

Use Cases

in Theory & Practice

Alistair Cockburn
Humans and Technology
Salt Lake City, UT

acockburn@aol.com
<http://Alistair.Cockburn.us>

**A use case tells a story of reaching a goal,
or a set of stories of both getting and failing.**

UC 4: ‘Place an order’

“ 1. *The clerk identifies the customer, each item and quantity.*

2. *System accepts and queues the order.*”

Extensions:

“1a. *Low credit: Clerk takes prepayment*”

“2a. *Low on stock: Customer accepts reduced...*”

Use cases address “how to make functional requirements readable, reviewable.”

- 😊 1. Use cases hold functional requirements in an easy-to-read, easy-to-track, text format.
2. A use case collects how a goal succeeds / fails; a scenario shows one specific condition; scenarios & use cases nest.
- ☹️ 3. Use cases show only the Functional req'ts.
- 😊 4. They make a framework for non-functional requirements & project details.
- ☹️ 5. Design is not done only in use case increments.

😊 **The IT industry now loves use cases, but...**
☹ **how do you write large volumes of them?**

Common agreement that use cases are useful.

Adopted by every OO methodology

Repeated commendation from developers, users

But what are they really?

How do you structure 180 of them?

what is their format? level of detail?

Jacobson invented “uses” and “extends” relations:

when do you use each, how do you type it in?

☹ Many meanings of “Use Case” are around.

😊 This model has both theory & practice.

A workshop of 14 leading OO consultants had 14 definitions of “use case”

Value: user story / requirements

Structure: none / semi-formal / formal

Content: contradictory / consistent / formal

Scenario =? Use case: same / different

Common agreement: ☹ no standard form
😊 valuable

The model: an interaction is a sequence or set of possible sequences of interactions.

Actor 1



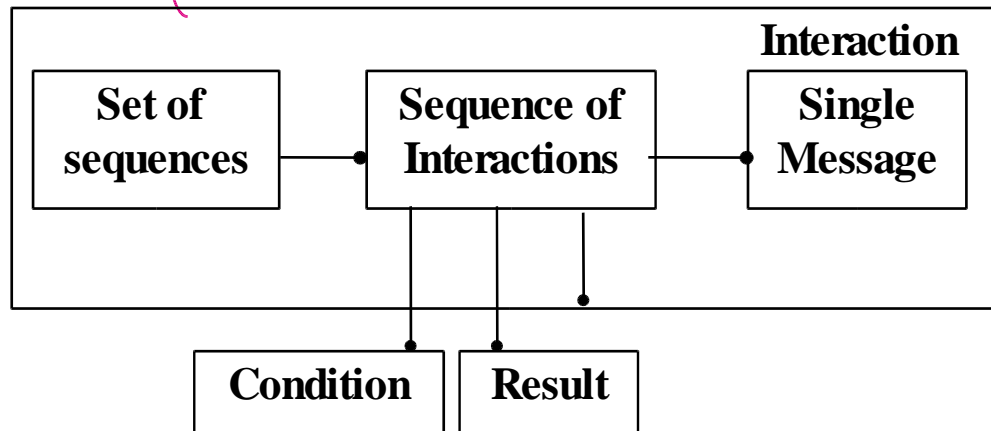
Goal



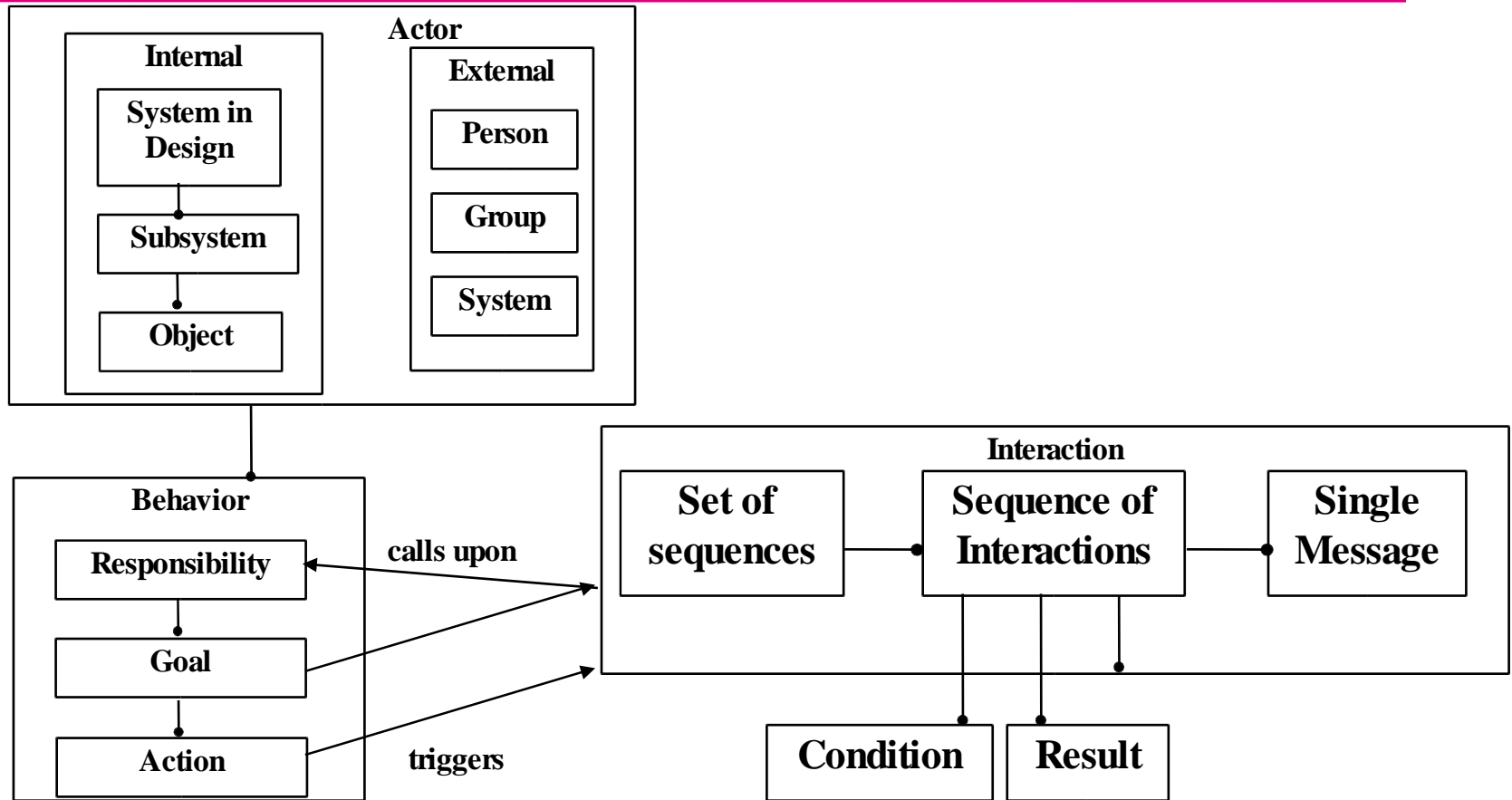
Actor 2



Responsibility



An actor's action triggers an interaction, calling upon another actor's responsibility.



The basic model of use cases is that Actors interact to achieve their goals

Primary Actor
person or system
with goal for s.u.d.

System under design
could be any system

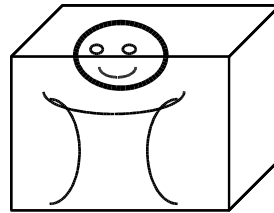
Secondary Actor
other system against
which s.u.d. has a goal



Responsibility

- Goal 1
- Goal 2
- ... action 1

(Interaction 1)



Responsibility

- Goal 1
- ...action 1

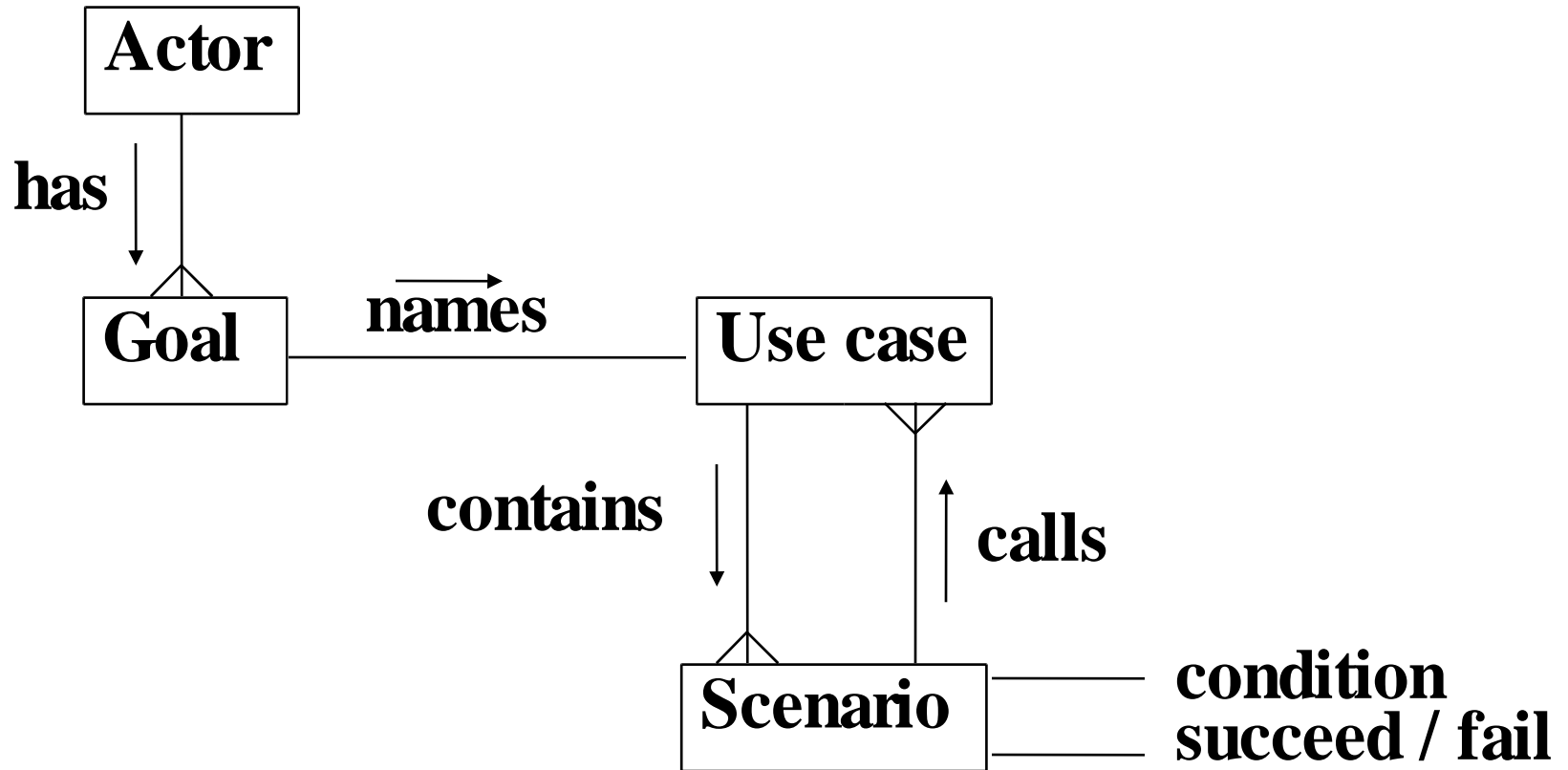
(Interaction 2)



Responsibility

- ⋮
- backup goal
for Goal 2

An actor has goals; goals name use cases; a use case has scenarios naming sub-use cases.



**Examining the Goals the system supports makes
☺ good functional requirements.**

“Place an order.”

“Get money from my bank account.”

“Get a quote.”

“Find the most attractive alternative.”

“Set up an advertising program.”

**Goals summarize system function in
understandable, verifiable terms of use.**

☺ **Users, executives and developers appreciate seeing requirements in the form of goals.**

Users:

“We can understand what these mean”

“You mean we are going to have to ...?”

Executives:

“You left out one thing here ...”

Developers:

“These are not just a pile of paragraphs!”

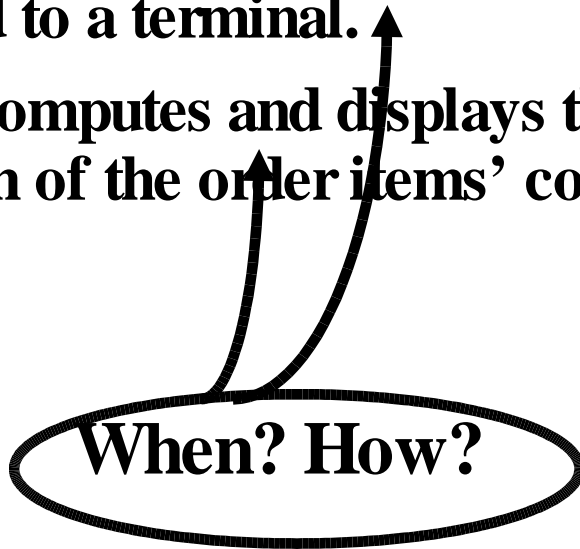
Structured narrative keeps the context visible, 😊 the value to the user clear.

Compare just paragraphs:

“The order entry system has an interface to system EBMS and to a terminal.

It computes and displays the sum of the order items’ cost.

...”



With structured narrative:

“The orderer (EBMS or an entry person) identifies the name of the customer & the items on the order.

The system displays the cost of the total order.

If the items are in stock and the client has sufficient credit, ...”

**The use case pulls goal & scenarios together,
Each scenario says how 1 condition unfolds.**

The use case name is the goal statement:

“Order product from catalog”

Scenario (1): Everything works out well ...

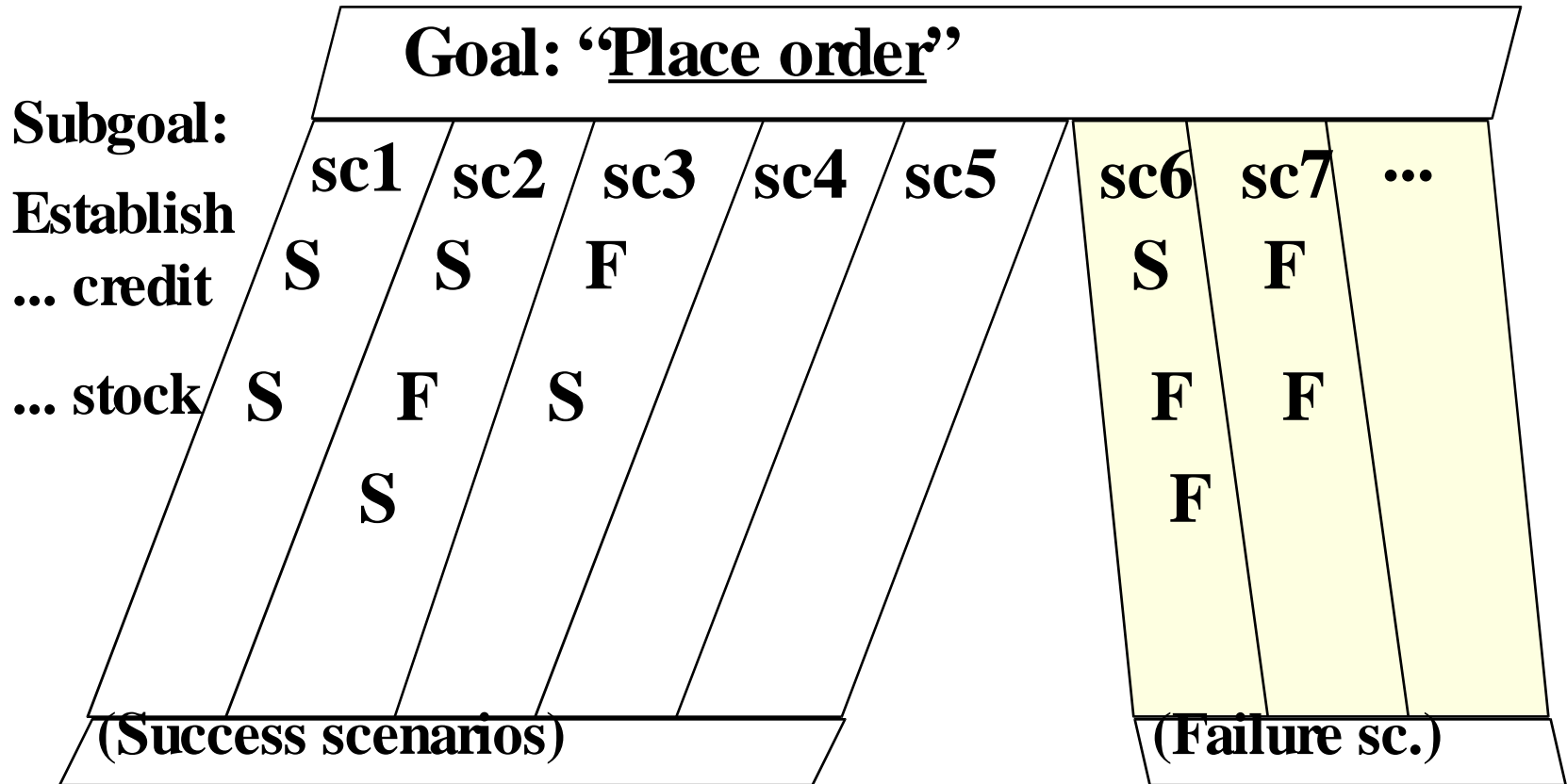
Scenario (2): Insufficient credit ...

Scenario (3): Product out of stock ...

Use case is goal statement plus the scenarios.

(Note the grammar: active verb first)

The collected scenarios are like stripes on trousers, with success and failure legs.



How to do it:

(1). Identify the actors and their goals.

What computers, subsystems and people will drive our system?

An “actor” is anything with behavior.

What does each actor need our system to do?

Each need shows up as a trigger to our system.

Result: a list of use cases, a sketch of the system.

Short, fairly complete list of usable system function.

How to do it: For each use case...

(2). Write the simple case: goal delivers.

The main success scenario, the happy day case.

Easiest to read and understand.

Everything else is a complication on this.

**Capture each actor's intent and responsibility,
from trigger to goal delivery.**

Say what information passes between them.

Number each line.

Result: readable description of system's function.

How to do it:

(3). Write failure conditions as extensions.

Usually, each step can fail.

Note the failure condition separately, after the main success scenario.

Result: list of alternate scenarios.

**How to do it: For each failure condition,
(4). Follow the failure till it ends or rejoins.**

Recoverable extensions rejoin main course.

Non-recoverable extensions fail directly.

Each scenario goes from trigger to completion.

“Extensions” are merely a writing shorthand.

Can write “if” statements.

Can write each scenario from beginning to end.

Result: Complete use case

How to do it:

(5). Note the data variations.

Some extensions are too low-level to cover “here”.

e.g. “*Reimburse customer*”

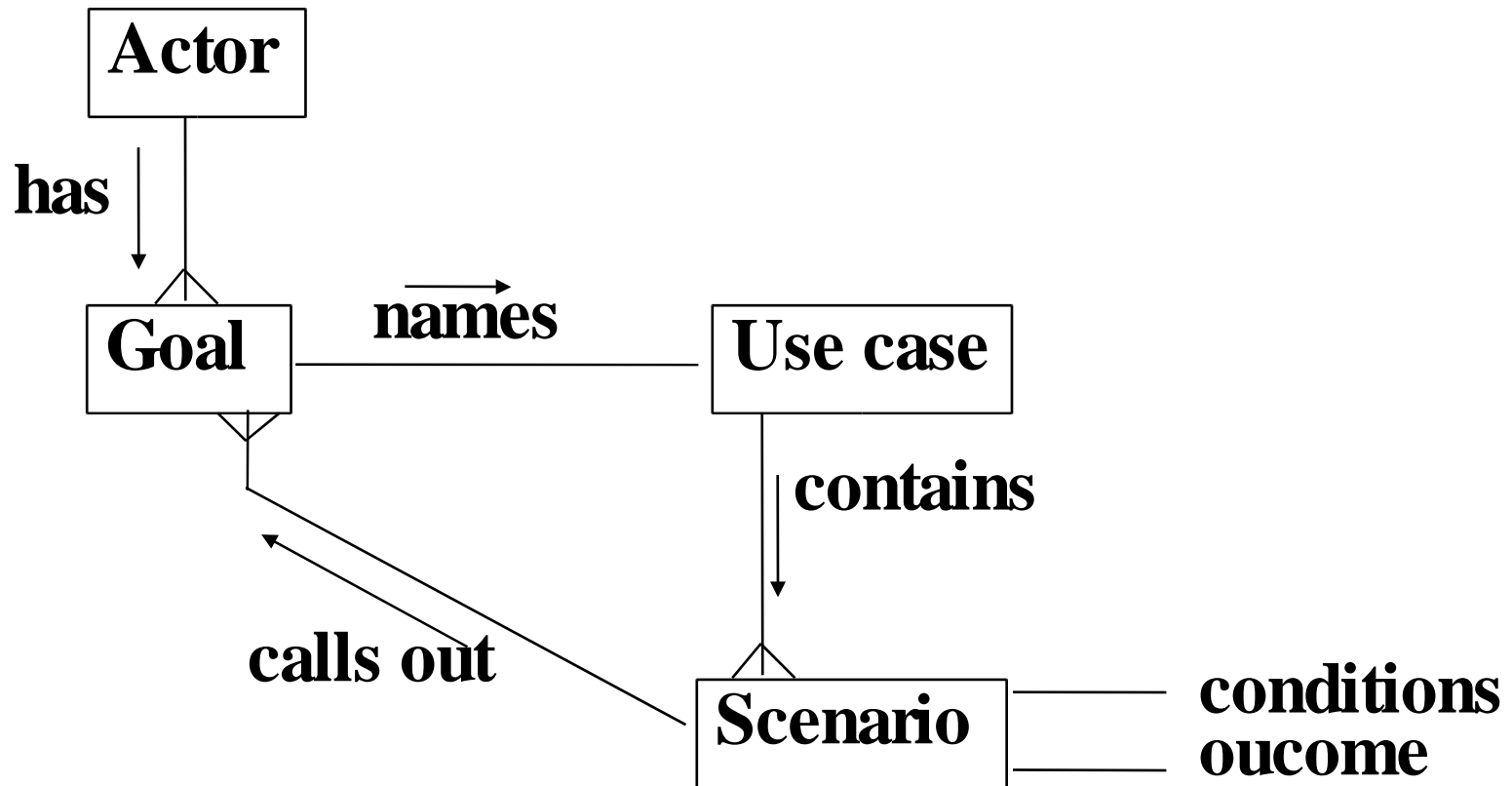
Reimburse by cash, check, EFT, or purchase credit?

Deferred variations note cases that must be handled eventually, by lower-level use cases.

Useful for tracking requirements at high level.

Result: Feed-forward information, rolled up into an easy-to-track format.

An actor has goals; goals name use cases; a use case has scenarios naming sub-use cases.



Review: Make scenarios run from trigger event to goal completion or abandonment.

UC 4: ‘Place an order’

Trigger: the request (phone call, EDI, ...).

End: order started or canceled.

Scenario	Scenario	Scenario	all
ok: Insufficient \$:	No product:		
Start order.	Refuse order and forget.	and forget.	Issue raincheck

(Use case in “decision-table” format)



The value of failure scenarios is detecting ☺ unusual situations, completeness

“What if their credit is too low?”

“What if they run over the extended credit limit?”

“What if we cannot deliver the quantity?”

“What if data is corrupt in the database?”

(These are commonly overlooked situations)

Both recoverable and non-recoverable failures are part of requirements.

Ideal situation (s1):

Good credit, items in stock -> accept order.

Recoverable situation (s2, s3):

Low credit -> valued customer? -> accept.

Low stock -> reduce quantity? -> accept.

Unrecoverable situations (s4, s5):

Not a valued customer -> decline order

Out of stock -> decline order

Write the recoverable and failure scenarios as “extensions” to the ideal one.

UC 4: “Place an order”

“ 1. Clerk identifies the customer, each item and quantity.

2. System accepts and queues the order.”

Extensions:

“1a. Low credit: Customer is ‘Preferred’...”

“1b. Low credit & not Preferred customer: ...”

“2a. Low on stock: Customer accepts reduced...”

**A scenario refers to lower-level goals:
subordinate use cases or common *functions*.**

UC 4: ‘Place an Order’

- 1. Identify customer (UC 41)**
- 2. ...**

Note the active verbs!

UC 41: ‘Identify Customer’

- 1. Operator enters name.**
- 2. System finds near matches.**

Extensions:

2a. No match found: ...

The outer use case only cares if the inner one succeeds, avoiding proliferation.

UC 4: ‘Place an Order’

1. *Identify customer*

2. ...

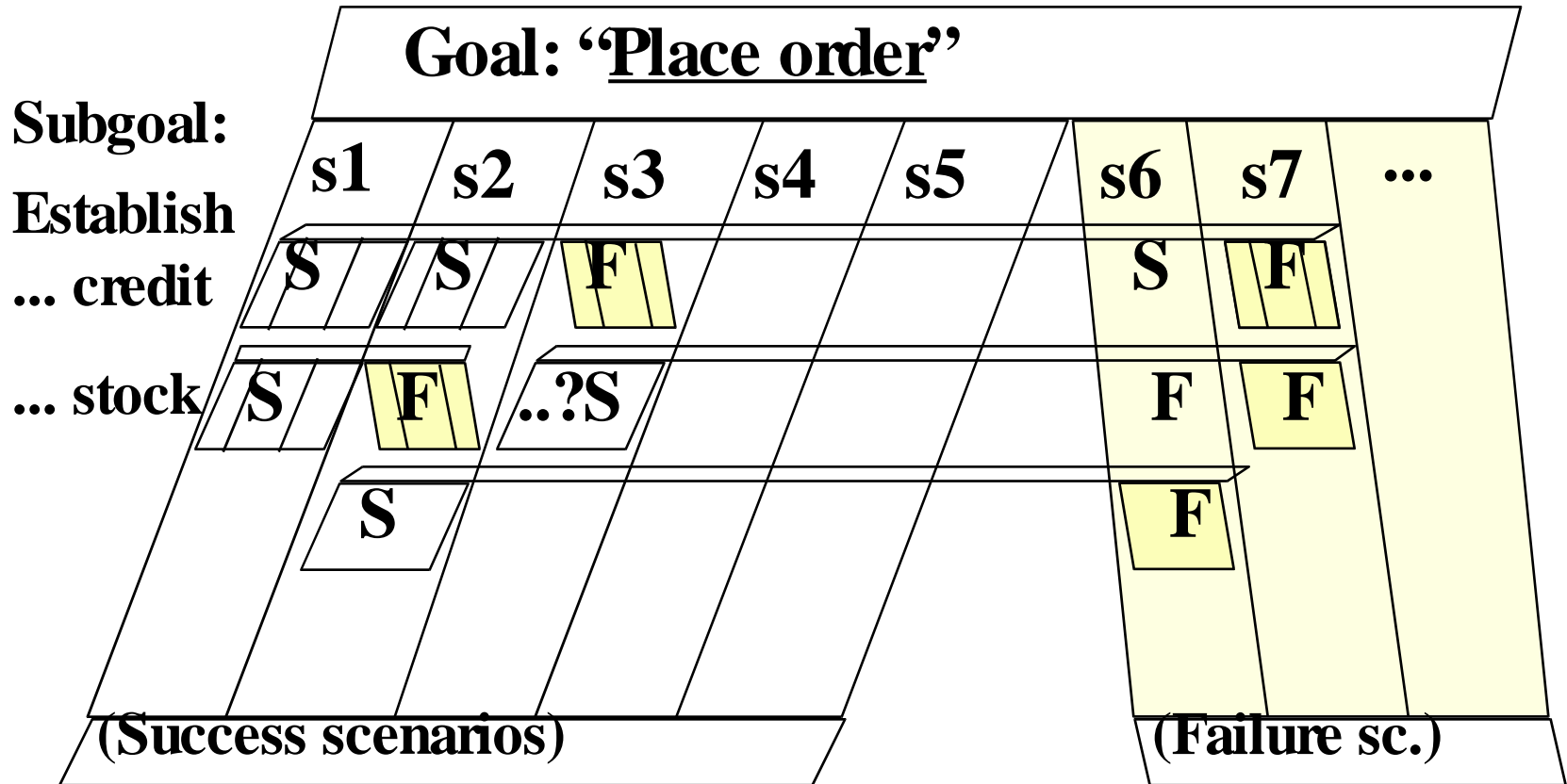
Extensions:

1a. *Customer not found:*

<- (assumes success)

<- (does not care why it failed, only if it is recoverable)

Each scenario step is a sub-goal, hiding a nested use case (striped trousers image).



**Every sentence at every level is a goal.
Use cases are one sentence style repeated.**

Goal: “Customer places an order.”

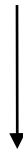


how

why



Step: “Customer prepays for the order.”



how

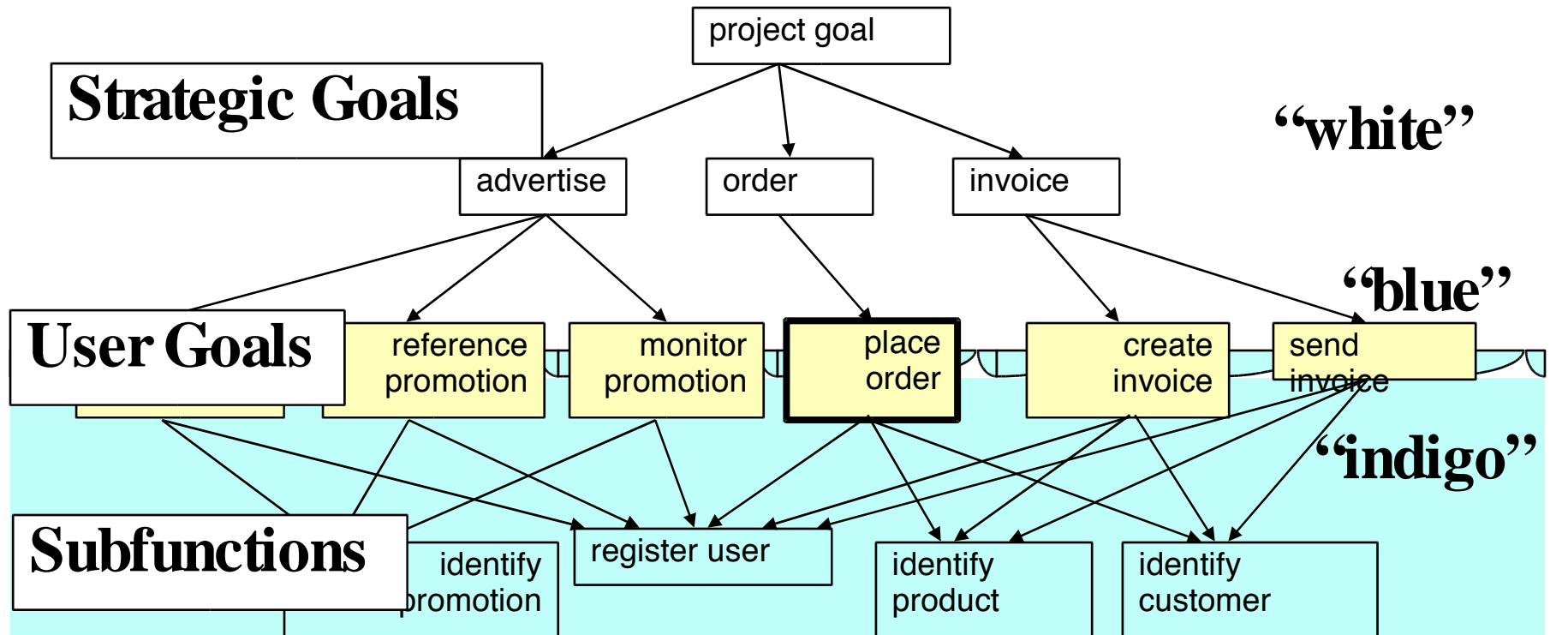
why



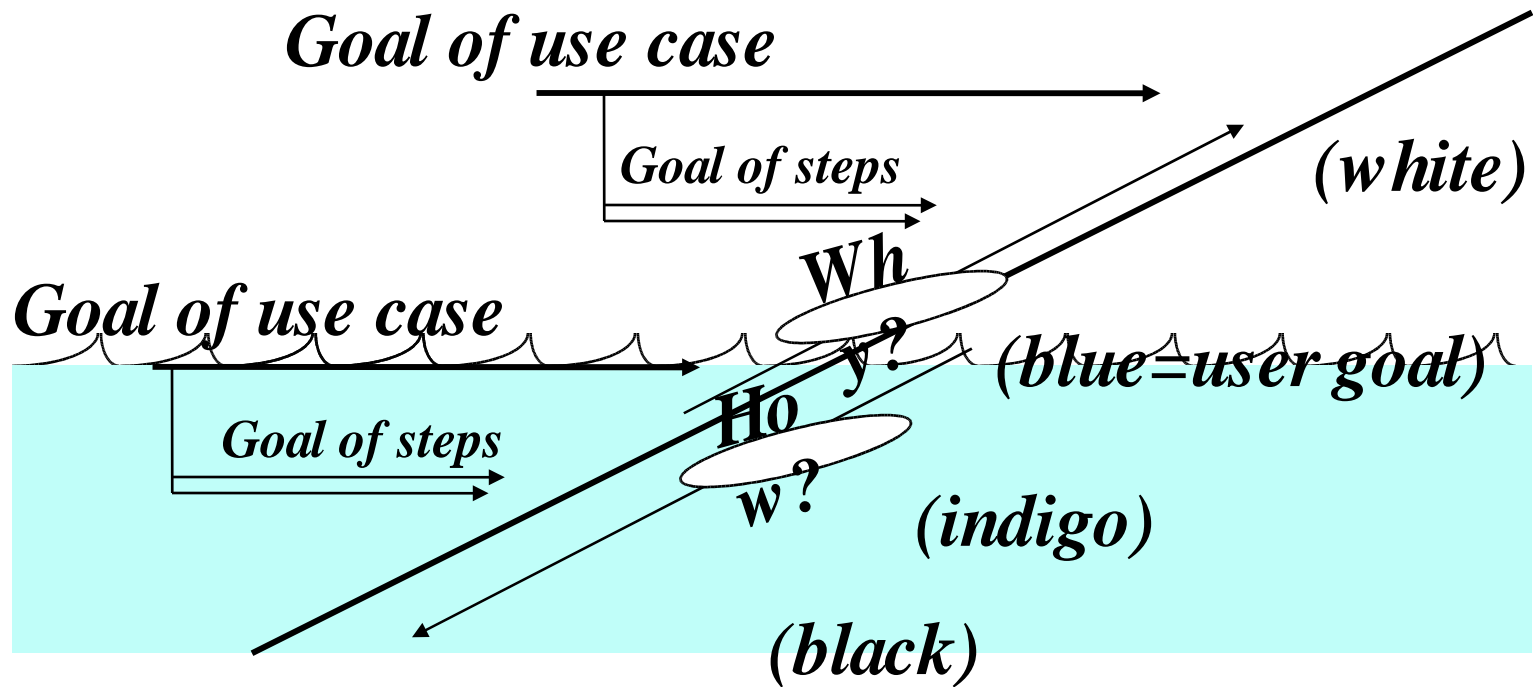
Substep: “Customer gives credit card number.”

Strategic use cases, tasks, and subfunctions link together as a graph.

Sailboat image: User goals are at sea level.



The use case goal level is higher than the steps'. They white to blue, indigo, black



Use cases nest by (level, scope, detail).

☹ Which should you write in?

Level: Why do we want this goal?

enter \$ amount, to *get* \$, to buy lunch

(“subfunction” vs. “task” vs. “strategic” goal)

Scope: Which system boundary do we mean?

The panel is part of the ATM, is part of the bank.

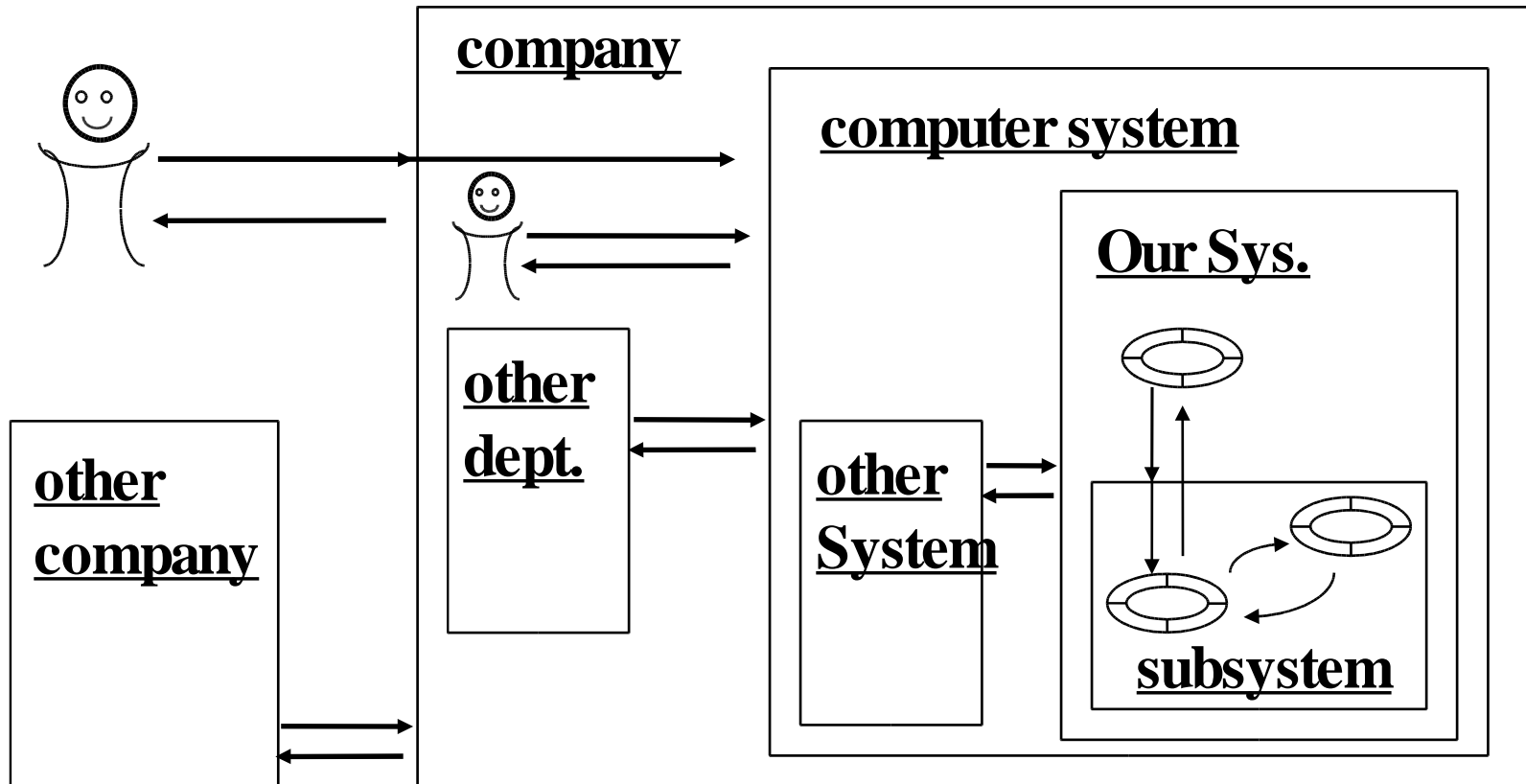
(“internal” vs. “system” vs. “organization/corporation”)

Detail: Do we describe intent, or action detail?

hit number buttons to enter \$ amount

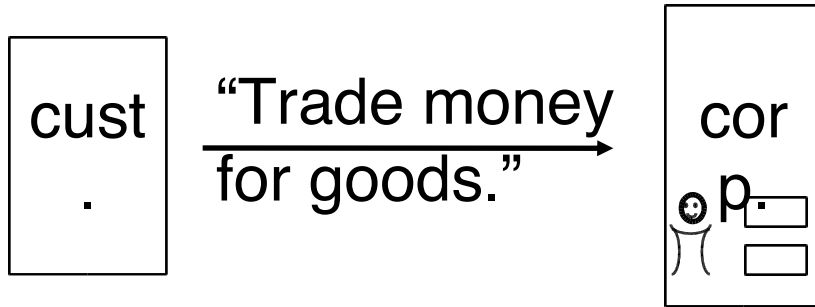
(“dialog description” vs. “semantic / intent”)

☹ **Systems are recursive in nature, from enterprise down to program modules.**

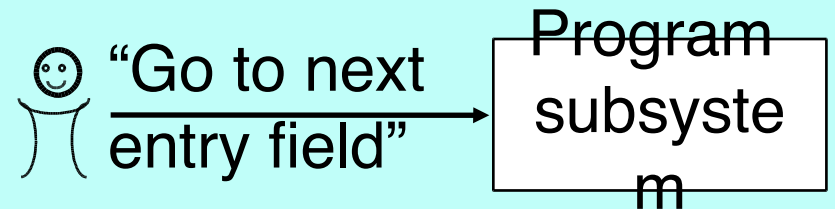


☹ Capture strategic & task goals; system & corporate scope; capture intent (semantics).

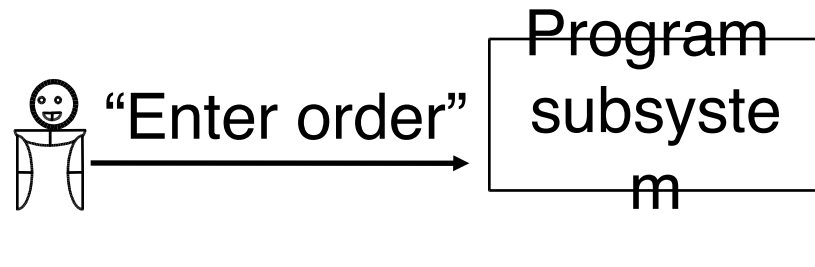
(strategic, corp., semantic)



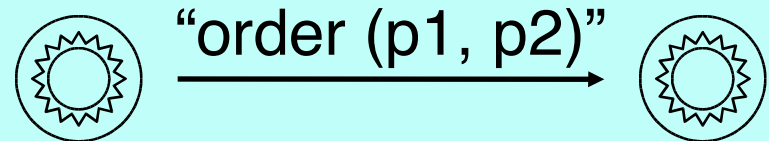
(task, system, dialog)



(task, system, semantic)



(task, internal, semantics)



☺ **Goals make a good structure on which to hang requirements & project details.**

☹ **Some requirements are not in the use cases:**

Performance, delivery time window.

Interface and business rule usage

Project planning capitalizes on goal structure:

☺ **(Useable) Releases.**

Priorities, schedule, staffing.

Growth of the somponent base.

☹ Unresolved scheduling problem: system features slice and cross use cases

Example:

1. Place an order - using standard pricing.
2. Place an order - using Preferred pricing.
3. Place an order - do not check credit limit.

With full use case / feature -> use case explosion.

Use case for many features -> scheduling difficulty.

(Still looking for a way to avoid use case duplication.
)

- ☹ Use cases do not show interface requirements.
Collect them by use case.

Use case	Form	In	Out
set up promotion	on-line	products, dates	new promotion
reference promotion	on-line	promotion #	promotion value
enter an order	on-line	customer, products, ...	new order
create an invoice	database	order number	new invoice
send an invoice	tape	invoice #	paper or EDI

☹ Use cases do not capture performance requirements. Connect them to use cases.

Use case	Frequency	Performance	...
set up promotion	10 / mo	interactive	
reference promotion	500 / day	sub-second	
enter an order	80 / day	interactive	
create an invoice	80 / dy	3 seconds	
send an invoice	1600/mo	420/hr (10 sec.)	

☹ **Use cases do not collect formulae, state, cardinality. Capture them separately.**

... in any form available (“just a tool problem”)

Examples:

- 1. Order sum = order item costs * 1.06 tax**
- 2. Promotions may not run longer than 6 months.**
- 3. Customers only become Preferred after an initial 6 month period.**
- 4. A customer has one and only one sales contact.**
- 5. An order item may use many promotions.**

**Need a list of actual clients of each use case,
☺ for use in packaging and training.**

Use case	Users
set up promotion	Marketing managers
reference promotion enter an order	Managers, order clerks, other subsystems
create an invoice	Managers, invoicing subsystem
send an invoice	Managers, invoicing subsystem

Project management: Schedule releases by clusters of use cases.

Release 1:

(#1) Set up & reference promotion

(#4) Enter order

Release 2:

(#5) Create invoice

(#3) Monitor promotion

Release 3:

(#6) Send invoice

Use cases can be managed in text, in Lotus ☺ Notes, or in “use case” OO database.

Group 1: Tried Word / Wordperfect (unsatisfactory)

Hard to share, hard to modify format.

Group 2: Used Lotus Notes (good)

Excellent for sharing, evolving format, data.

Contents slowly became inconsistent.

Group 2 converting to Intelligent Software Factory

Has places for performance, etc. (like L.Notes)

OODB with semantic model, consistency, etc.

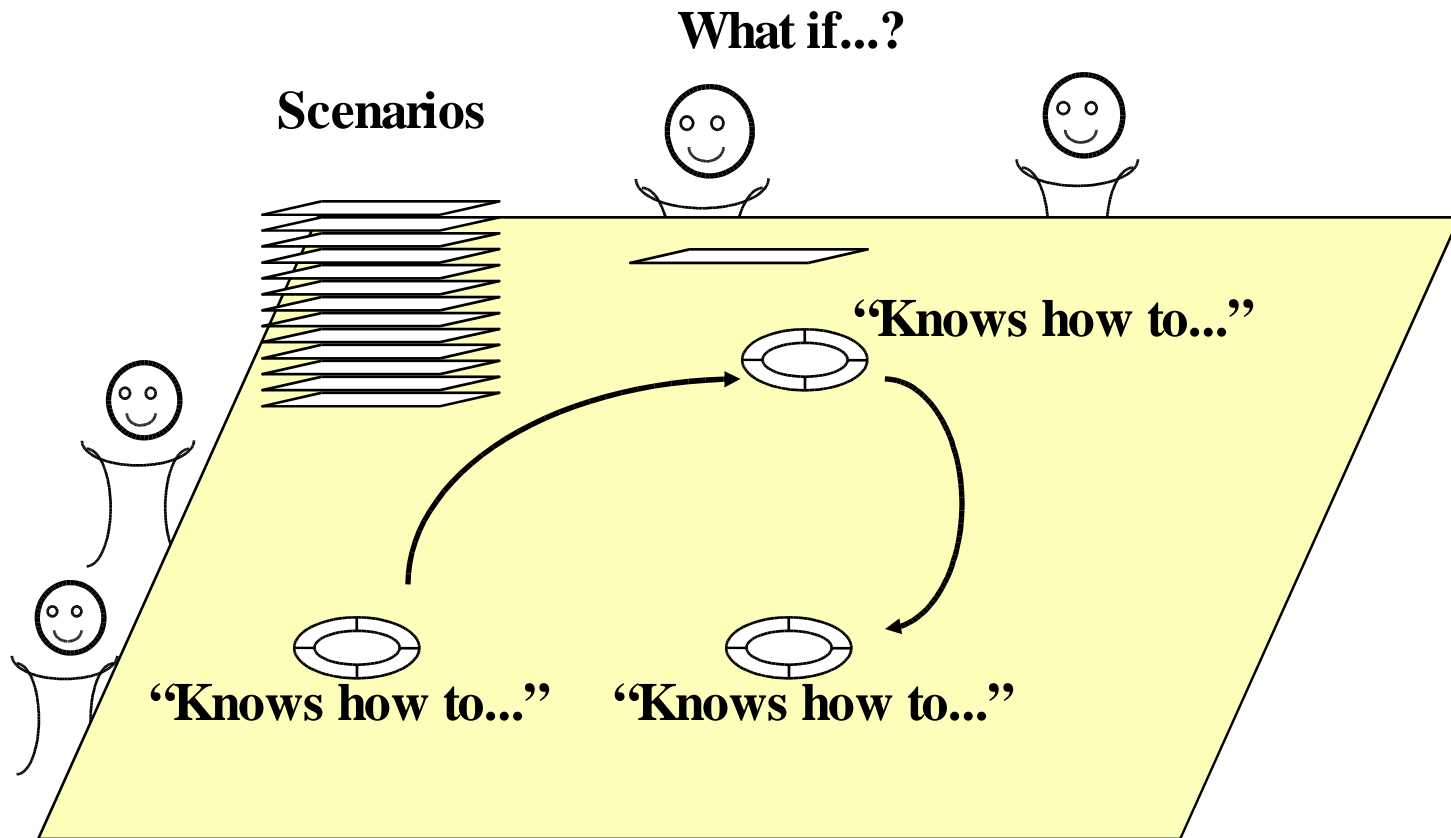
Now for Design: Scenarios provide the basis ☺ for design with responsibilities.

Responsibility-based design is based on role-playing a walkthrough of a scenario.

Multiple scenarios provide the basis for asserting that the design delivers the required function.

Use of failure scenarios make the design complete & robust.

Designers can use the scenarios directly to ☺ design the system.



Robust design requires examining use cases out of schedule sequence.

A new use case may add to existing classes.





- > Harder to schedule class design milestones.**
- > The object model is never complete, only “complete with respect to these use cases.”**

New use cases may change optimal design

- > Use all use cases or do incremental design?**

Just make sure someone is responsible for delivering the total function.

Visit *<http://Alistair.Cockburn.us> and <http://usecases.org> to read more.*

-  **1. Use cases hold functional requirements in an easy-to-read, easy-to-track, text format.**
- 2. A use case collects how 1 goal succeeds or fails; a scenario shows one specific condition; scenarios / use cases nest inside each other.**
-  **3. Use cases show only the Functional req'ts.**
-  **4. They make a framework for non-functional requirements & project details.**
-  **5. Design is not done only in use case increments.**

The system protects the interests of all the stakeholders.

