist side.
4. Artists may write messages back.
5. SMS messages OK on cell phones, but no MMS.
6. NO speaking or drawing between Specifiers and Artists.

Reflection technique: Write in this chart. What worked? What might you try next time?

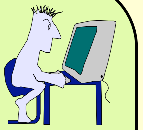
Keep These

Ongoing Problems

Try These

(this is a core technique you should use every month on every project !)

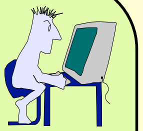
PAUSE
5 MINUTES



1. Pause. Most teams do not create a way to change process "on the fly". We need a way to evolve the process.
2. Discuss and reflect: what went good and bad. (DO NOT SHOW THE DRAWING !)
3. Adjust your strategy for the next round.
4. Use the reflection technique & chart.

(this corresponds to "iterative development" with process feedback.)

Draw a Drawing
2nd round -- 5 minutes



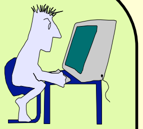
1. As before, but use incremental technique:
2. Specifiers describe only ONE shape.
   (Advanced teams: two people one shape each.)

3. After Artists have drawn the shape,
   they send it (or copy) to the Specifiers to see.

4. Specifiers can decide whether to correct that
   shape, or go on with the next shape.

(this corresponds to "incremental delivery" with
product feedback.)

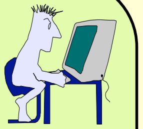
PAUSE
5 MINUTES



1. Assume only 1 Specifier on next drawing (everyone else is an Artist).

2. Strategize on the Best way to work (including sitting together, talking in person).

Draw a Drawing
3rd round -- 5 minutes



1. Specifier from each team comes and looks at drawing, puts into her/his memory.

2. Specifier, WITHOUT DRAWING ANYTHING, communicates it as well as possible to rest of team.

(this corresponds to Customer attempting to describe needed product to development team)

End. You have just seen a cooperative game of invention & communication in action.

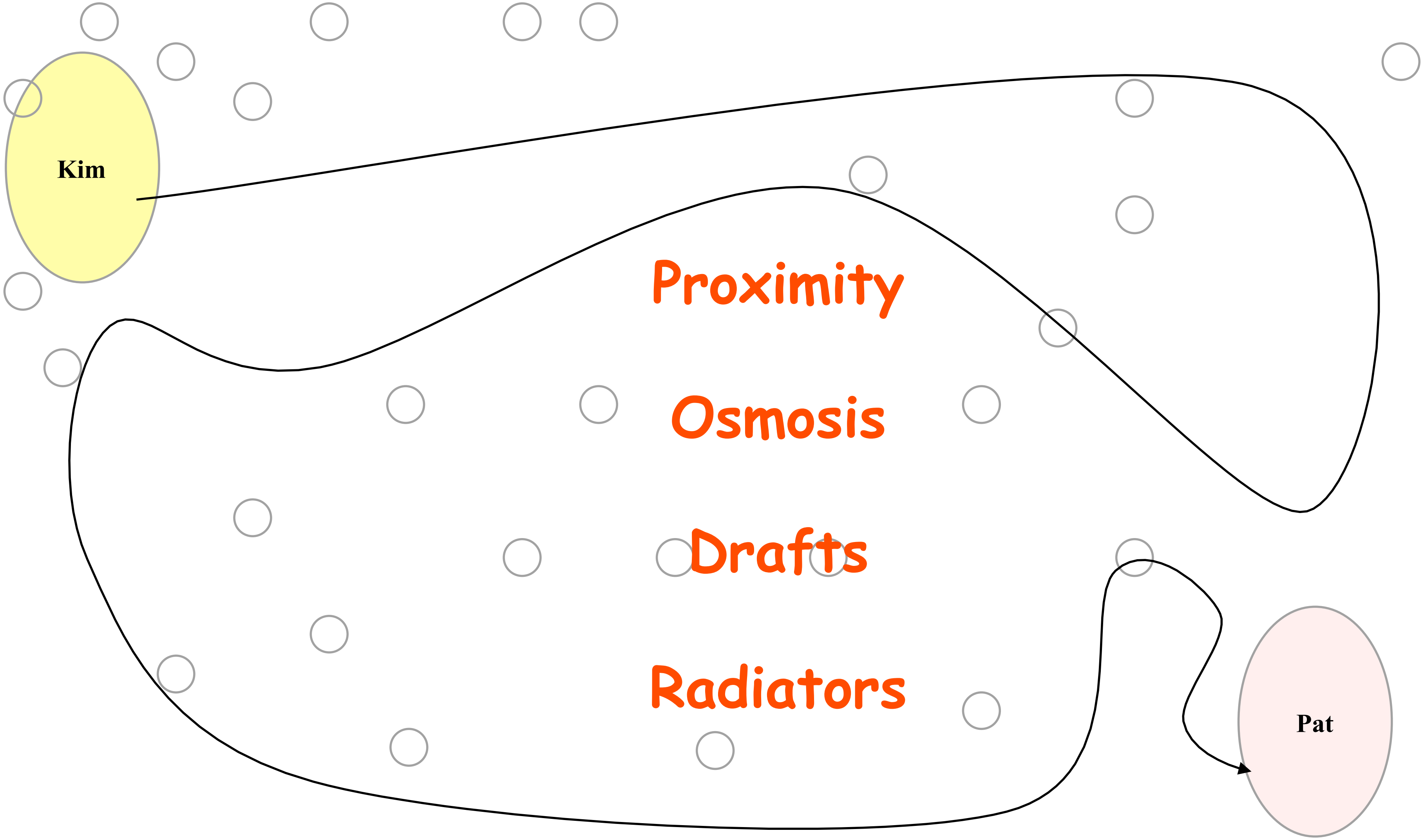
8 things you may have observed:

- Distance hurts. (So DON'T DO IT !)
- Sitting together, multimodal communication help.
- Communication has its limits.
- Both *Process* and *Product* feedback are needed.
- A process needs to allow for its own evolution.
- Action & Feedback help reduce ambiguities.
- Pause and reflect to get better.
- One possible *reflection workshop* technique.

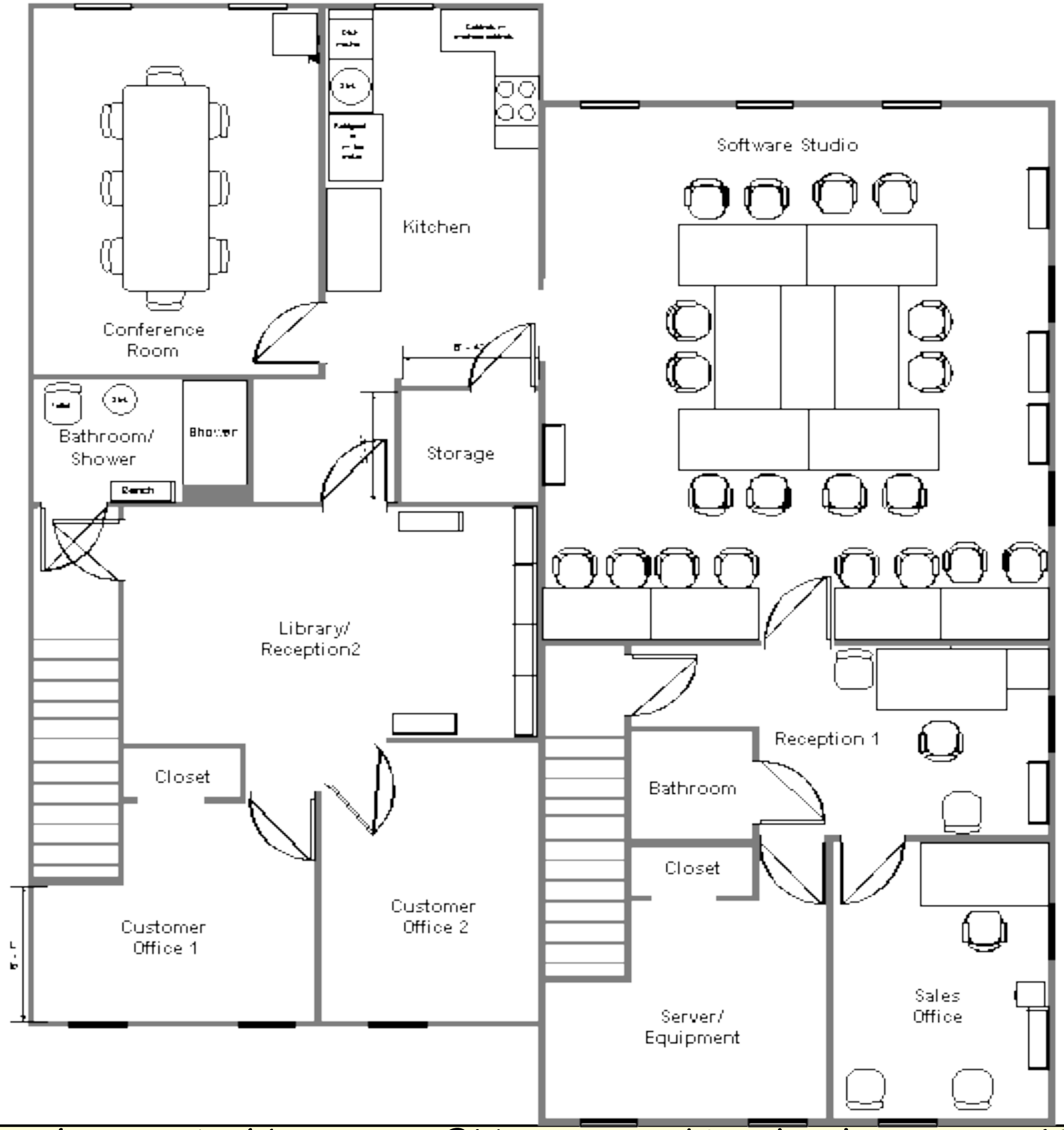
Things you may take away from today:

- The Cooperative Game vocabulary
- Manage the *incompleteness* of communication
- Take advantage of face-to-face communication.
- Pause and reflect on your Process with a ***Reflection workshop***
- Distance hurts. (So DON'T DO IT !)
- Use ***Incremental delivery*** to get Product feedback
- Use of ***information radiators***
- Relevance of ***amicability*** and ***convection currents***
- Three levels of skill (***shu, ha, ri***)
- Why methodologies need ***tuning*** to fit their ecosystem
- How to tune yours to your project and your people
- ***Timeboxing / increments*** as core technique
- ***Burn-down*** charts for visibility, ***iceberg list*** for scheduling
- ***Concurrent development*** saves time
- Optimize the rules to fit project-specific bottenecks
- All agile methodologies use ***empiriical process***, ***cooperative game*** concepts
- How to mix in Scrum, consider other named agile methodologies
- How to look for agile methodologies in your environment

COMMUNICATION uses “Convection Currents of Information”



What can we tell from an office plan?



Courtesy of RoleModel Software

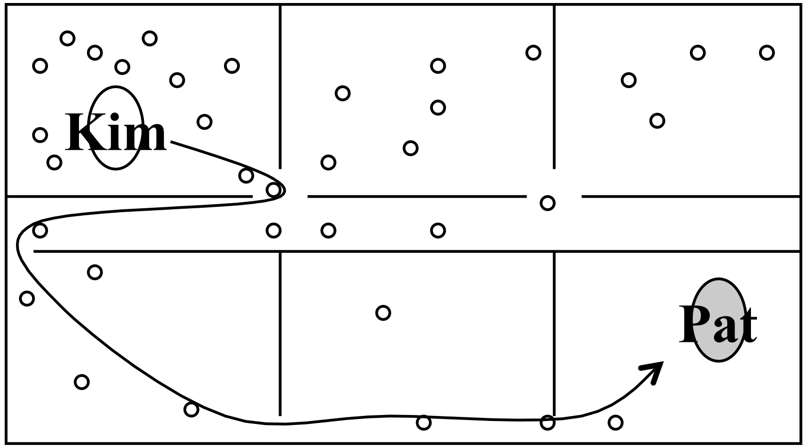
Poor office layout costs the project a lot

Programmers cost = $ 2.10 / minute (± 50%)

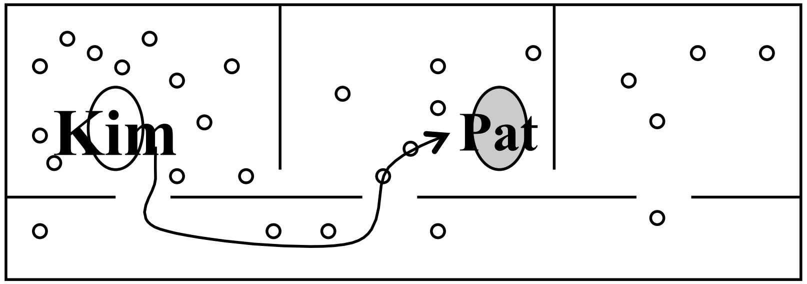
Reference pair programming @ 100 questions/week
1 minute delay / question = $ 210 / week
... for 12 people on a project = $ 2,500 / week
... for 12 month project = $ 100,000

... plus Lost Opportunity Costs for questions not asked!

People don't ask questions if they have to climb stairs.



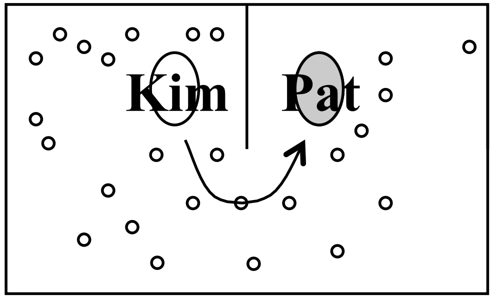
Think $300,000 / yr penalty.



Think $100,000 / yr penalty.

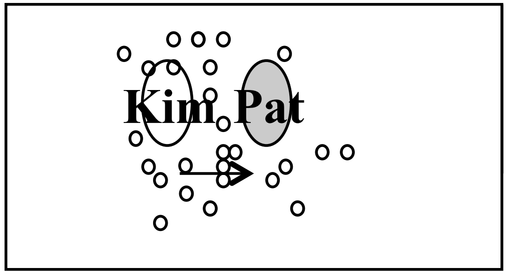
Information drifts in currents -- (not unlike perfume)

Still Effective.



Nearby programming

Most effective.



Programming in pairs

“Managing the Flow of Technology,” Thomas J. Allen,
M.I.T. Sloan School of Management
“Distance Matters,” Olson & Olson

"Nearby programming" is effective.
"Programming in pairs" is more effective.

Photo courtesy of Thoughtworks corp.

Notes: Put barriers up to reduce DRAFTS !
Morale also flows through convection currents!

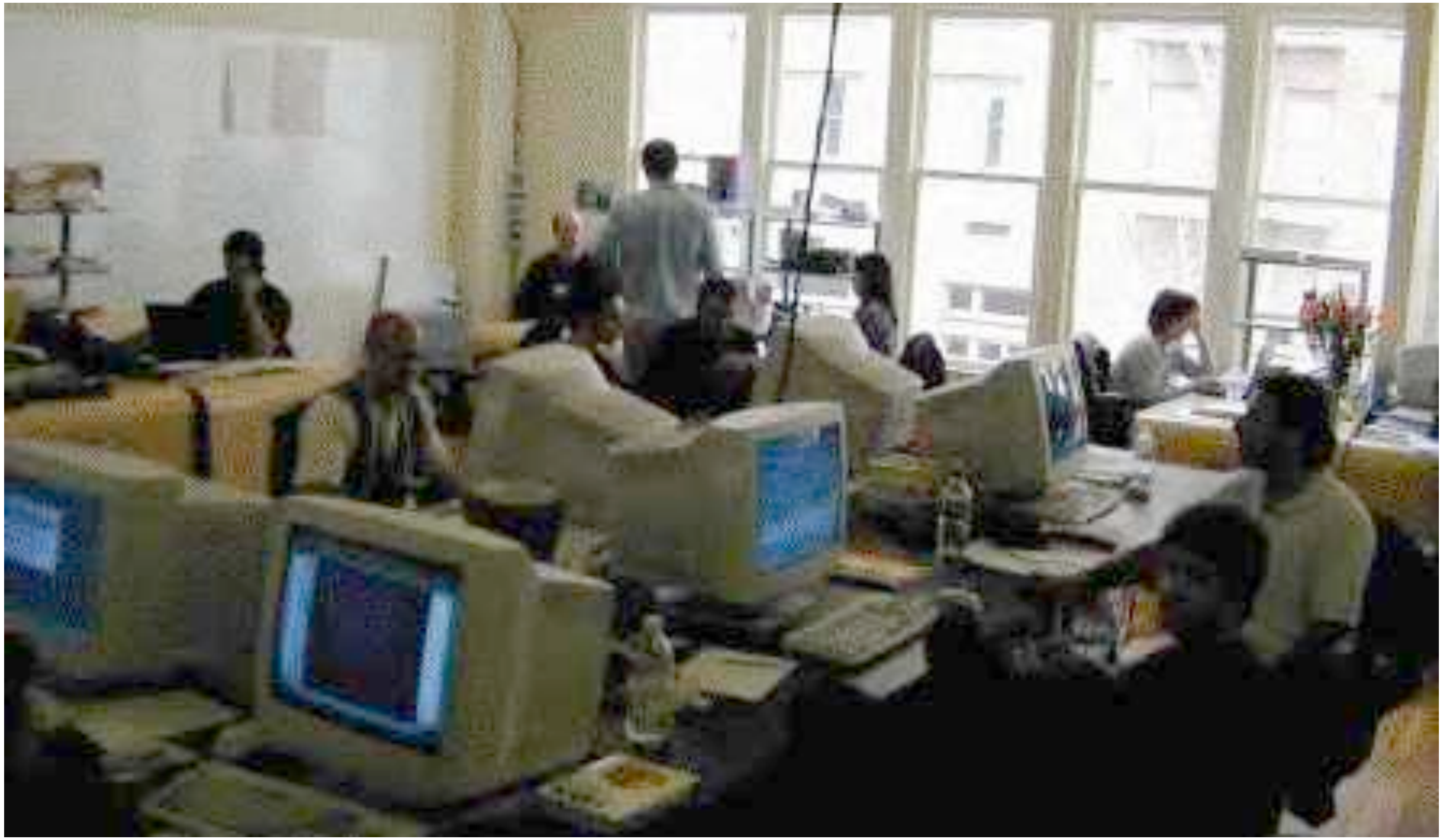
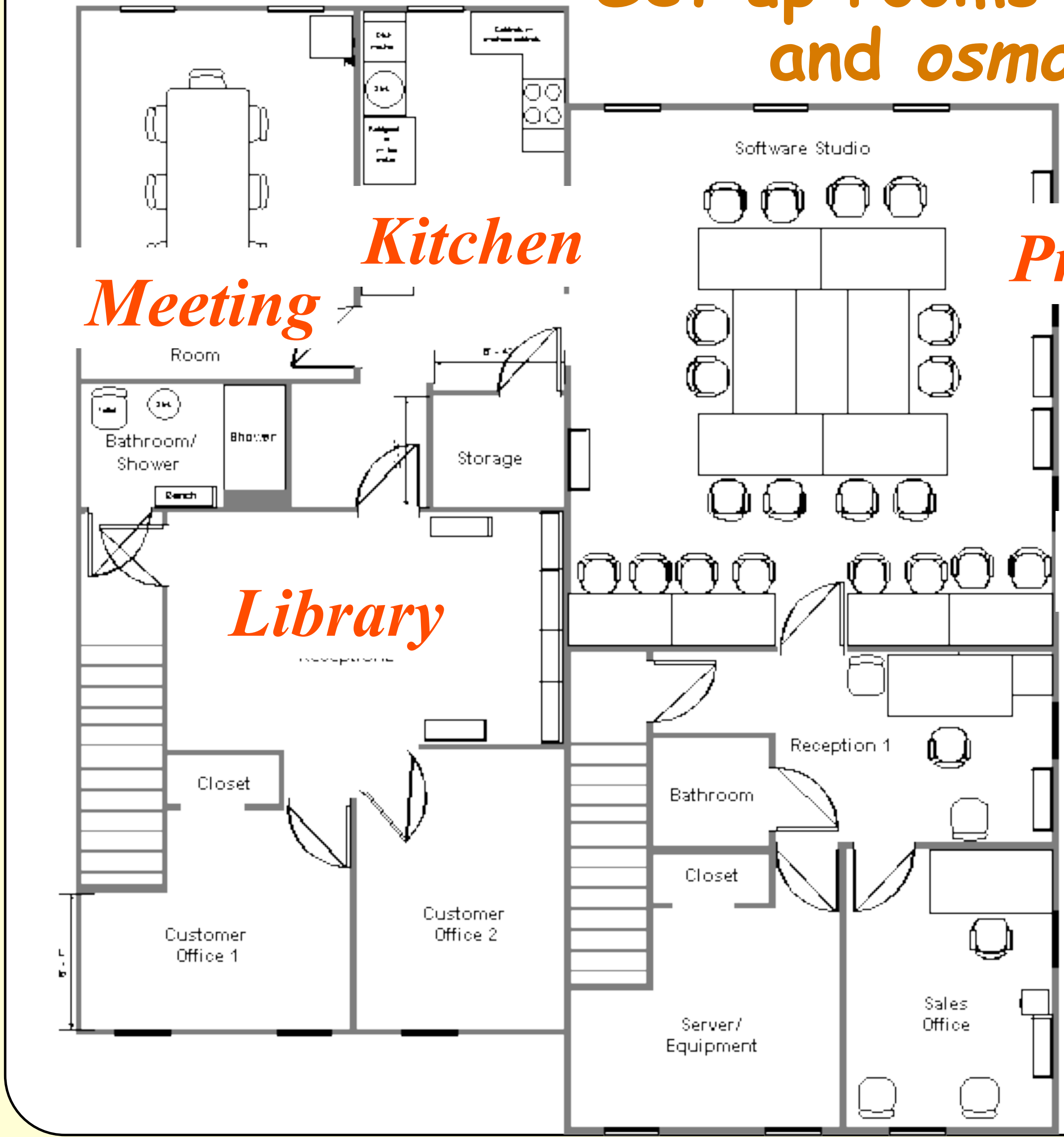


Photo courtesy of Evant corp.

Set up rooms to balance *drafts* and *osmotic communication*



Watch for:

- ***drafts***
- ***convection currents***
- ***communities***

Courtesy of Ken Auer,
RoleModel Software, Inc.

Information Radiators are the 4th piece of the Convection Currents story...

✓ Proximity

✓ Osmosis

✓ Drafts

→ Radiators

Information radiators emit passively

People get information just by walking past !

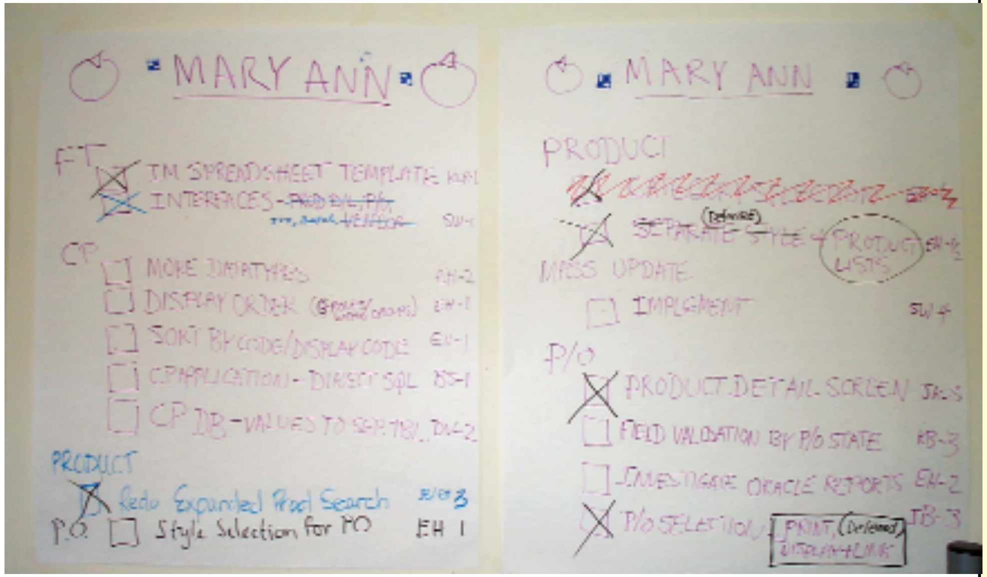
Photos courtesy of Thoughtworks

Information radiators are suited for plans, status, initiatives

Reflection workshop results

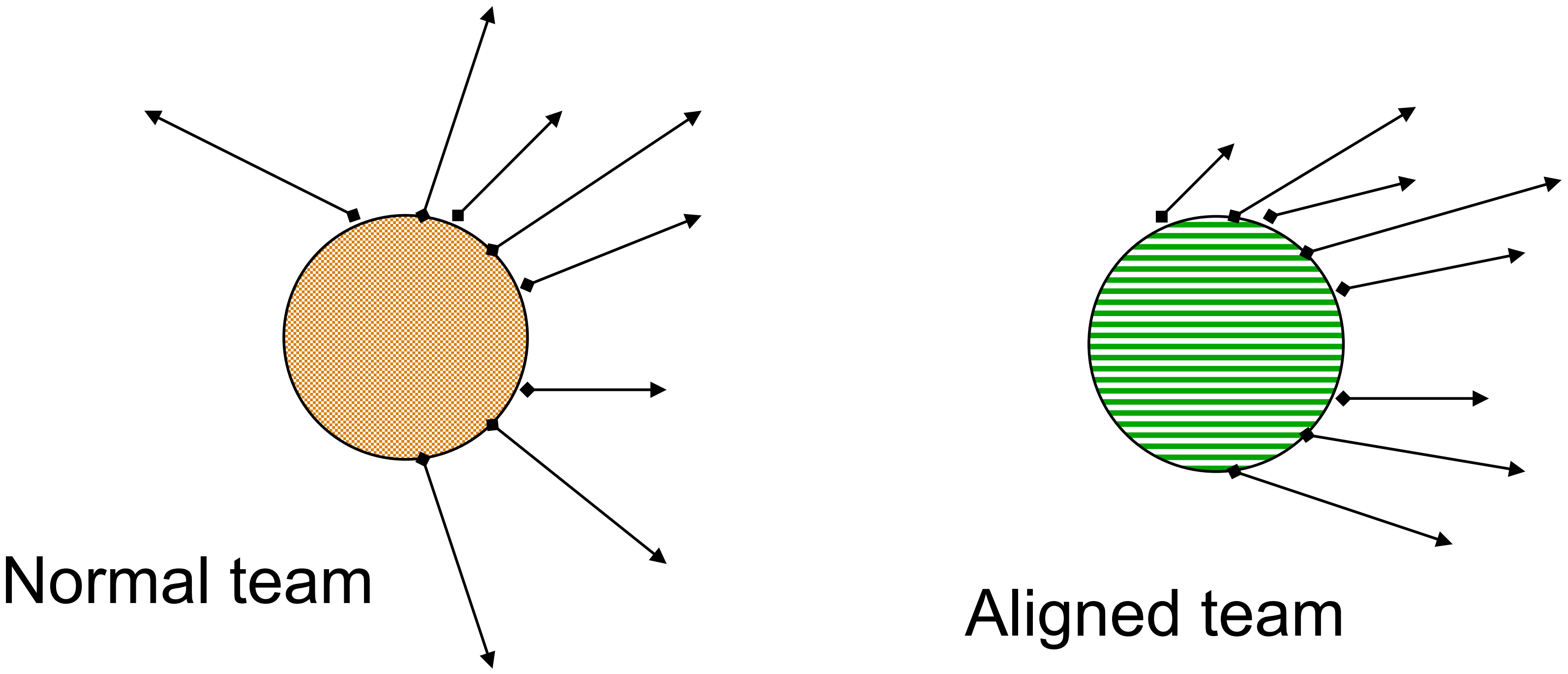
Courtesy Joshua Kerievsky

Iteration Plan


MARY ANN
F.T.
☒ IM SPREADSHEET TEMPLATE
☒ INTERFACES
CP
□ MORE DATATYPES EH-2
□ DISPLAY ORDER EH-1
□ SORT BY CODE/DISPLAY CODE EH-1
□ CP APPLICATION - DIRECT SQL
□ CP DB - VALUES TO SEP. TBL
PRODUCT
☒ Redo Expanded Prod Search
P.O. □ Style Selection for PO EH 1
MARY ANN
PRODUCT
☒ SEPARATE STYLE + PRODUCT LISTS
MASS UPDATE
□ IMPLEMENT SW 4
P/O
☒ PRODUCT DETAIL SCREEN
□ FIELD VALIDATION BY P/O STATE KB-3
□ INVESTIGATE ORACLE REPORTS EH-2
☒ P/O SELECTION PRINT, (Deferred) DISPLAY+LINK TB-3

Courtesy of Evant corp.

COOPERATION: The alignment of people's goals affects the team efficiency



(Dirty Dozen again)

COOPERATION: Amicability between people determines how quickly information moves

Amicability : Willingness to listen with good will

The "amicability index" indicates how easily information passes from one part of the organization to another.

A low amicability index implies that people block the flow of information, intentionally or through not listening well.

Amicability grows and rots fastest in osmotic communication settings !

PEOPLE are essential but non-linear active components in the development process

Weak on:

- Consistency
- Discipline
- Following instructions

Strong on:

- Communicating
- Looking around
- Copy / modify

Motivated by:

- Pride in work
- Pride in contributing
- Pride in accomplishment

People, cooperation, communication issues determine much of a project's speed

Can they easily detect something needs attention?
(*Good at Looking Around*)

Will they care enough to do something about it?
(*Pride-in-work*; *Amicability*)

Can they effectively pass along the information?
(*Proximity*; *face-to-face*, *convection currents*)

PEOPLE: The teams with the best people usually win

Talent is not skill

Skills can be developed at the rate of talent

Improve the people instead of adding people

Pay fewer, better people higher salaries

PEOPLE learn skill in a 3-stage progression
Following - Breaking away - Fluency

Level 1: Following (*Shu*) 守

Learn “a technique that works”

“Success” is following the technique

Level 2: Breaking away (*Ha*) 破

Learn limits of the technique

Learn to shift from one technique to another

Level 3: Fluency (*Ri*) 離

Shift techniques by moment

Unable to describe the techniques involved

These apply to design, management, methodology, are relevant to project staffing.

Things you may take away from today:

- The Cooperative Game vocabulary
- Manage the *incompleteness* of communication
- Take advantage of face-to-face communication.
- Pause and reflect on your Process with a ***Reflection workshop***
- Distance hurts. (So DON'T DO IT !)
- Use ***Incremental delivery*** to get Product feedback
- Use of ***information radiators***
- Relevance of ***amicability*** and ***convection currents***
- Three levels of skill (***shu, ha, ri)***
- Why methodologies need ***tuning*** to fit their ecosystem
- How to tune yours to your project and your people
- ***Timeboxing / increments*** as core technique
- ***Burn-down*** charts for visibility, ***iceberg list*** for scheduling
- ***Concurrent development*** saves time
- Optimize the rules to fit project-specific bottenecks
- All agile methodologies use ***empiriical process***, ***cooperative game*** concepts
- How to mix in Scrum, consider other named agile methodologies
- How to look for agile methodologies in your environment

Agile Software Development,Cooperative Game
Schedule of the day

I. Programming / Cooperative Games

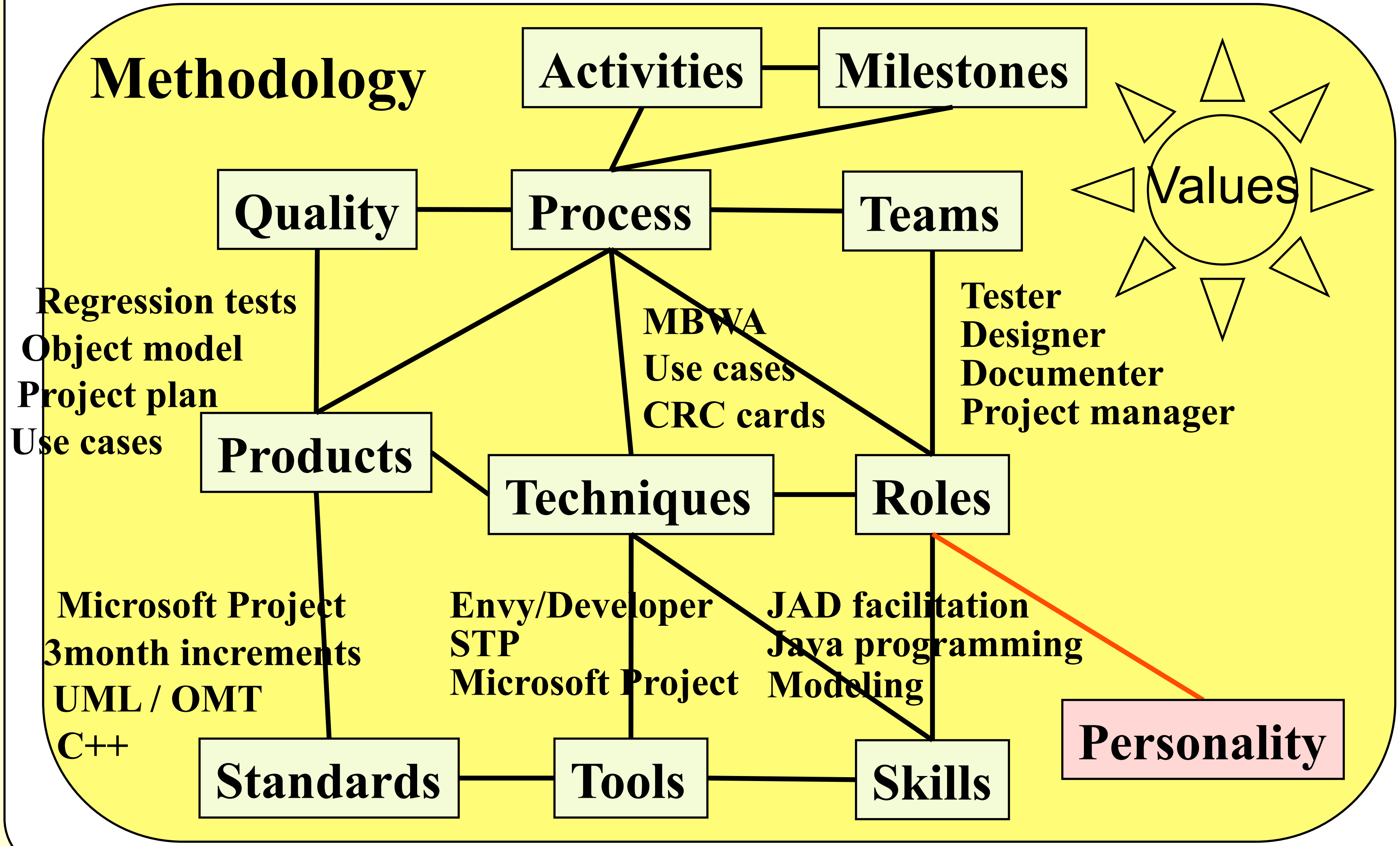
II. People / Communication / Cooperation

III. Self-Evolving Methodologies

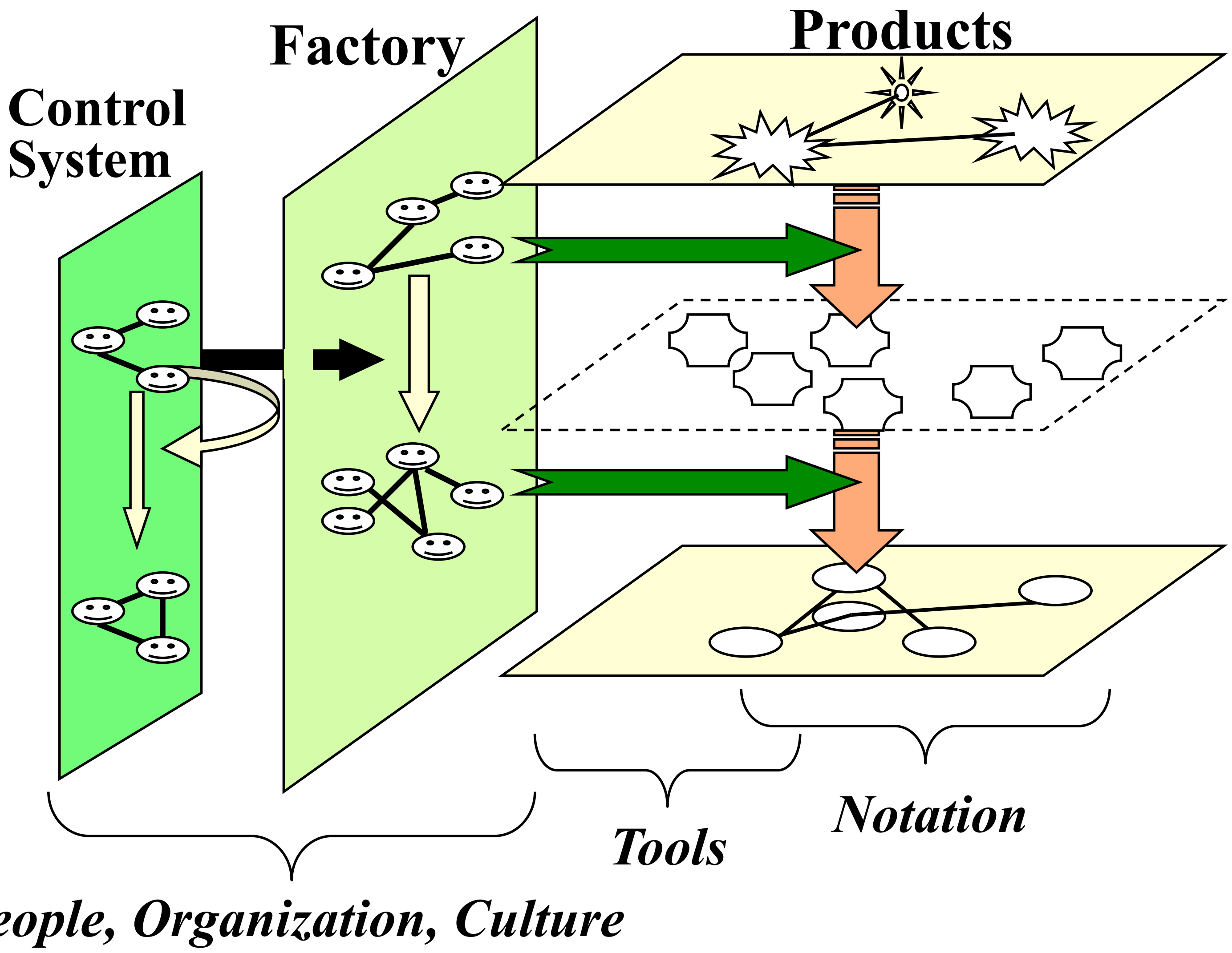
IV. Agile Techniques

V. Named Agile Methodologies

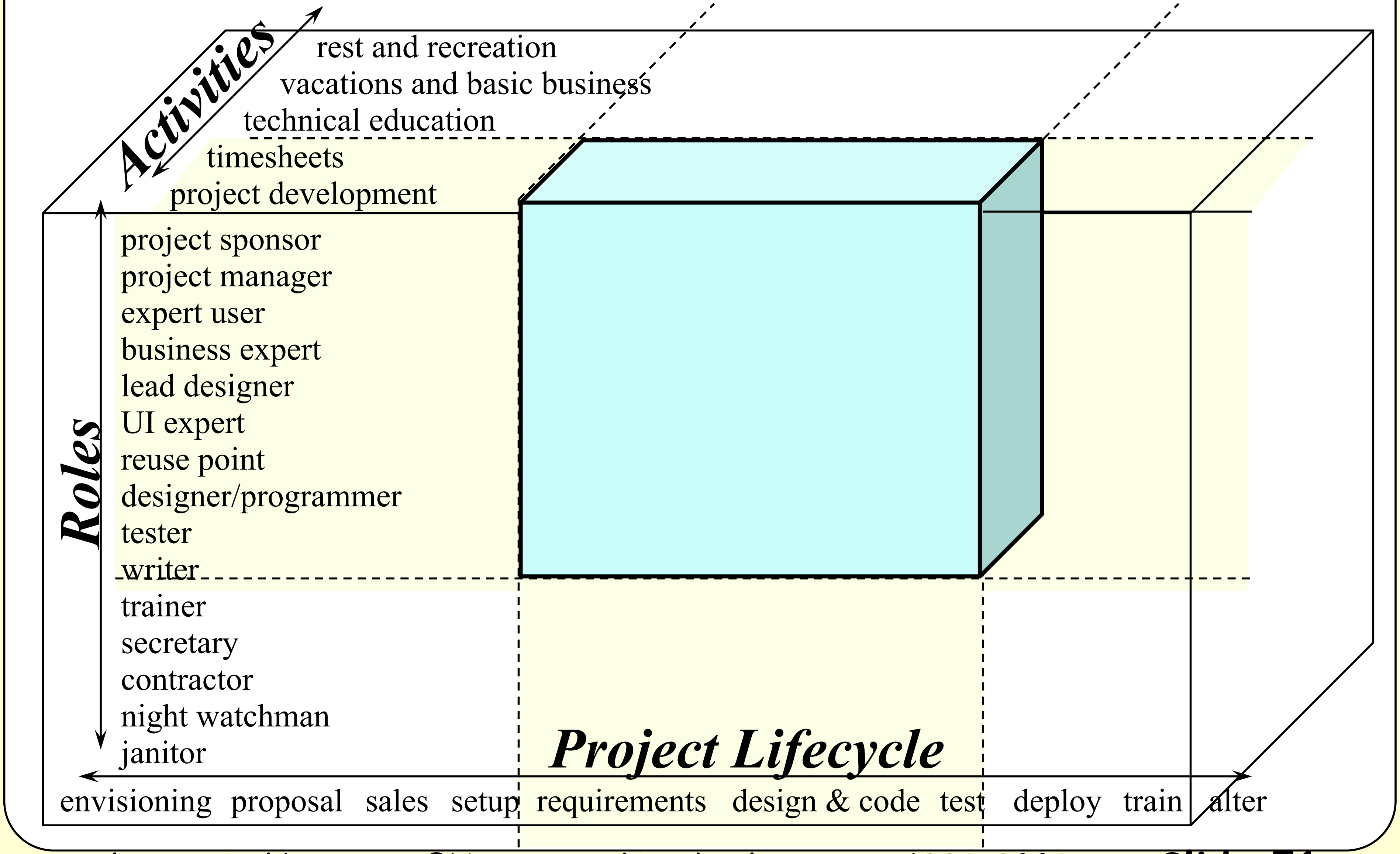
A methodology is a formal structure that idealizes personality



A methodology addresses *notations, tools,* AND people, organization and culture



Any methodology attacks a limited subset of the project lifecycle, roles, role activities



Activities

rest and recreation
vacations and basic business
technical education
timesheets
project development

Roles

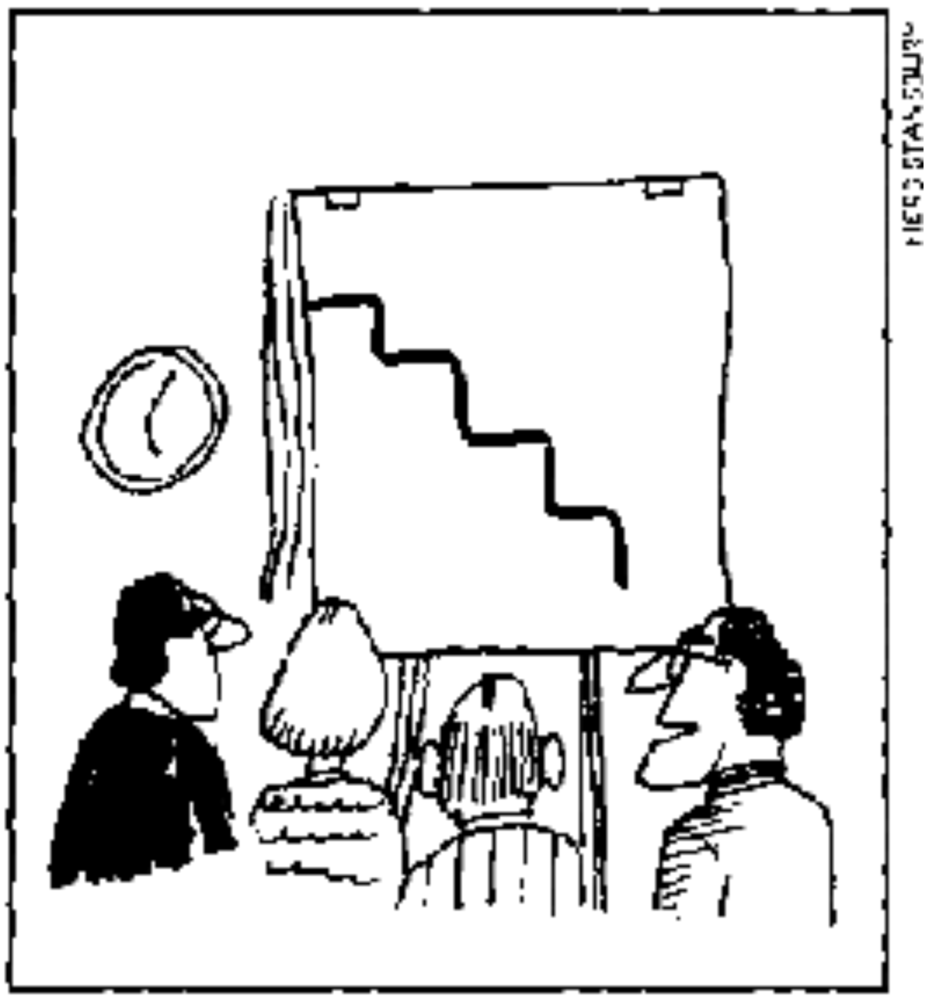
project sponsor
project manager
expert user
business expert
lead designer
UI expert
reuse point
designer/programmer
tester
writer
trainer
secretary
contractor
night watchman
janitor

Project Lifecycle

envisioning proposal sales setup requirements design & code test deploy train alter

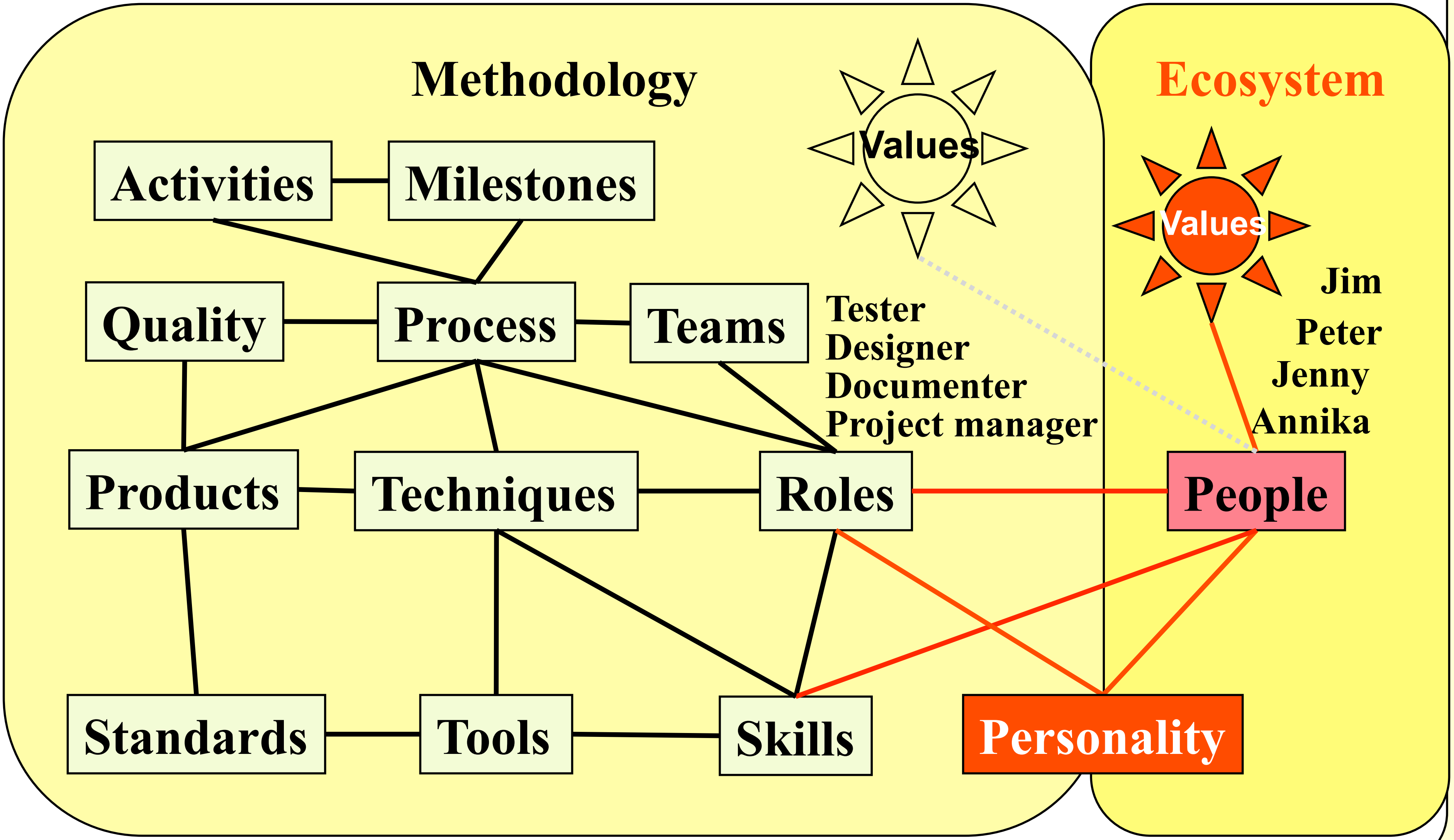
A methodology meets its limits when it meets *-people-*

“Your strategic plan, brilliant in concept and magnificent in execution, isn’t working.”



Reprinted with permission from Executive Smart Charts, copyright © 1993 by Herb Stansbury, Berrett-Koehler Publishers, Inc., San Francisco, CA. All rights reserved. 1-800-929-2929

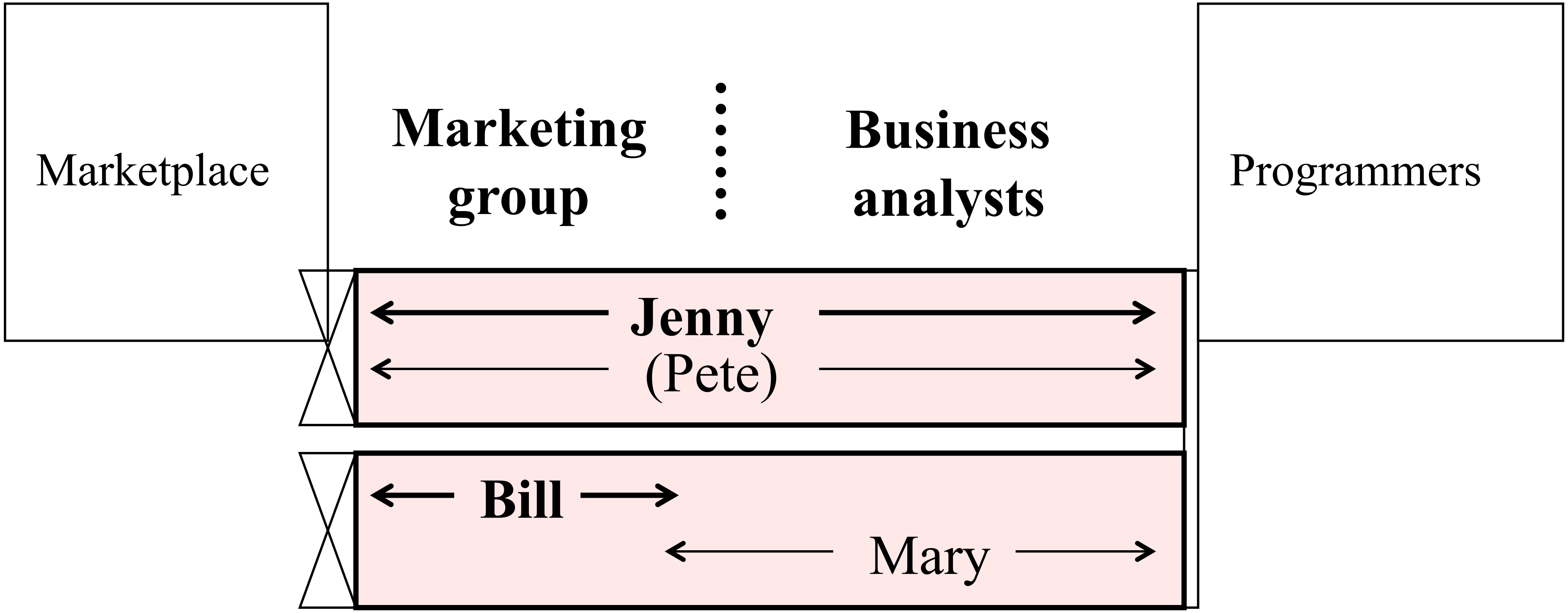
A methodology is a formula across people, but People are stuffed full of personality



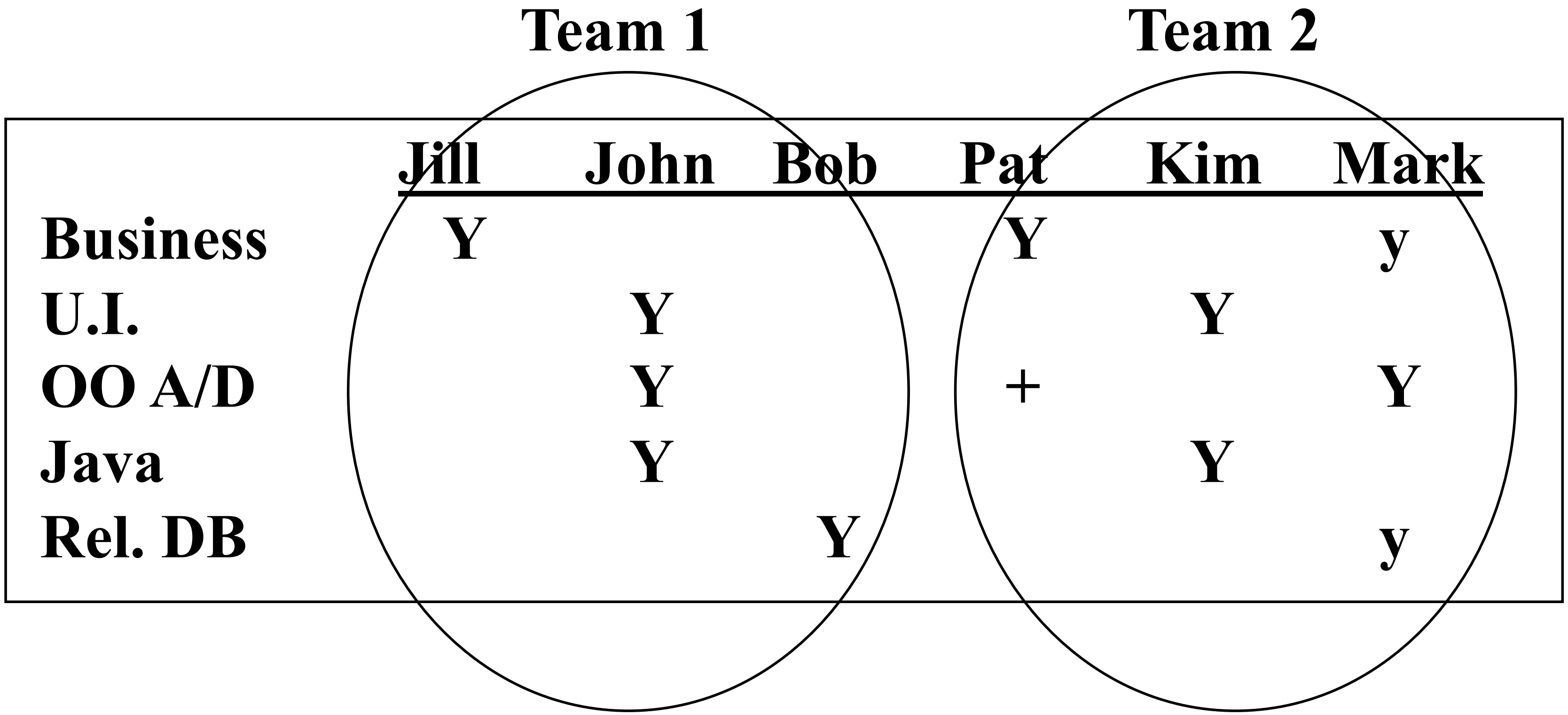
People come in different shapes.

The person who shows up may not fit the role's profile.

Level 1 practice: Stick to the methodology
Level 2, 3 practice: Adapt to the situation



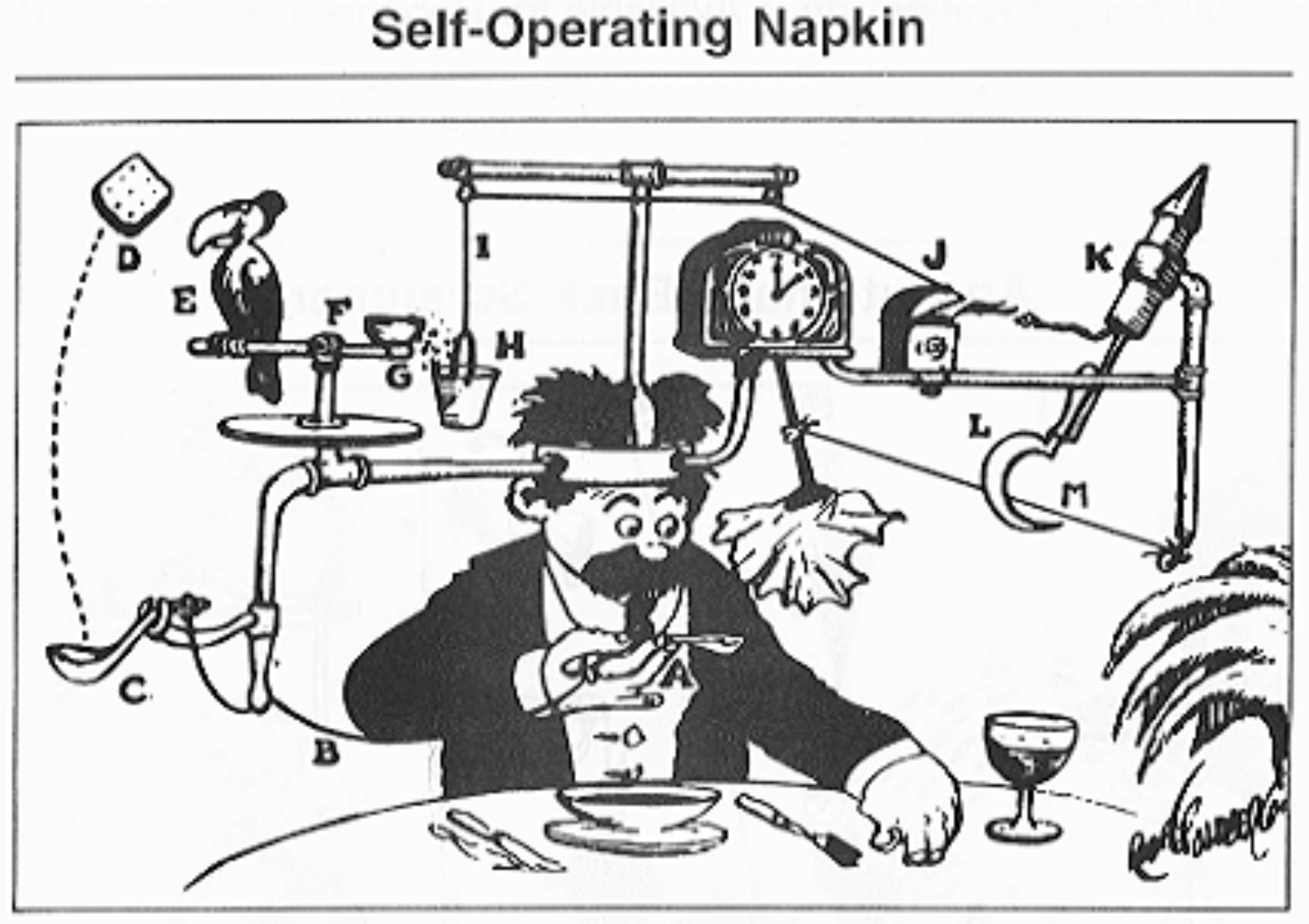
Consider clustering subteams around skills with an Assignment X Skills matrix



	Jill	John	Bob	Pat	Kim	Mark
Business	Y			Y		y
U.I.		Y			Y	
OO A/D		Y		+		Y
Java		Y			Y	
Rel. DB			Y			y

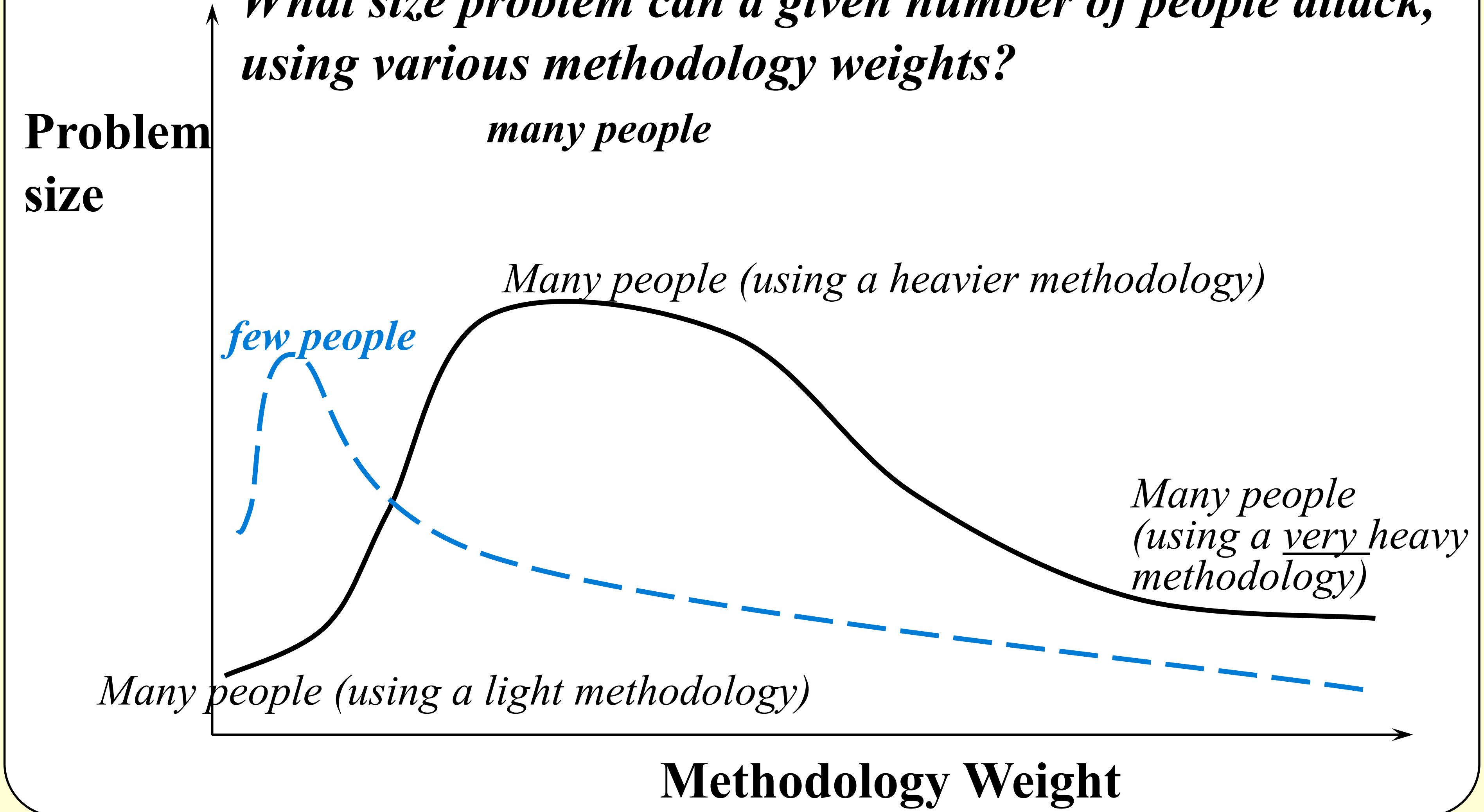
This is the [Assignment x Skills] matrix

Most methodologies are just too complicated! Start with less than you need, add to it.

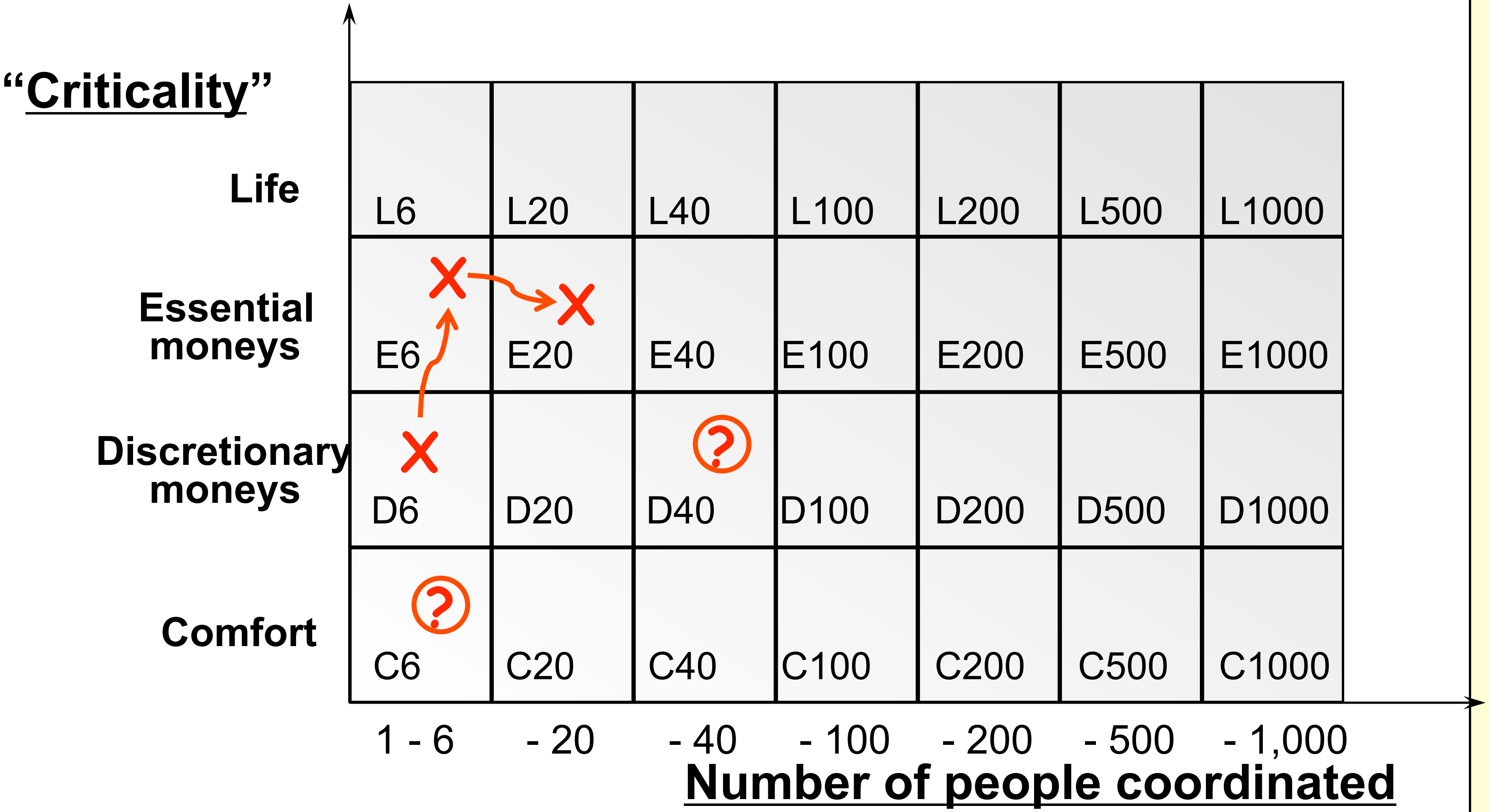


Methodologies have quick payoff, early diminishing returns. Larger teams need more

What size problem can a given number of people attack, using various methodology weights?



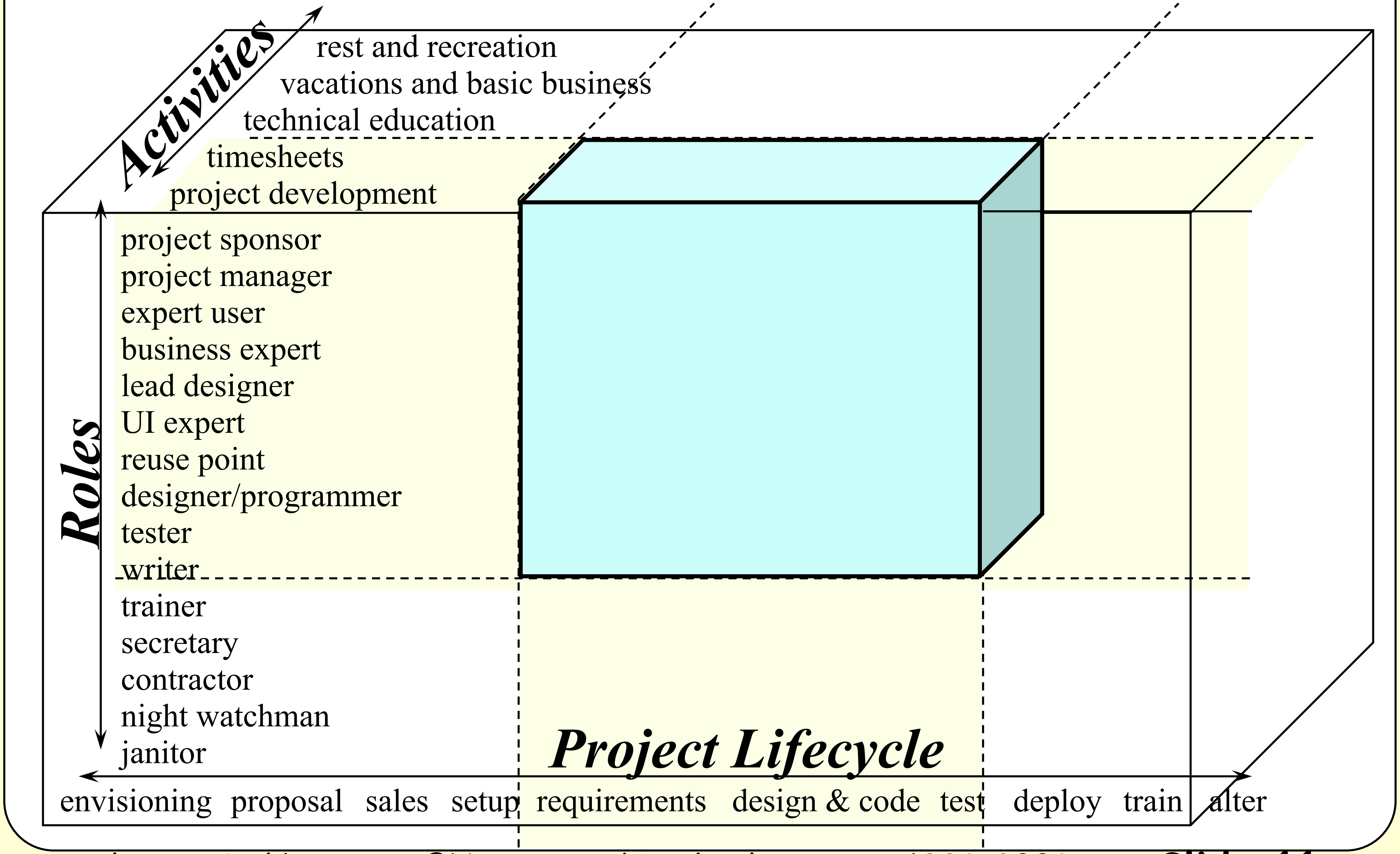
No methodology will fit the whole company. Select one to suit your project



"Criticality"							
Life	L6	L20	L40	L100	L200	L500	L1000
Essential moneys	X E6	X E20	E40	E100	E200	E500	E1000
Discretionary moneys	X D6	D20	? D40	D100	D200	D500	D1000
Comfort	? C6	C20	C40	C100	C200	C500	C1000
	1 - 6	- 20	- 40	- 100	- 200	- 500	- 1,000

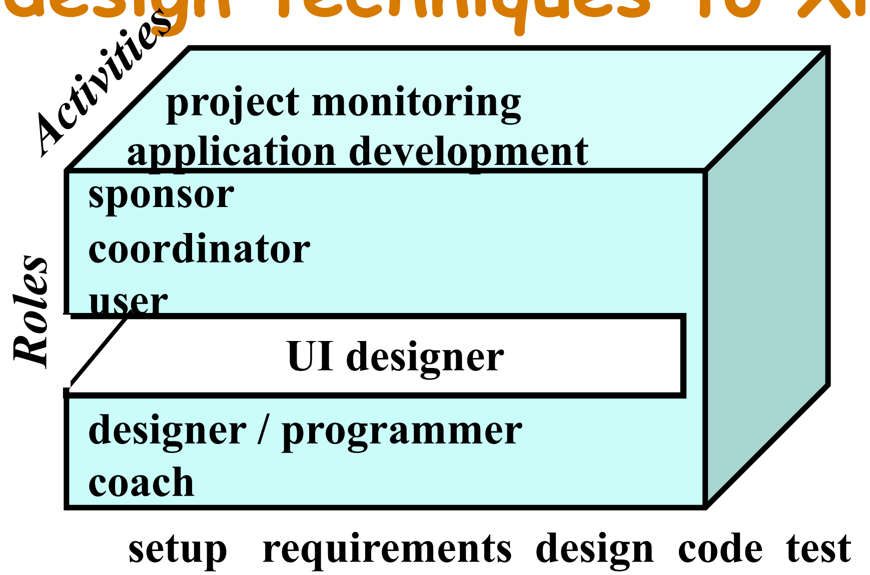
Number of people coordinated

Any methodology attacks a limited subset of the project lifecycle, roles, role activities

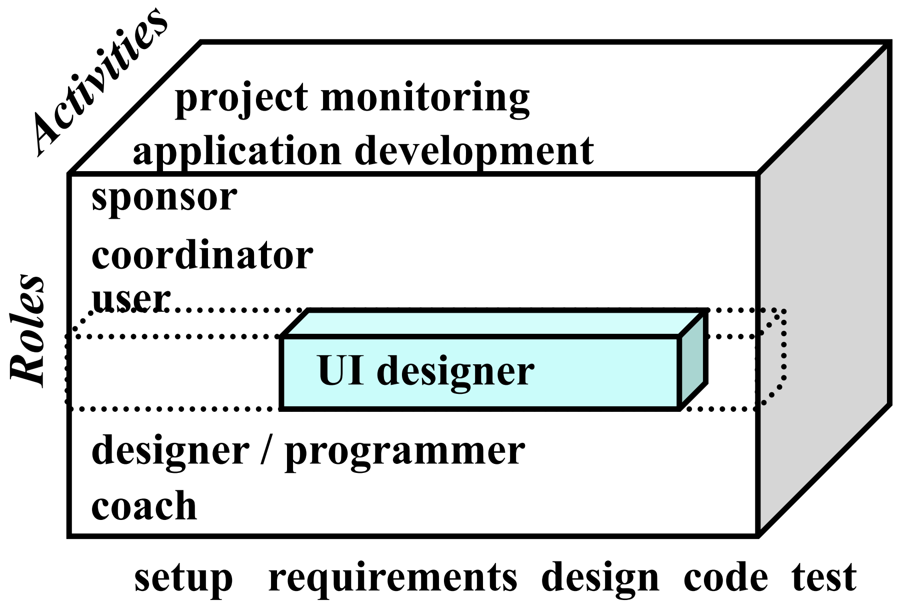


You can mix in complementary segments. e.g. add UI design techniques to XP

XP



Usage-centered design



Most importantly, a methodology needs to allow itself to EVOLVE

Almost none do

A methodology encapsulates the best that the methodologist knows.

"The word methodology begins with Me."(Weinberg)

"Let's talk about Me for a minute." (Marick, STQE)

But the Best changes over time.
Ergo, the methodology must change over time.

How?

Crystal self-adapts: methodology tuning early; reflection workshops monthly/quarterly

Start of project:

- Choose an iteration (increment) duration,
- Interview people to learn key issues, hazards.
- Hold tuning workshop to build starter set of rules.
- That is your "starter" methodology.

Get feedback periodically:

- Hold "reflection workshop" periodically:
  - monthly / quarterly / mid- & post-increment;
  - one hour, half hour, half day
- Update the project with new rules and conventions

Post the results prominently for all to see!

- (Hidden results are no results)

Reflection technique: Write in this chart. What worked? What might you try next time?

Keep These	Try These
Ongoing Problems	

(this is a core technique you should use every month on every project !)

Things you may take away from today:

- The Cooperative Game vocabulary
- Manage the *incompleteness* of communication
- Take advantage of face-to-face communication.
- Pause and reflect on your Process with a *Reflection workshop*
- Distance hurts. (So DON'T DO IT !)
- Use *Incremental delivery* to get Product feedback
- Use of *information radiators*
- Relevance of *amicability* and *convection currents*
- Three levels of skill (*shu, ha, ri)*
- Why methodologies need *tuning* to fit their ecosystem
- How to tune yours to your project and your people
- *Timeboxing / increments* as core technique
- *Burn-down* charts for visibility, *iceberg list* for scheduling
- *Concurrent development* saves time
- Optimize the rules to fit project-specific bottenecks
- All agile methodologies use *empiriical process*, *cooperative game* concepts
- How to mix in Scrum, consider other named agile methodologies
- How to look for agile methodologies in your environment

Agile Software Development, Cooperative Game
Schedule of the day

I. Programming / Cooperative Games

II. People / Communication / Cooperation

III. Self-Evolving Methodologies

IV. Agile Techniques

V. Named Agile Methodologies

The Agile Manifesto embodies the cooperative game ideas

“We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

Individuals and interactions *over* Processes and Tools
Working software *over* Comprehensive documentation
Customer collaboration *over* Contract negotiation
Responding to change *over* Following a plan

That is, while there is value in the items on the right, we value the items on the left more. “

(©2001, Beck, Beedle, van Bennekum, Cockburn, Cunningham, Fowler, Grenning, Highsmith, Hunt, Jeffries, Kern, Marick, Martin, Mellor, Schwaber, Sutherland, Thomas)

Agile Software Development

How to Cheat (Legally) and Win

Alistair Cockburn
Humans and Technology

alistair.cockburn@acm.org
http://Alistair.Cockburn.us

If you wanted to 'cheat' and win, what could you do?

Hire only the best people;
Seat them close together, helping each other out, learning each other's skills;
Give them good tools and training as they need;
Get them close to the Customers and users, who show them what they want;
Have them show / deliver results to the users frequently for direct feedback;
Cut out the bureaucracy
Let them find inventive, inexpensive ways to document their work

(Let them work in Agile fashion)

Key techniques in Agile Development

✓ Collocate people (osmotic communication)

Access to expert users (early feedback)

Timebox & deliver (accomplishment, feedback)

Burn-down charts (visibility)

Distribute responsibilities (managers, programmers)

Daily stand-up (convection current of information)

✓ Post-iteration reflection workshop (self-evolve)

Concurrent development (respond to change)

Simpler designs (faster, easier to change)

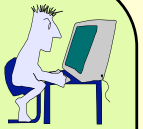
Automated testing (permits change)

Frequent / continuous integration (synchronizes)

House packing exercise

Work in groups of 4-6 people

Create an Information Radiator for Packing your House (15 minutes)



Your (American) house has 14 rooms total, including garage.
In 33 days, the movers will pick up the packed boxes.
Pack everything into boxes and put them into the garage.
The next morning you fly to India.
You cannot be late.

1. **Work in groups of 4-6 people.**
2. **Decide on a work plan for packing the house.**
3. **Create information radiator (table/graph/chart) so that you always know how well you are doing.**
4. **Show how your display looks after 18 days.**

Timeboxing is a critical element in iteration scheduling

2-week, 1-month (,quarterly) timeboxes.

Each timebox ends with integrated, tested code.

Cut scope as needed but complete on time.

- Deliver whatever you have
- Whatever you accomplished this time is a predictor of what you will accomplish next time ("Yesterday's Weather")

(some timeboxing fixes requirements, some don't)

The “burn-down” chart and variations show a project’s progress visibly and publicly

Visibility of *rate-of-progress* and *achievements* are critical project management information

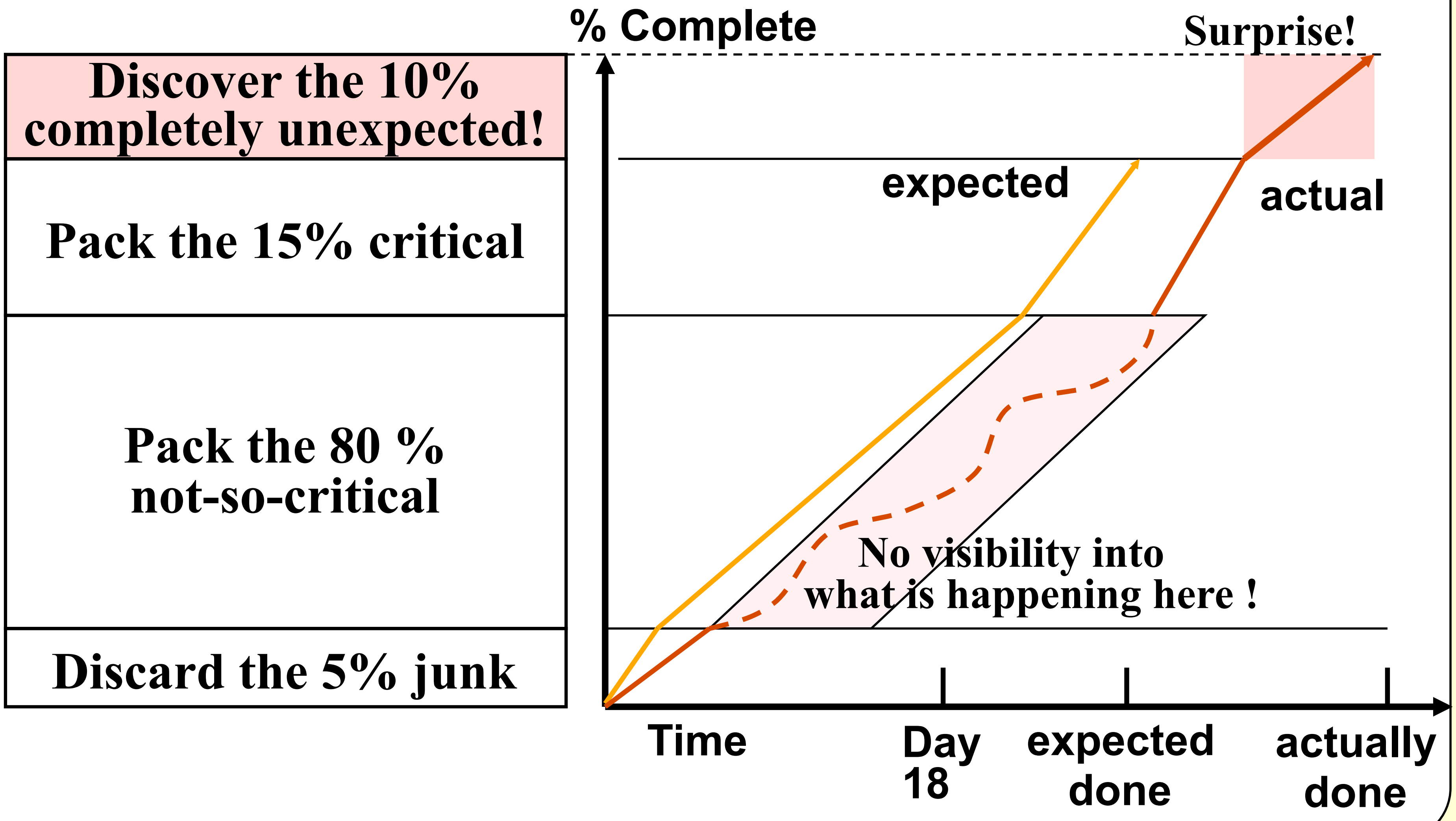
Works because task list is *fixed in size*

“Iceberg” list useful when task list changes daily

- Use spreadsheet or similar
- List tasks
- Developers estimate time, Managers prioritize
- Sort in priority order
- *Derive* which tasks are in schedule for this current iteration / delivery.
- Managers can change priorities

Method #1. The whole house, by "importance"

No view into progress & no warning of trouble



Discover the 10% completely unexpected!

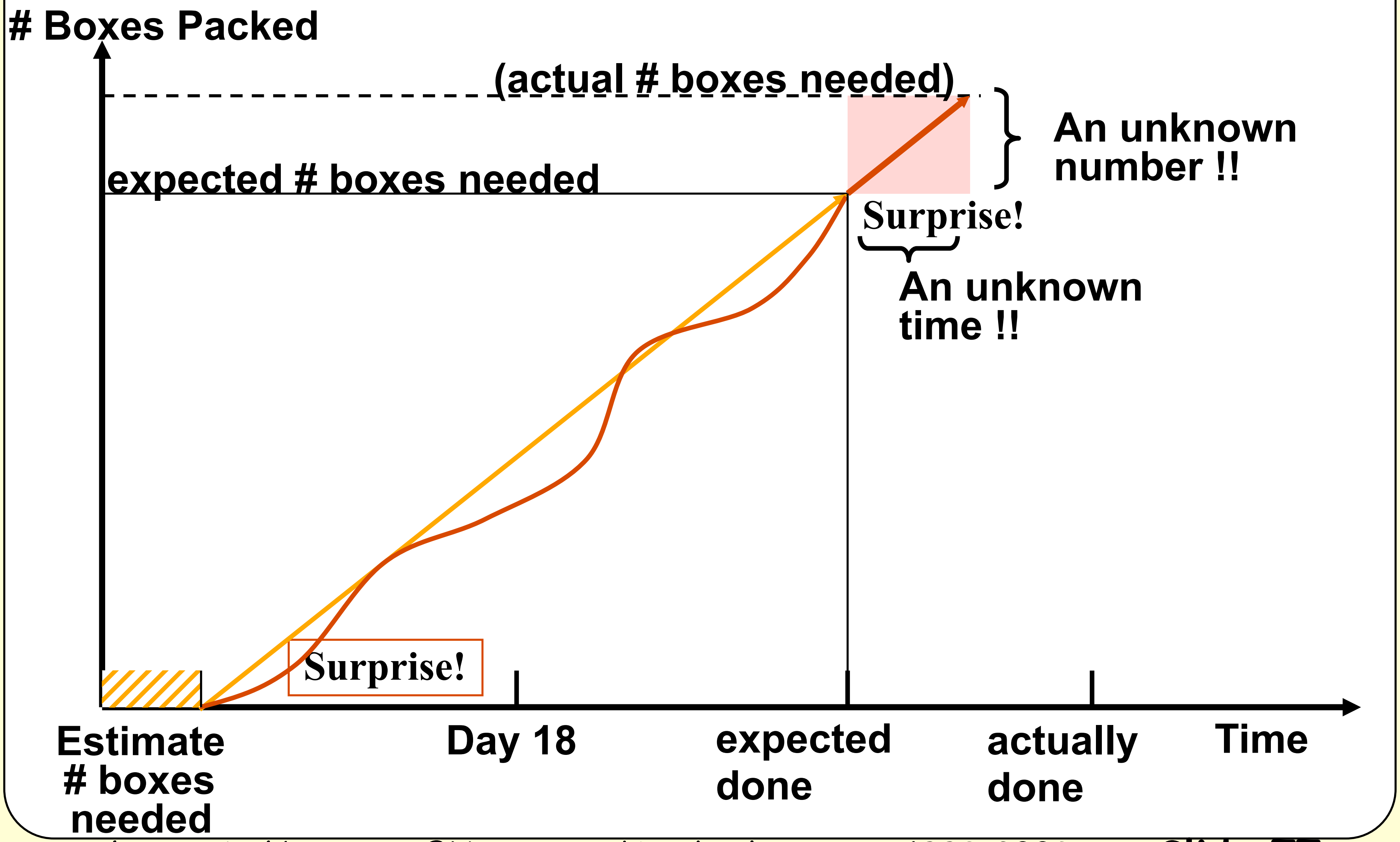
Pack the 15% critical

Pack the 80 % not-so-critical

Discard the 5% junk

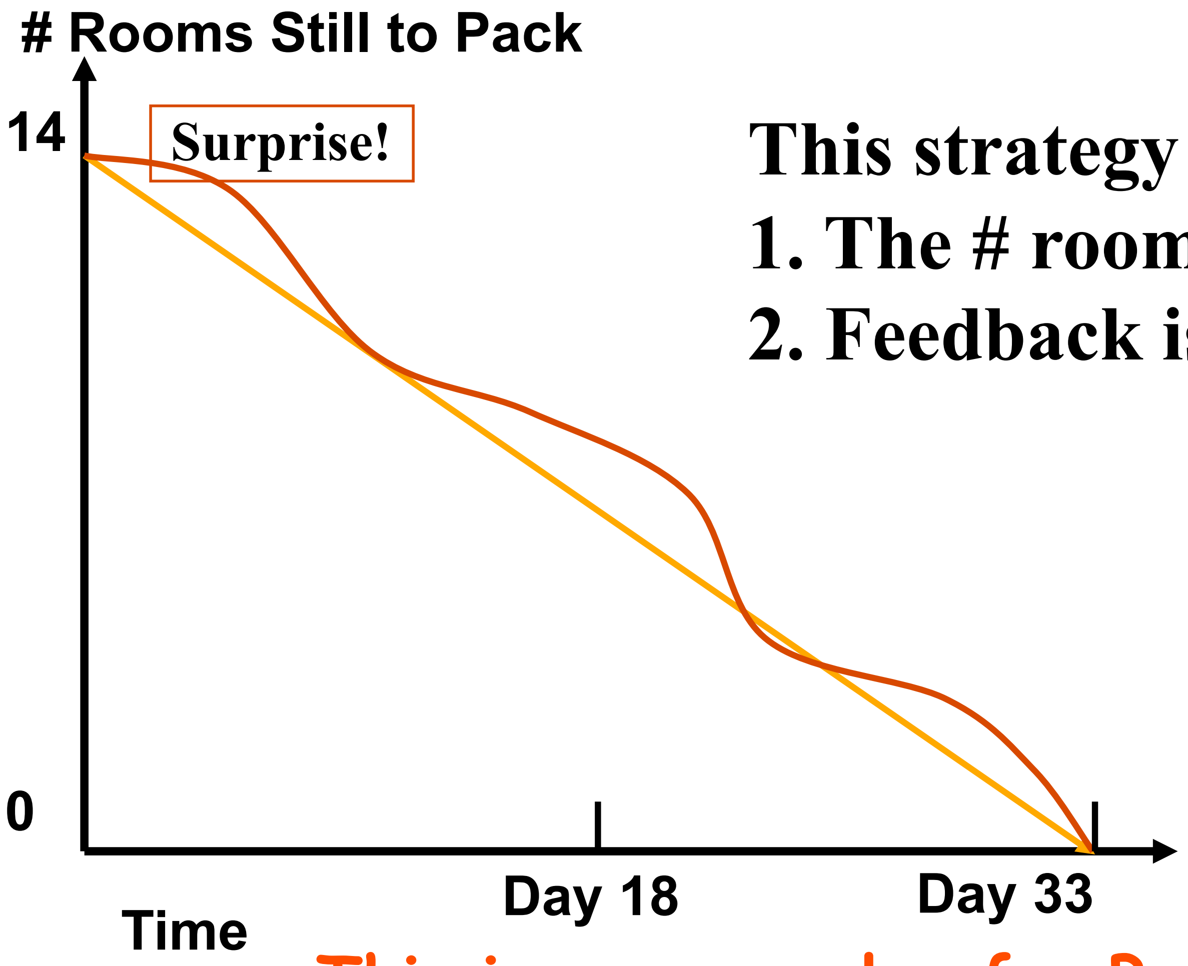
Method #2. Estimate boxes to be packed.

Good visible progress. No warning of trouble!



Method #3. Packing each room to completion.

Good visibility & early warning.

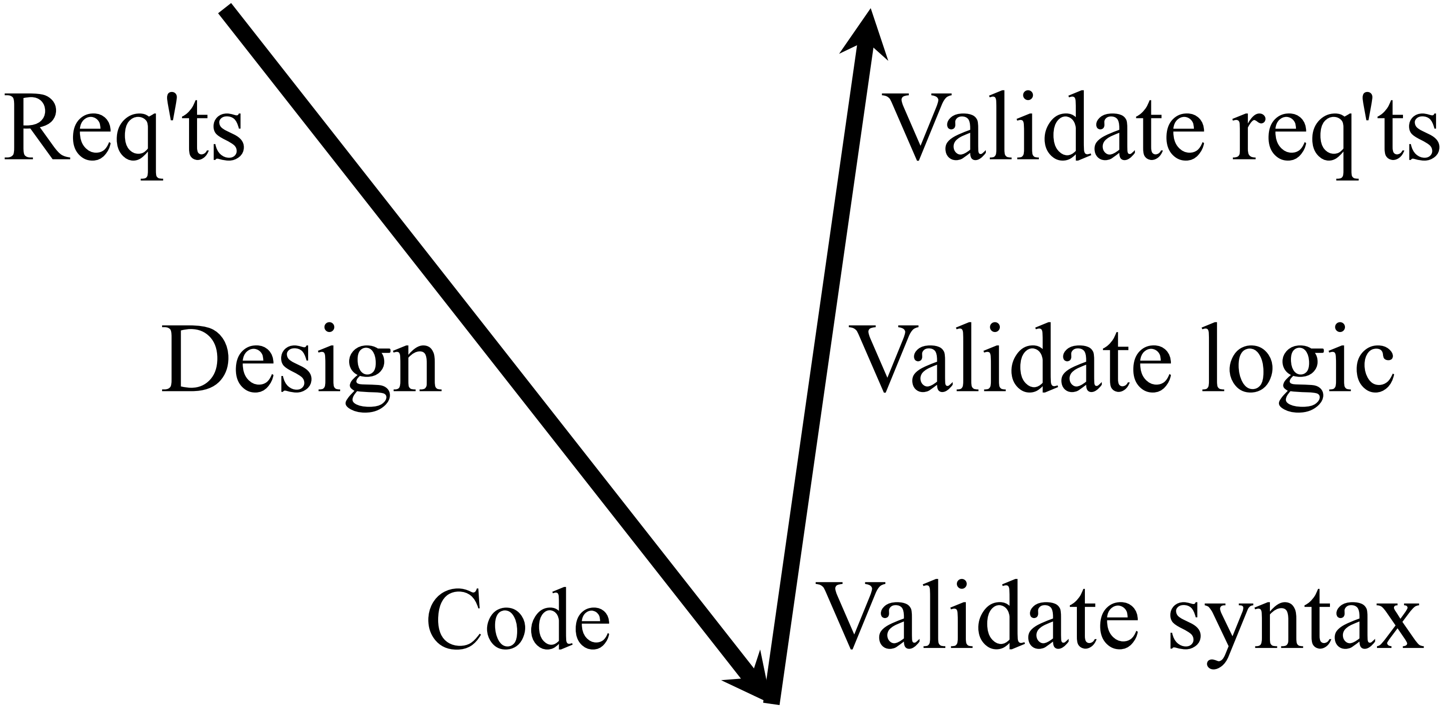


This strategy works because:

1. The # rooms won’t increase.
2. Feedback is on full process.

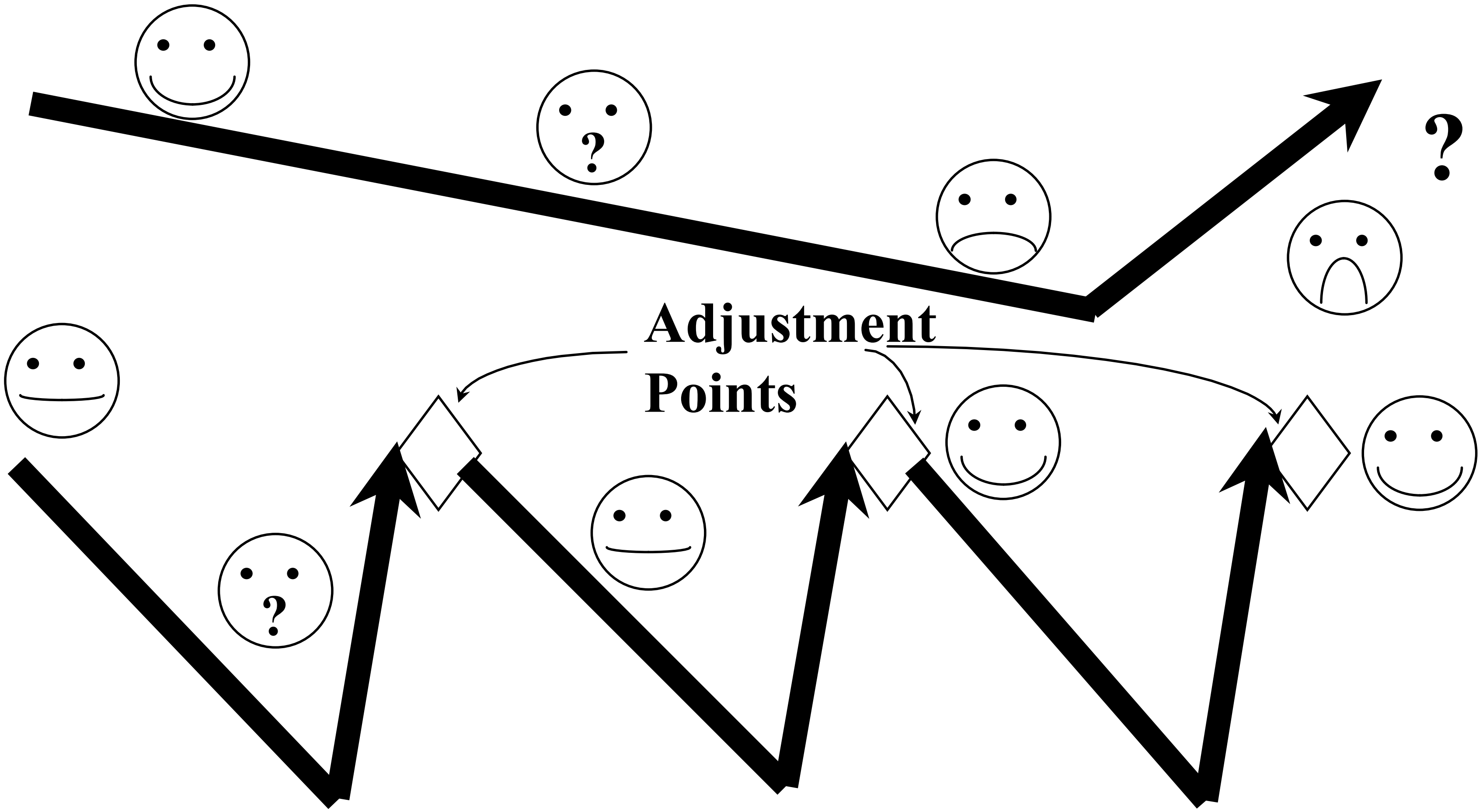
This is an example of a Burn-Down chart

Timeboxing uses the 'Validation V':
Not waterfall, but a fact of life.



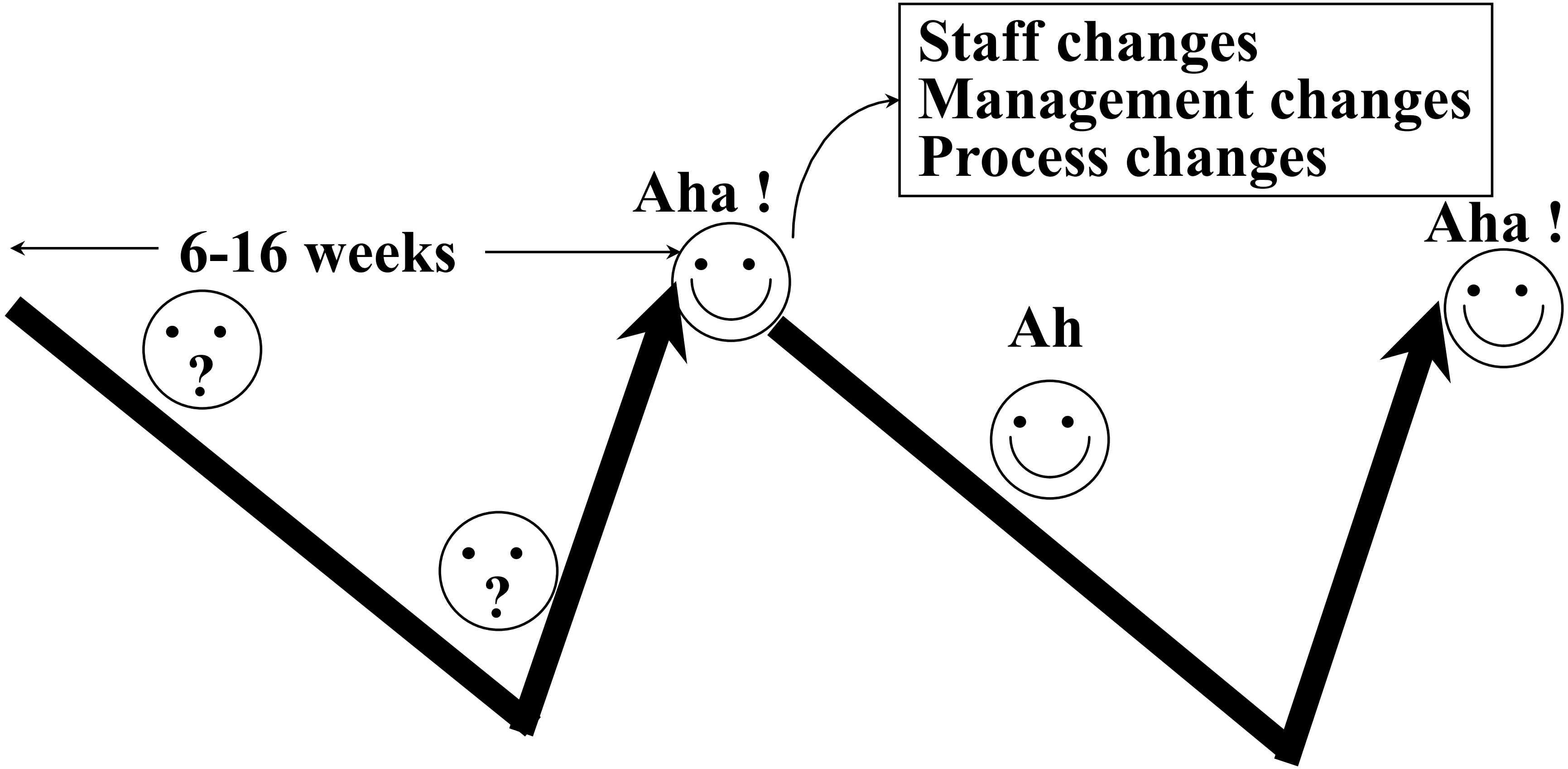
How can we use this 'fact of life'?

1 24-month V is "waterfall".
6 4-month Vs are incremental, effective.



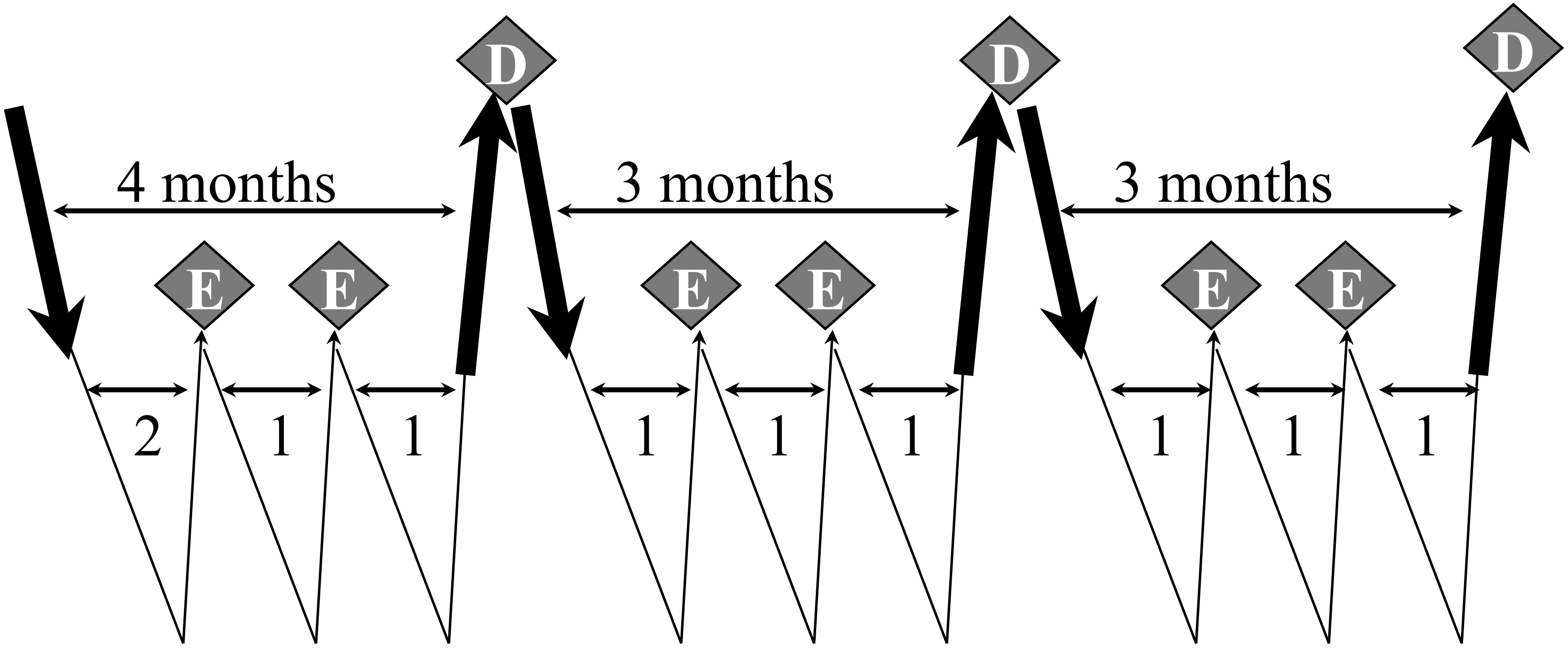
1-10

Short Vs allow focus of attention, learning, better estimation, process & team changes.



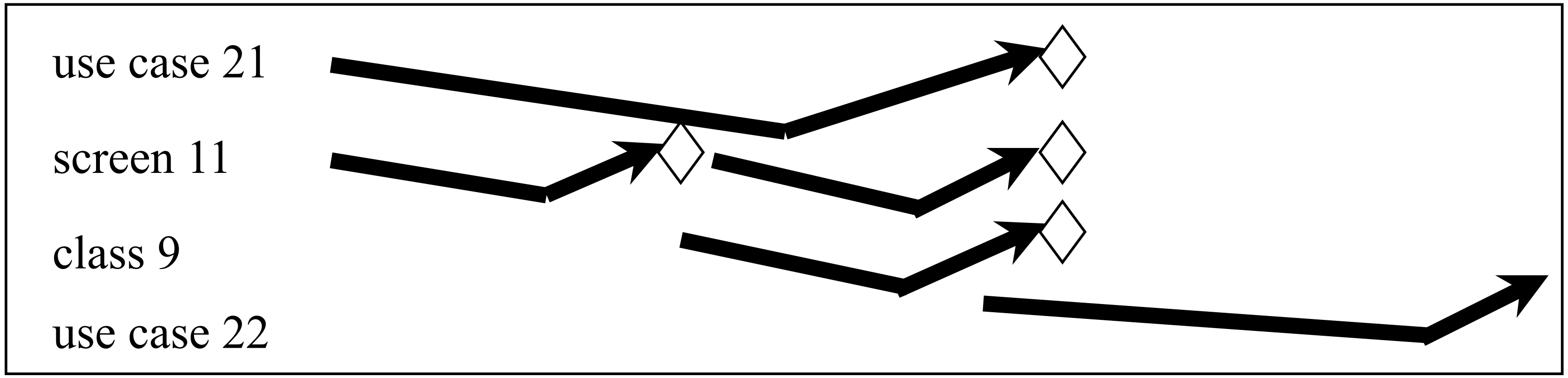
1-10

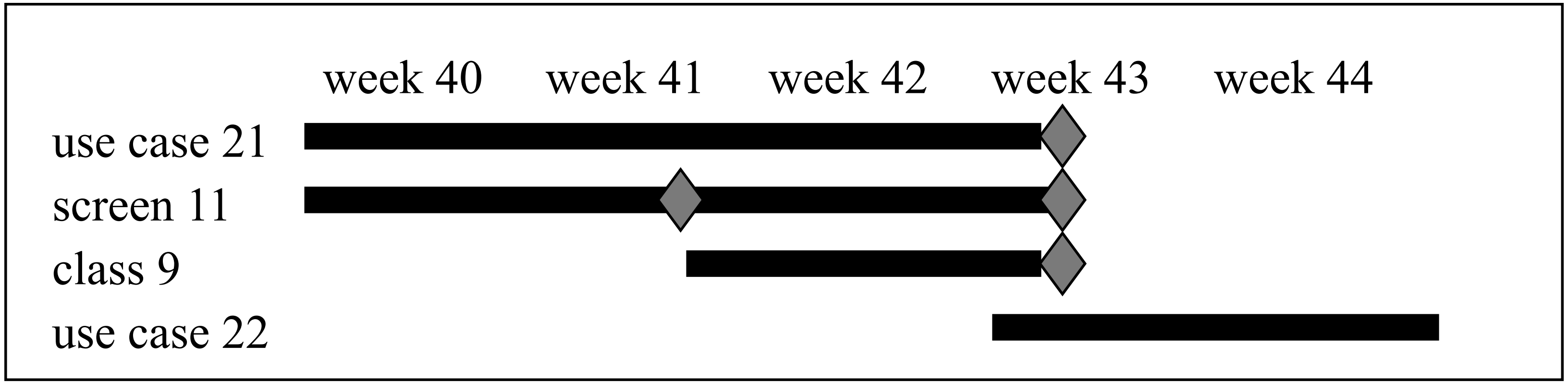
Some Vs (*iterations*) end with "examination" Some Vs (*increments*) end in "delivery" Increments should be 4 months or less.



Example of incremental plan from one project.

Every V is a major milestone for the Project Manager's calendar





"Every team must deliver working function every 2 - 4 months."

The most important single project policy to apply.

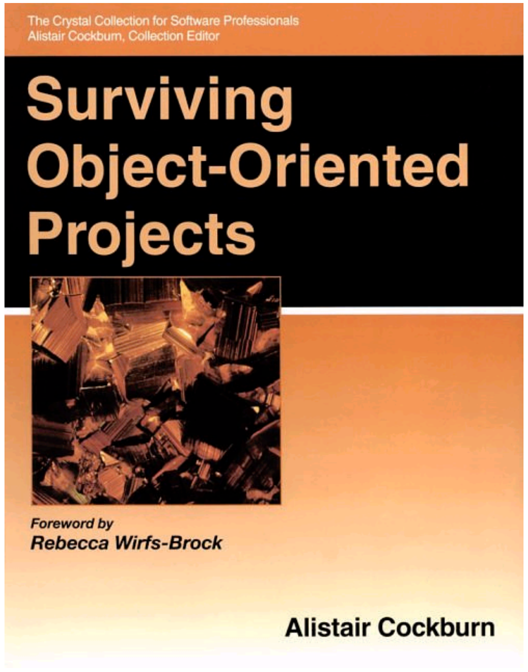
Read about incremental development:

"*Surviving Object-Oriented Projects*" pages 117 - 130

(now available in Japanese)

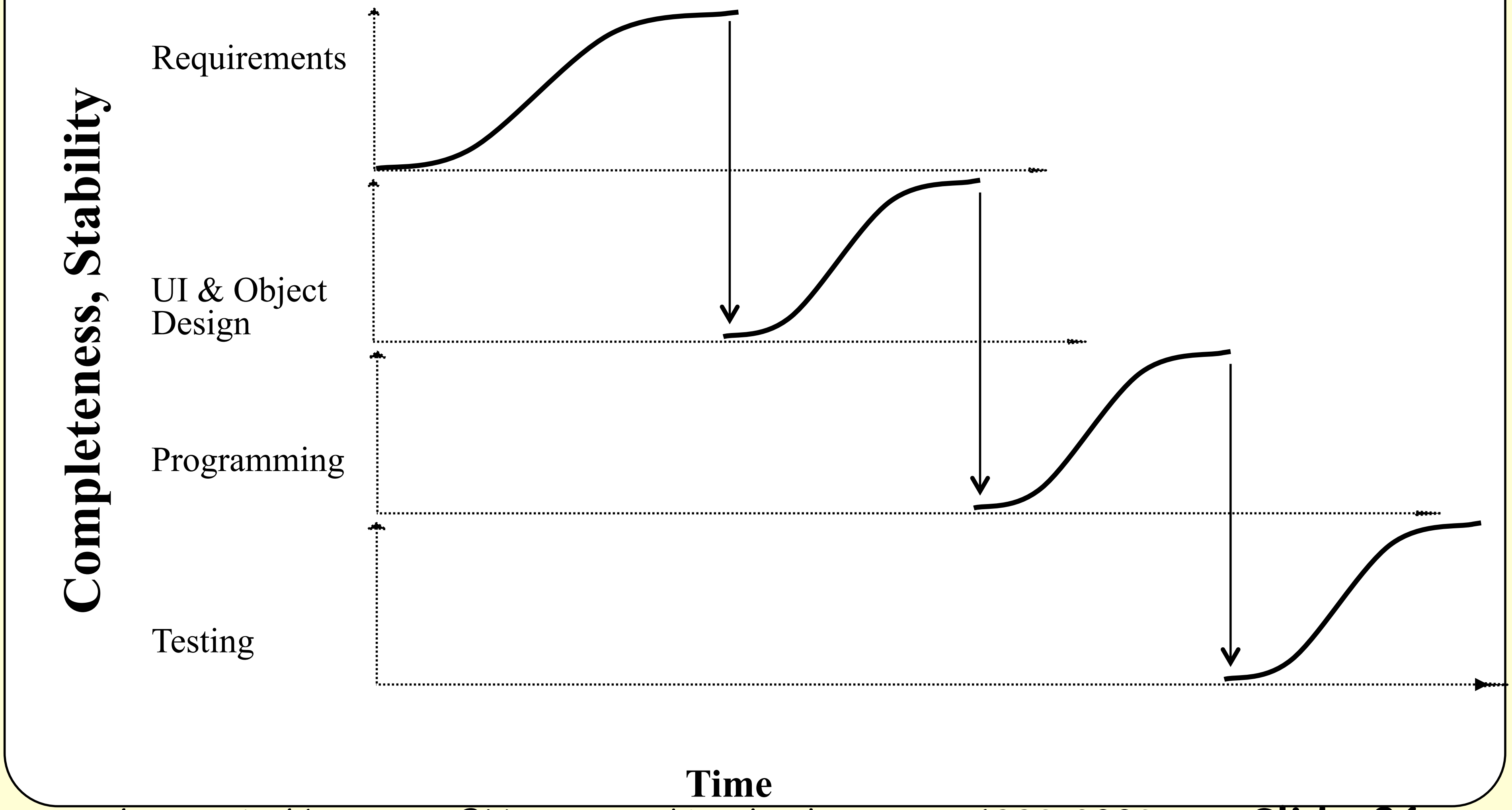
also on the web:

http://alistair.cockburn.us
/crystal/articles/vws/vwstaging.html

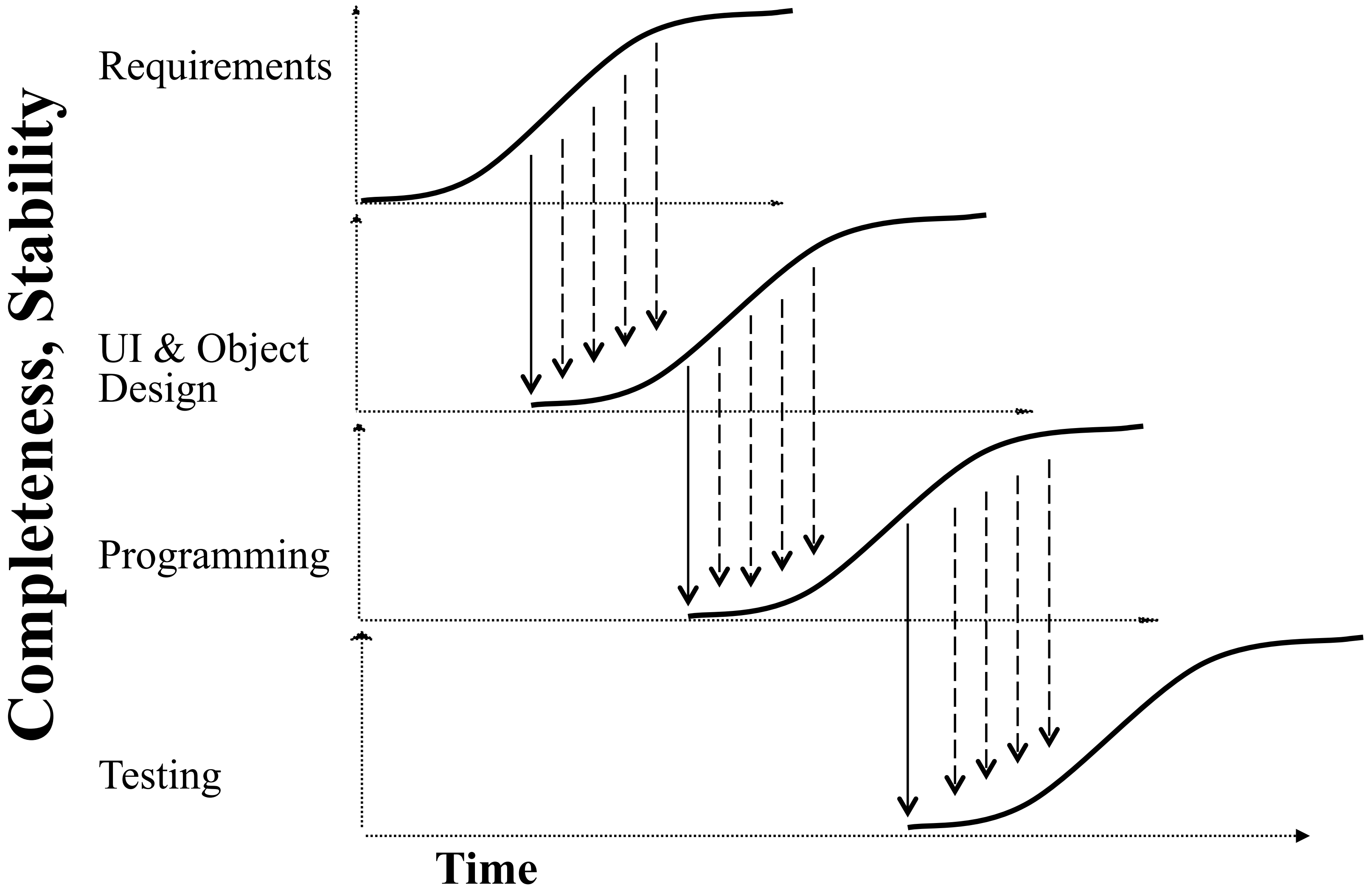

The Crystal Collection for Software Professionals
Alistair Cockburn, Collection Editor
Surviving Object-Oriented Projects
Foreword by
Rebecca Wirfs-Brock
Alistair Cockburn

Concurrent Development

Serial Development takes most time, least money (maybe)

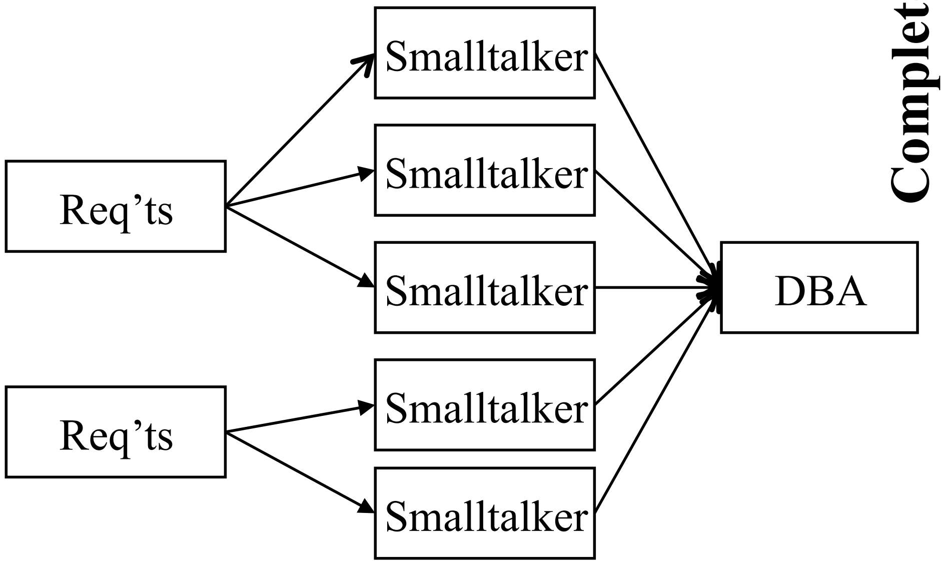


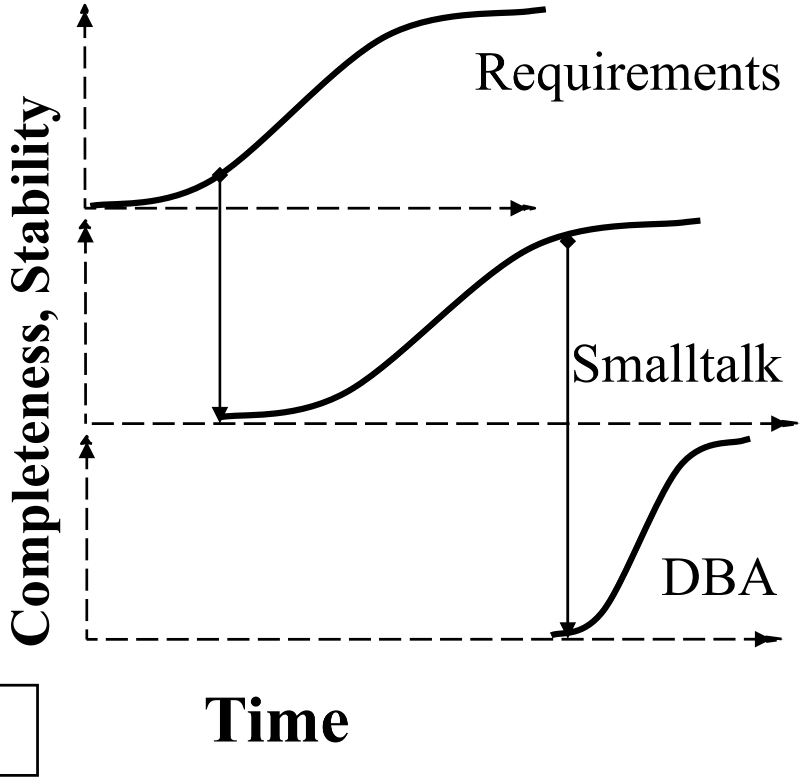
Concurrent Development takes less time more money (maybe)



Correct amount of Stability depends on the Bottlenecks !

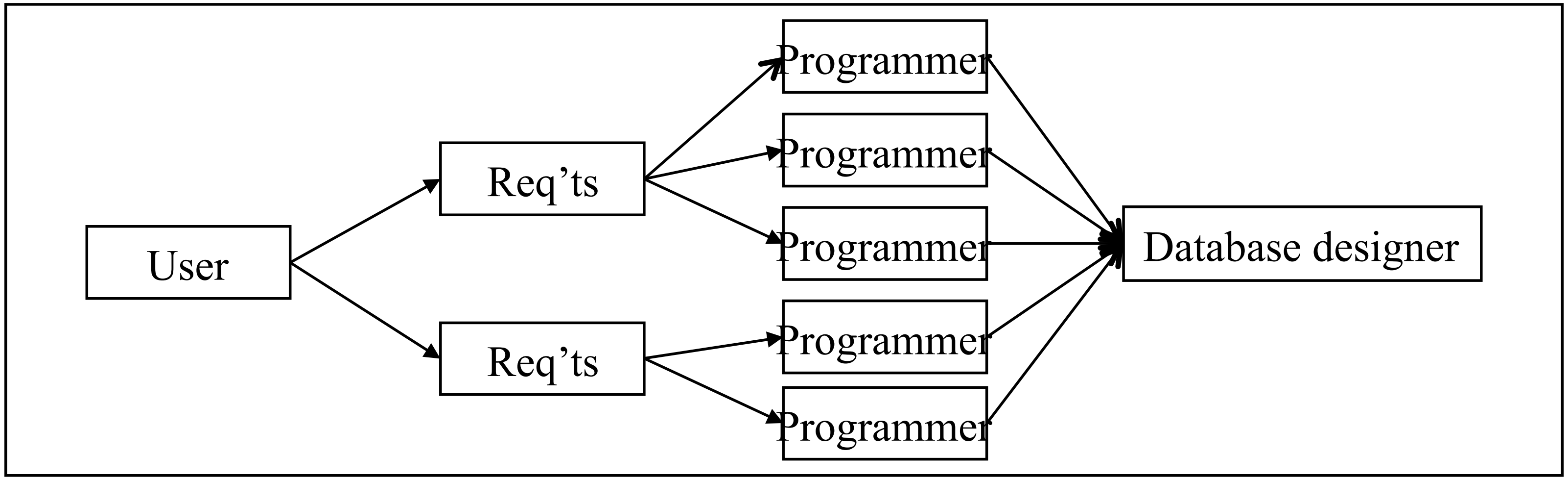
You can *spend* excess capacity in rework to speed the game!



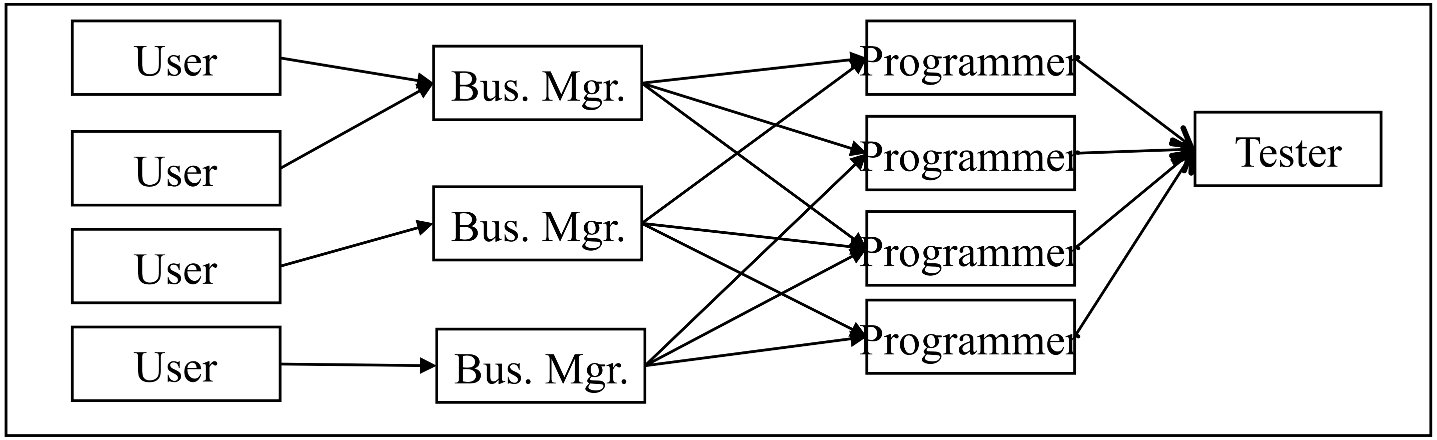


Different project have different bottlenecks, and need different strategies

Project 1: Simplify life for the **DB designer**, make programmers do extra



Project 2: Simplify life for the **programmers / tester**, make Bus. Mgrs. do extra



Things you may take away from today:

- **The Cooperative Game vocabulary**
- **Manage the *incompleteness* of communication**
- **Take advantage of face-to-face communication.**
- **Pause and reflect on your Process with a *Reflection workshop***
- **Distance hurts. (So DON'T DO IT !)**
- **Use *Incremental delivery* to get Product feedback**
- **Use of *information radiators***
- **Relevance of *amicability* and *convection currents***
- **Three levels of skill (*shu, ha, ri)***
- **Why methodologies need *tuning* to fit their ecosystem**
- **How to tune yours to your project and your people**
- ***Timeboxing / increments* as core technique**
- ***Burn-down* charts for visibility, *iceberg list* for scheduling**
- ***Concurrent development* saves time**
- **Optimize the rules to fit project-specific bottenecks**
- **All agile methodologies use *empiriical process*, *cooperative game* concepts**
- **How to mix in Scrum, consider other named agile methodologies**
- **How to look for agile methodologies in your environment**

Agile Software Development, Cooperative Game
Schedule of the day

I. Programming / Cooperative Games

II. People / Communication / Cooperation

III. Self-Evolving Methodologies

IV. Agile Techniques

V. Named Agile Methodologies

Some agile methodologies are named, Many never get named.

Scrum

XP

FDD

DSDM

Crystal

Grizzly

....and an indefinite number of others

Scrum: Software needs an "empirical process", therefore *check on it regularly !*

Ken Schwaber, Jeff Sutherland, Michael Beedle

- www.controlchaos.com
- *Agile Software Development with Scrum* by Ken Schwaber and Mike Beedle

Defined process: you can leave it to a robot.
Empirical process: A *human* has to look at it.

Software development is NOT a "defined" process

- Expect it to behave unpredictably
- Therefore:

  *** Check on it daily and monthly ***

Scrum: timebox monthly with *iceberg list*, use *burn-down chart* monthly, *daily stand-up*

Each month, *iceberg list* for month (“sprint”)
lock the list for the month

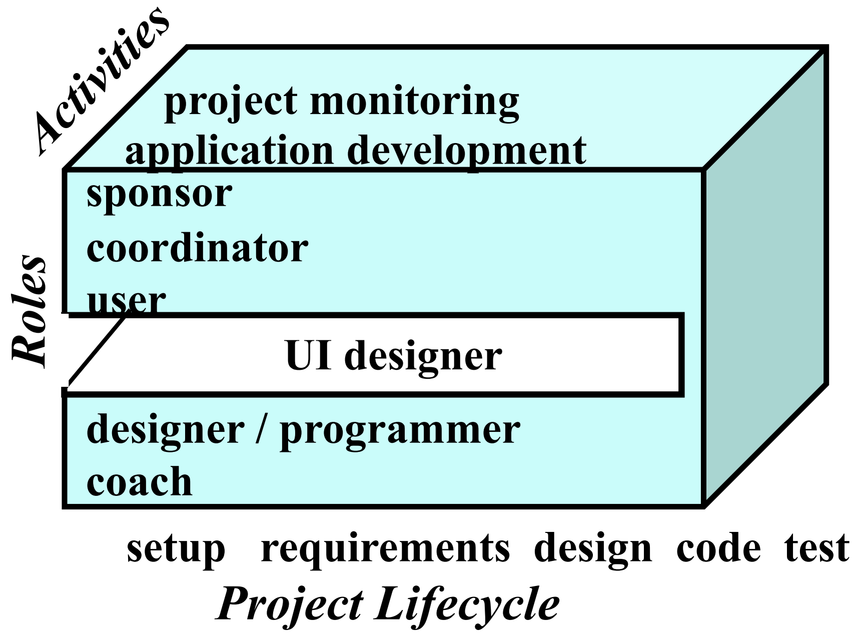
Daily stand-up meeting <15 minutes long
“What I did yesterday,
“What I plan to do today,
“What’s holding me up”

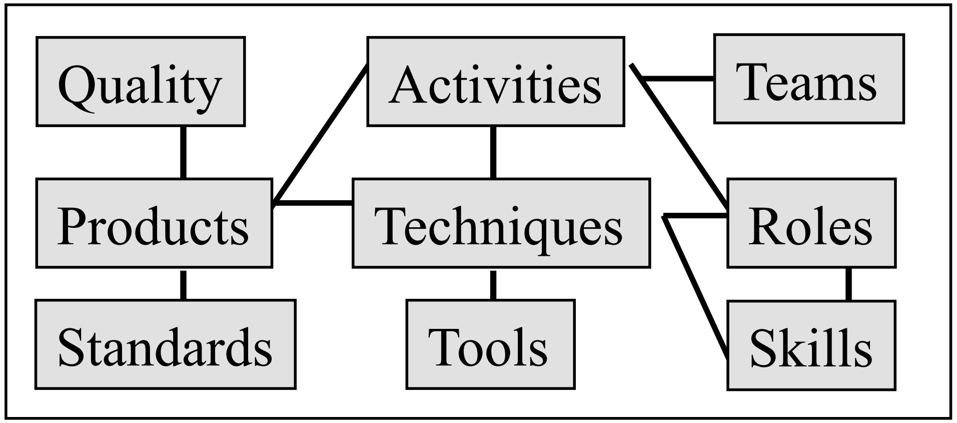
“Scrum master” removes obstacles to progress.

Demo or ship results each month.

Scrum is a perfect “mix-in” for any project

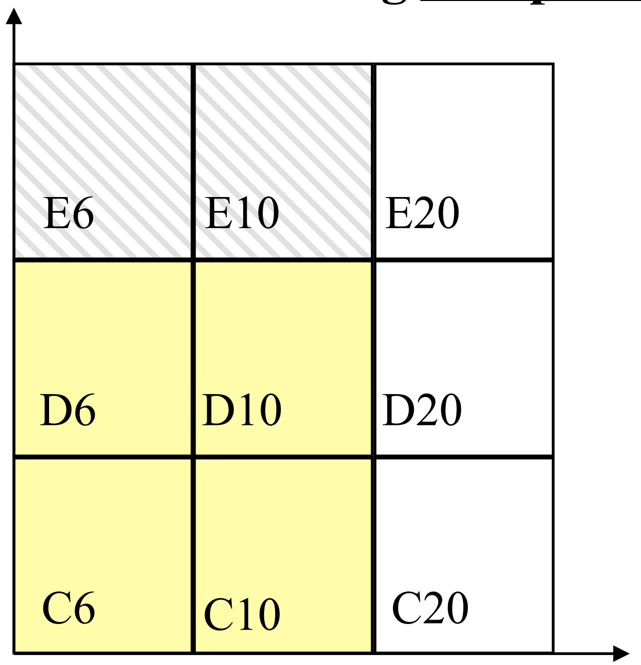
XP: A high-discipline, programmer-centric minimalist methodology





Priorities

Productivity, maintainability.
Rely on tools, communication.
Lighten methodology by increasing discipline.



E6	E10	E20
D6	D10	D20
C6	C10	C20

Extreme Programming characterized

Quality
System comparison test
Functional test
Full unit tests
Coding standards
Code cost model
"Once and only once"

Products
Release Plan
Story Cards
Task list & estimates
Running code
Migration programs
Tests
Reports

Standards
Coding style
Episodal development
3-week deliveries
Test case style
Always perfect unit test
Programming in Pairs

Activities
Setting the metaphor
Planning
Daily stand-upmeeting
DesigningProgramming
Testing
Postpartum

Techniques
Metaphor building
Planning Game
Teamwork motivation
Test-case-first development
Refactoring

Tools
Smalltalk / Java / ...
Envy/Developer
Refactoring browser
Test-case framework
Performance tuning

Values:
Communication
Keeping it simple
Testing
Courage

Teams
Programming teams
User team
Production team

Roles
Sponsor
User
Coordinator
Designer/Programmer
Production support
Coach

Skills
Programming
Refactoring
Testing

XP expects discipline in the team, distributes responsibilities across roles

3-week iterations, variable delivery schedule

Pair programming (osmotic communication)

Planning game (iceberg chart)

Automated unit tests, frequent integration

Must test, must refactor, must pair-program

"Customer" on site

- "Customer" role is overloaded

DSDM: Industrialized "rapid application development"

Developed in the UK in the mid-1990s from RAD

Managed by the DSDM Consortium

References:

- www.dsdm.org
- *DSDM: A Framework for Business-Centered Development* Jennifer Stapleton, Addison-Wesley 2003.

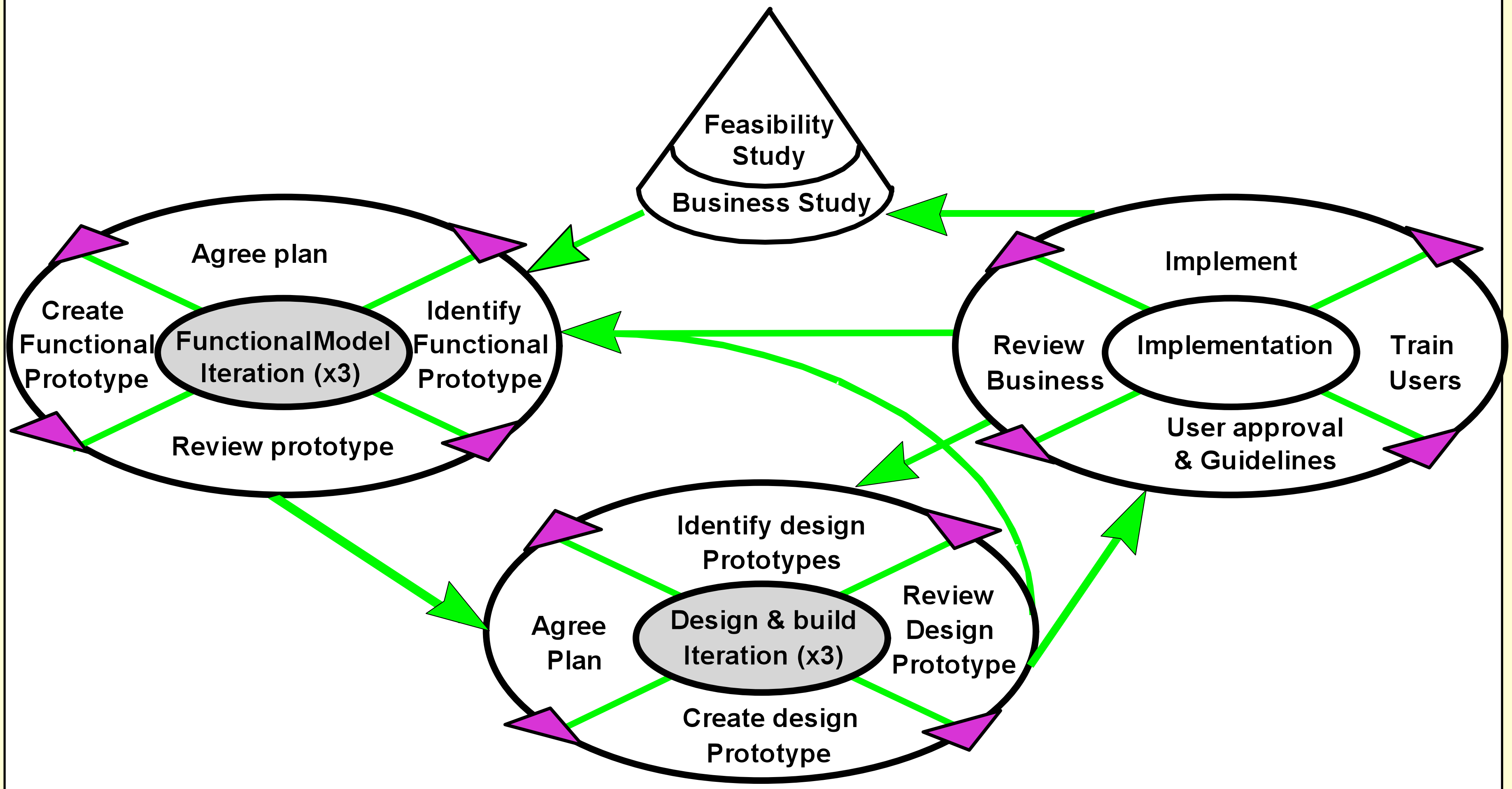
Core practices:

- Business study, prototyping, project management

DSDM is built around 9 core principles

1. Active user involvement is imperative.
2. DSDM teams must be empowered to make decisions.
3. The focus is on frequent delivery of products.
4. Fitness for business purpose is the essential criterion for acceptance of deliverables.
5. Iterative and incremental development is necessary to converge on an accurate business solution.
6. All changes during development are reversible.
7. Requirements are baselined at a high level.
8. Testing is integrated throughout the lifecycle.
9. A collaborative and co-operative approach between all stakeholders is essential.

DSDM Model


Feasibility Study
Business Study
Agree plan
Create Functional Prototype
FunctionalModel Iteration (x3)
Identify Functional Prototype
Review prototype
Implement
Review Business
Implementation
Train Users
User approval & Guidelines
Identify design Prototypes
Agree Plan
Design & build Iteration (x3)
Review Design Prototype
Create design Prototype

[1] Slides on DSDM provided by Dane R. Falkner - DSDM North American Chairman.

Feature-Driven Development is based on Chief-Programmer, modeling, feature lists

Developed by Jeff De Luca

www.FeatureDrivenDevelopment.com

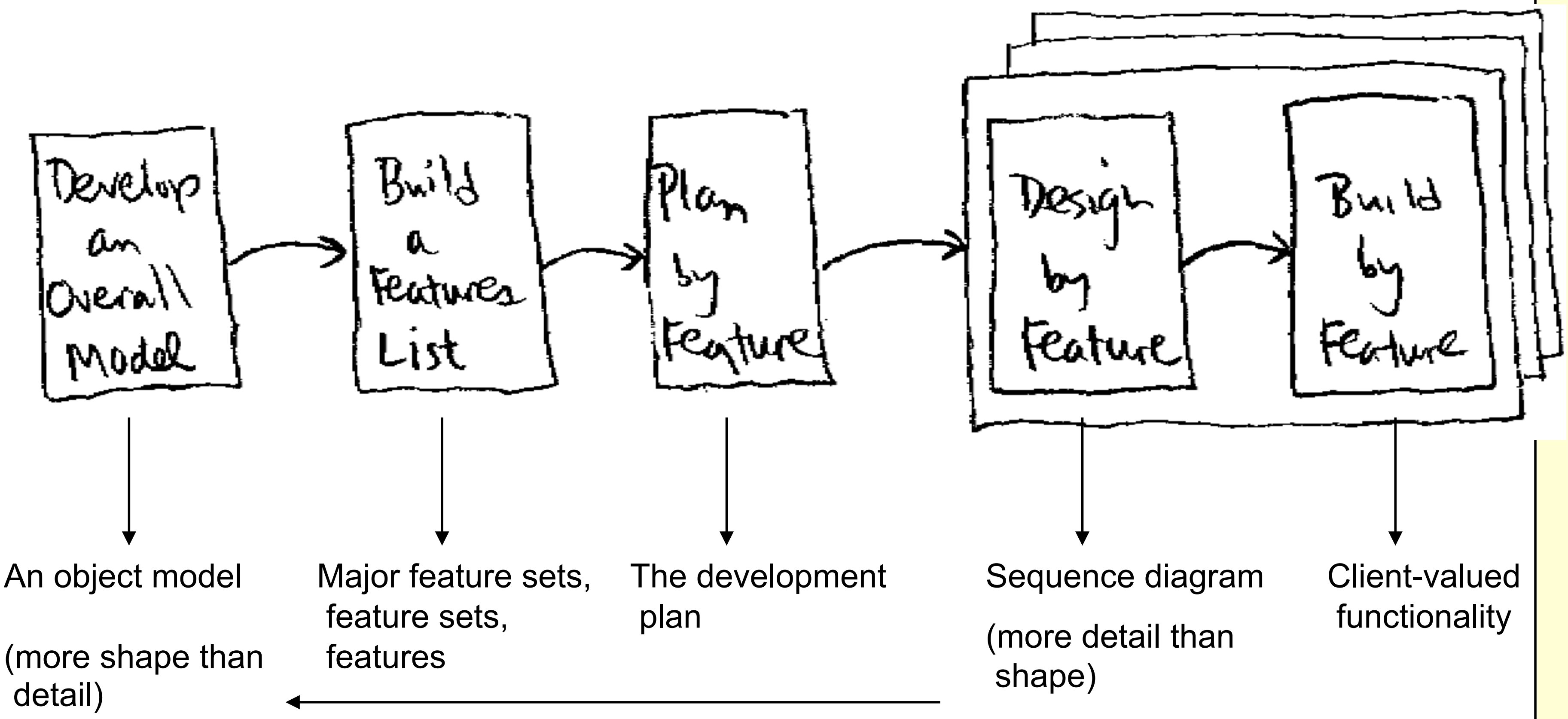
Java Modeling In Color With UML (chapter 6) by Peter Coad, Eric Lefebvre, Jeff De Luca, 1999.

Minimalist 5-step process

Key elements: (Contrast with XP !)

- Tight "process" with entry/exit criteria
- Fix time, scope and cost early
- Chief Programmer with class owners
- A central model, features are matched to it
- Review domain, Design feature, Inspect feature

FDD and Its Five Parts



An object model

(more shape than detail)

Major feature sets, feature sets, features

The development plan

Sequence diagram

(more detail than shape)

Client-valued functionality

"Adaptive" Software Development is an agile mindset for running projects

Developed by Jim Highsmith

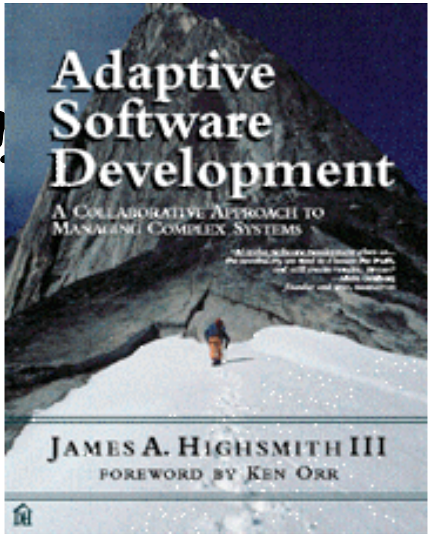
Grew out of RAD approach in the mid 1990s

Practices for scaling to larger projects

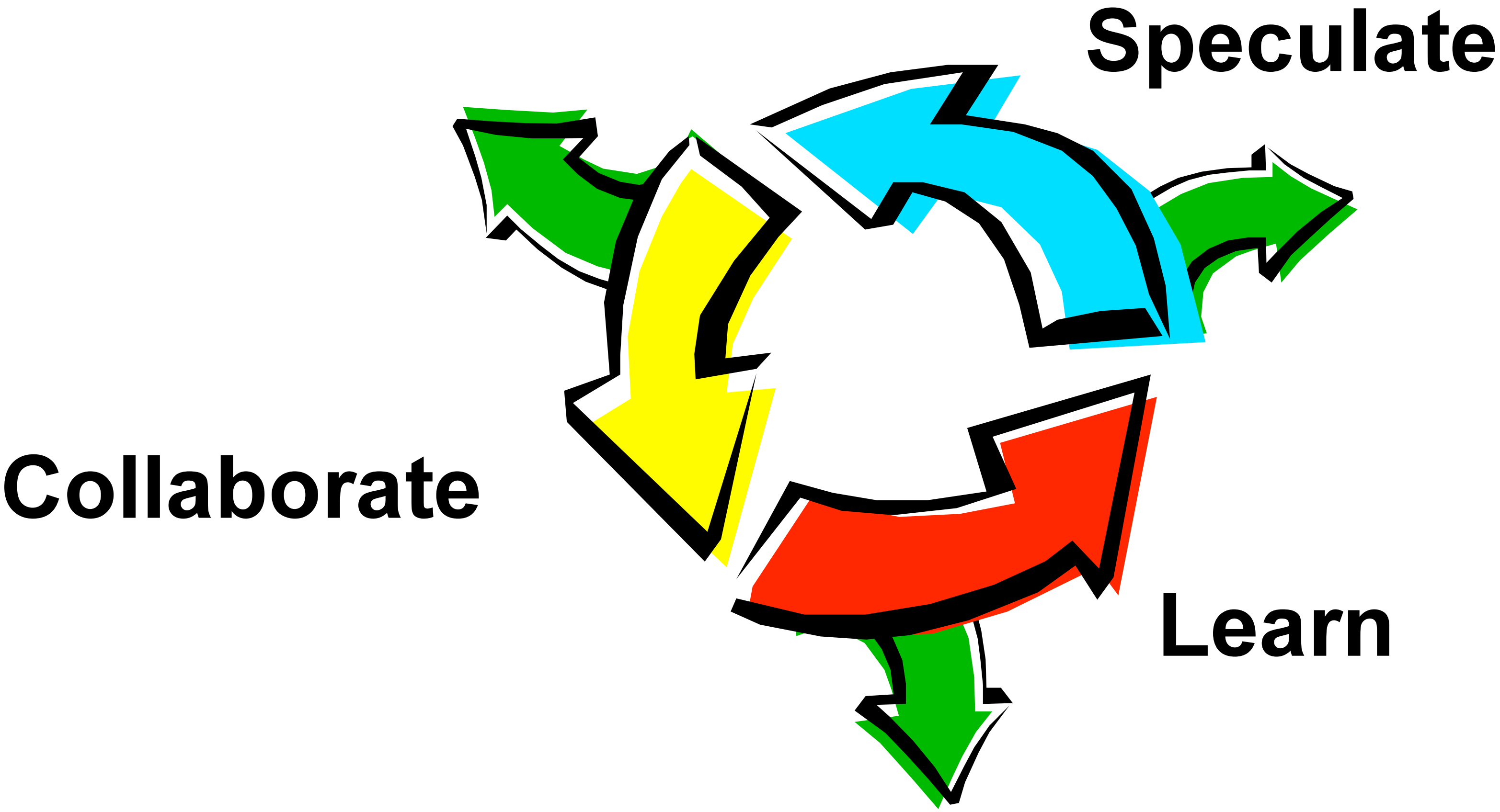
Introduces an "Agile" management style called Leadership-Collaboration

References:

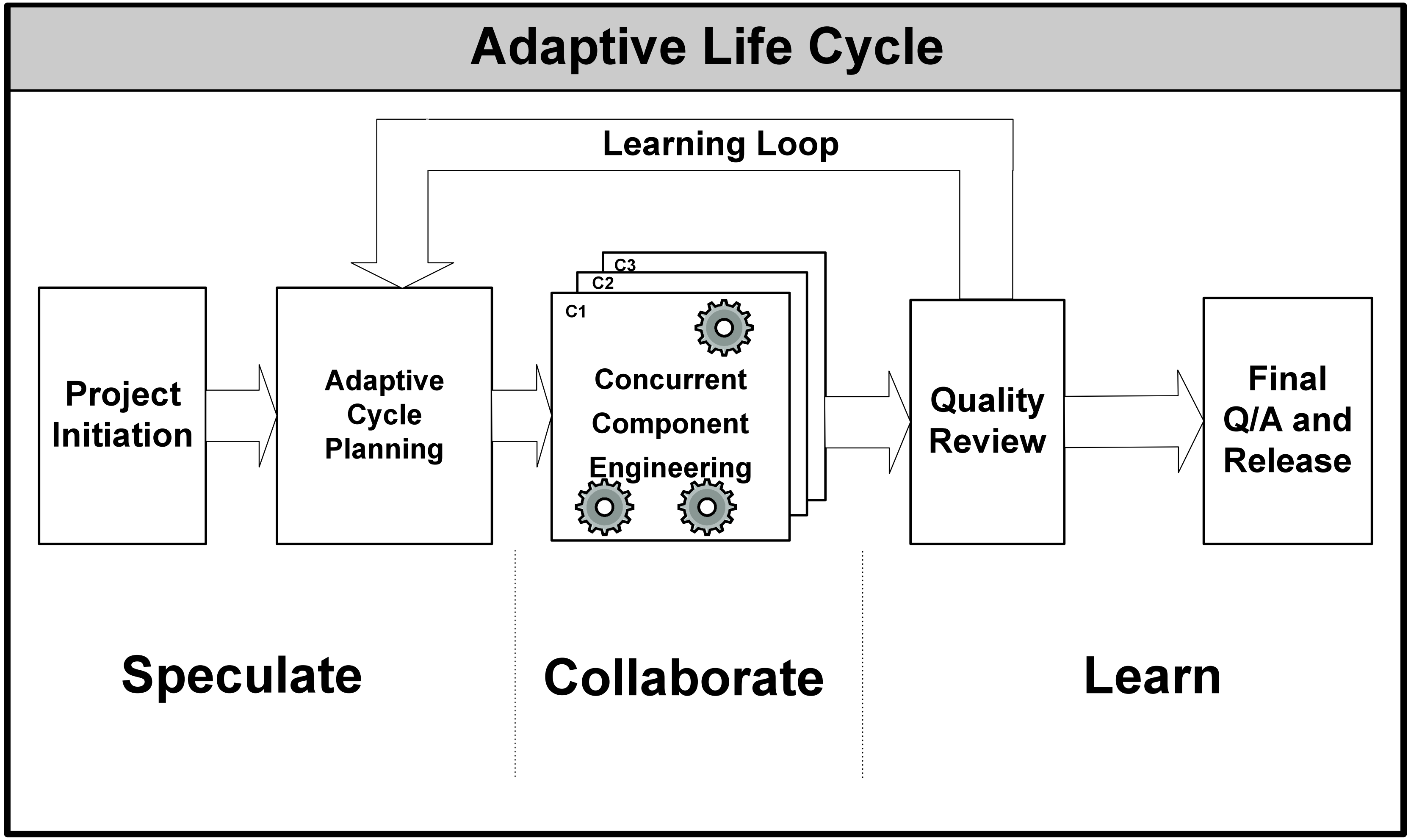
- *Adaptive Software Development: A Collaborative Approach to Managin Complex Systems*, James Highsmith, Dorset House 2000.
- www.adaptivesd.com
- www.crystalmethodologies.org


Adaptive Software Development
A Collaborative Approach to Managing Complex Systems
James A. Highsmith III
Foreword by Ken Orr

The Adaptive Life Cycle



The Adaptive Life Cycle



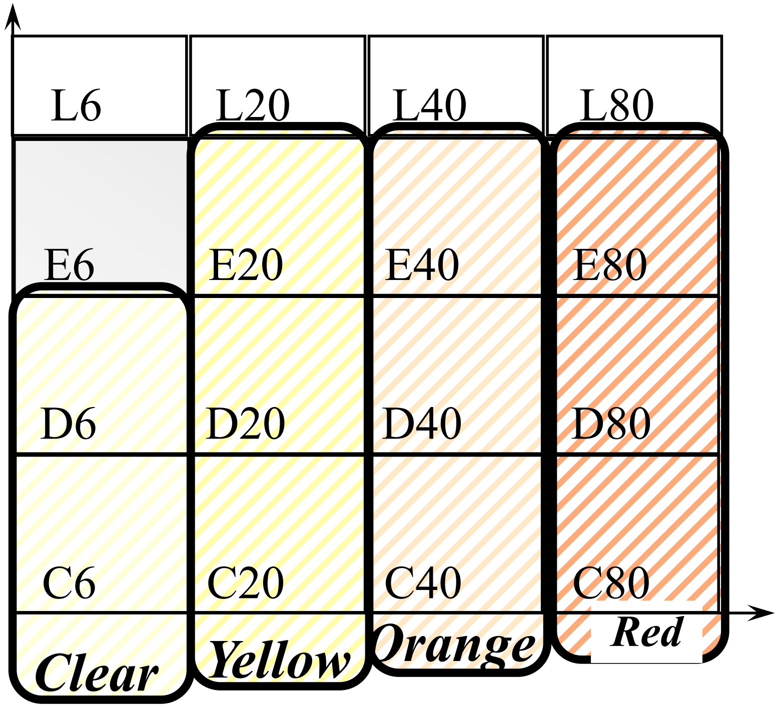
Crystal is a family of self-adapting, *tolerant* agile methodologies

Every project is different needs its own methodology

Crystal is indexed by color for # people on the project.

Only frequent deliveries and post-delivery reflection workshops are mandatory.

Keep it light and self-adapting



L6	L20	L40	L80
E6	E20	E40	E80
D6	D20	D40	D80
C6	C20	C40	C80
Clear	*Yellow*	*Orange*	*Red*

Attend to cooperative game & concurrent development principles

Crystal comes with
Philosophy, Principles, Techniques, Samples

Philosophy: Software development is a resource-limited cooperative game of invention and communication. (There is no "formula")

Principles: Deliver early & regularly,
Maximize fast, informal communication,
Reflect and improve

Techniques: Reflection workshop,
Project planning *jam session*,
Process miniature

Samples: Crystal Clear, Orange, OrangeWeb
See *Agile Software Development*, Cockburn 2002

Crystal is based on 7 principles:

1. Interactive face-to-face is the cheapest, fastest channel to exchange information.
2. Methodology weight is costly.
3. Larger teams need heavier methodologies.
4. More critical projects need more ceremony.
5. More feedback & communications mean fewer intermediate work products.
6. Discipline, skills, understanding counter process, formality, documentation.
7. Efficiency is expendable at non-bottleneck activities (concurrent development principle).

Crystal works from a base methodology tuned by people and principles

Start of project:

- Choose an iteration (increment) duration,
- Interview people to learn key issues, hazards.
- Hold tuning workshop to build starter set of rules.
- That is your "starter" methodology.

Get feedback periodically:

- Hold "reflection workshop" periodically:
  - monthly / quarterly / mid- & post-increment;
  - one hour, half hour, half day
- Update the project with new rules and conventions

Post the results prominently for all to see!

- (Hidden results are no results)

Most “agile” methodologies are home-grown, but share essential qualities:

Frequent delivery of running, tested code

Close communication / collaboration with developers, end users and customers

High levels of tacit knowledge

Low ceremony, fewer “paper” products

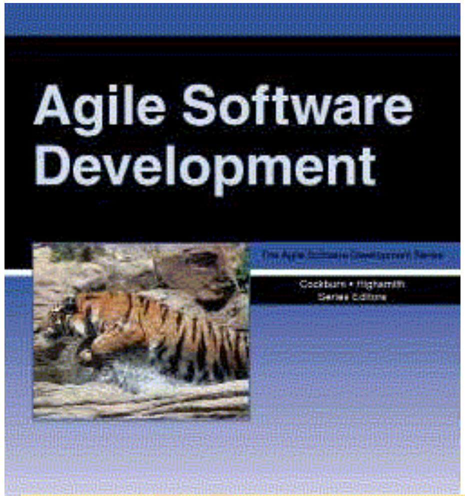
Short-term design horizon, periodic redesigns

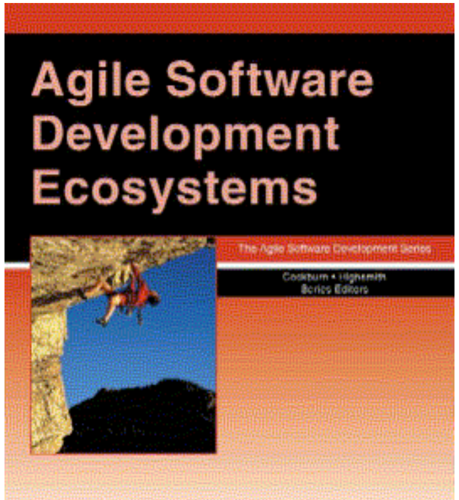
... (see the Agile Manifesto) ...

Things you may take away from today:

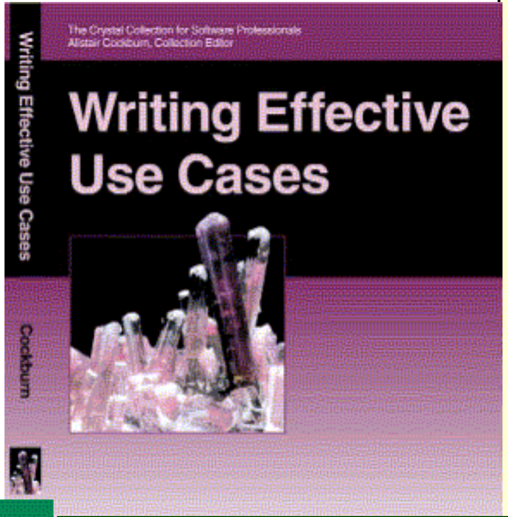
- The Cooperative Game vocabulary
- Manage the *incompleteness* of communication
- Take advantage of face-to-face communication.
- Pause and reflect on your Process with a *Reflection workshop*
- Distance hurts. (So DON'T DO IT !)
- Use *Incremental delivery* to get Product feedback
- Use of *information radiators*
- Relevance of *amicability* and *convection currents*
- Three levels of skill (*shu, ha, ri)*
- Why methodologies need *tuning* to fit their ecosystem
- How to tune yours to your project and your people
- *Timeboxing / increments* as core technique
- *Burn-down* charts for visibility, *iceberg list* for scheduling
- *Concurrent development* saves time
- Optimize the rules to fit project-specific bottenecks
- All agile methodologies use *empirical process*, *cooperative game* concepts
- How to mix in Scrum, consider other named agile methodologies
- How to look for agile methodologies in your environment

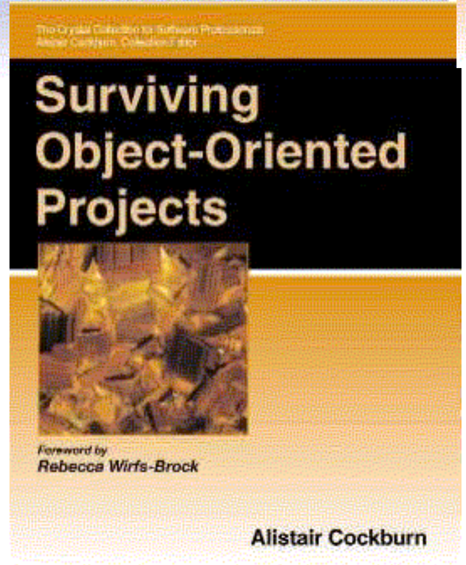
The end
Read more at http://Alistair.Cockburn.us

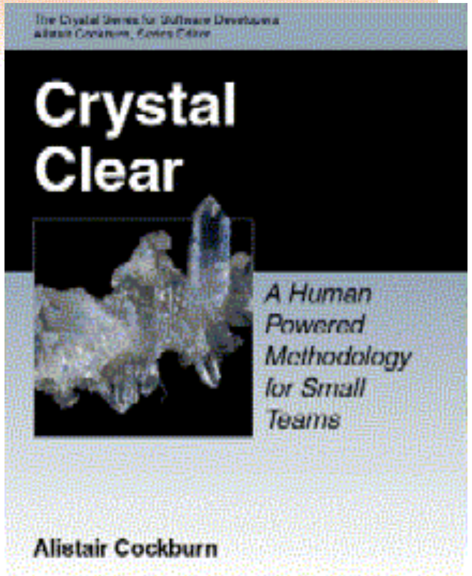

Agile Software Development
The Agile Software Development Series
Cockburn • Highsmith
Series Editors


Agile Software Development Ecosystems
The Agile Software Development Series
Cockburn • Highsmith
Series Editors

(more to come?)


The Crystal Collection for Software Professionals
Alistair Cockburn, Collection Editor
Writing Effective Use Cases
Writing Effective Use Cases
Cockburn


The Crystal Collection for Software Professionals
Alistair Cockburn, Collection Editor
Surviving Object-Oriented Projects
Foreword by
Rebecca Wirfs-Brock
Alistair Cockburn


The Crystal Series for Software Development
Alistair Cockburn, Series Editor
Crystal Clear
A Human Powered Methodology for Small Teams
Alistair Cockburn