UNIVERSITY of HOUSTON

DEPARTMENT OF COMPUTER SCIENCE

COSC 4351 Fall 2023

Software Engineering

M & W 4 to 5:30 PM

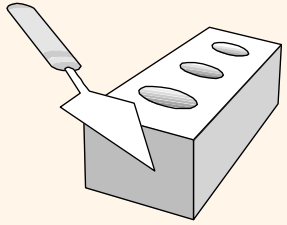
Prof. Victoria Hilford

PLEASE TURN your webcam ON

NO CHATTING during LECTURE

COSC 4351

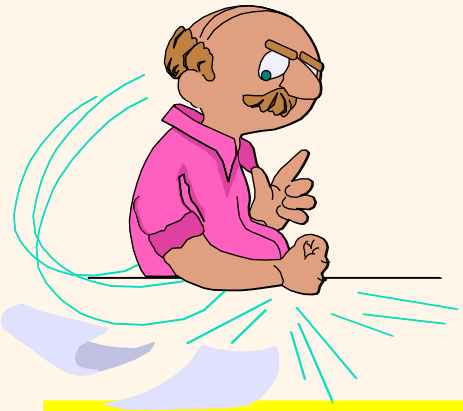
4 to 5:30



PLEASE

LOG IN

CANVAS



Youyi [A-L]

Kevin [M-Z]

Please close all other windows.

09.06.2023 (W 4 to 5:30) (5)		Lecture 3: Software Development Process			
09.11.2023 (M 4 to 5:30) (6)		EXAM 1 REVIEW (CANVAS) (ZyBook)	Download ZyBook: Sections 1-5		
09.13.2023 (W 4 to 5:30) Optional (7)					Q & A Set 1 topics.
09.18.2023 (M 4 to 5:30) (8)					EXAM 1 (CANVAS) (ZyBook)

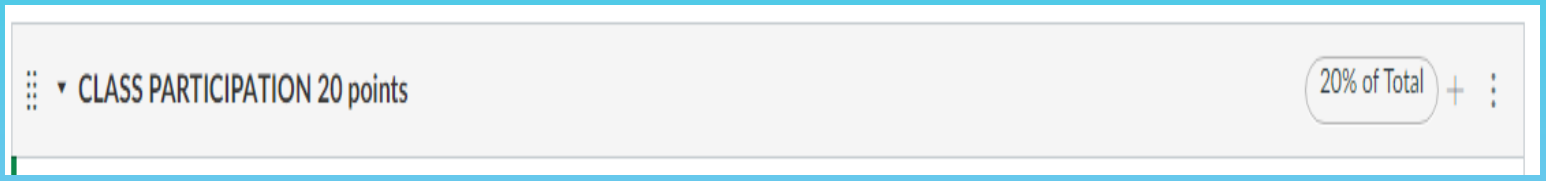
Class 5

Software Engineering

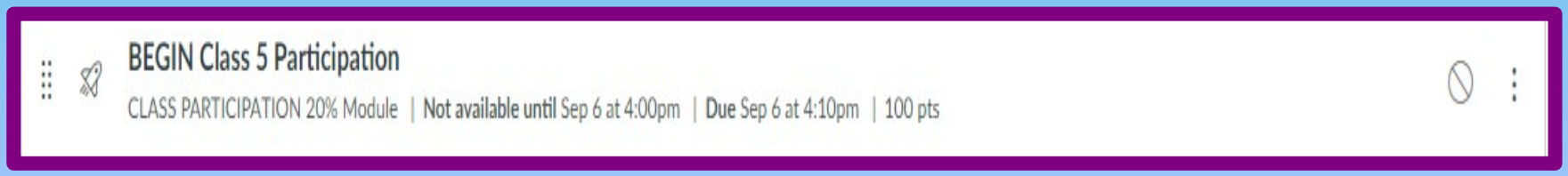
09.06.2023 (W 4 to 5:30) (5)		Lecture 3: Software Development Process			

From 4:00 to 4:10 – 10 minutes.

09.06.2023 (W 4 to 5:30) (5)		Lecture 3: Software Development Process			

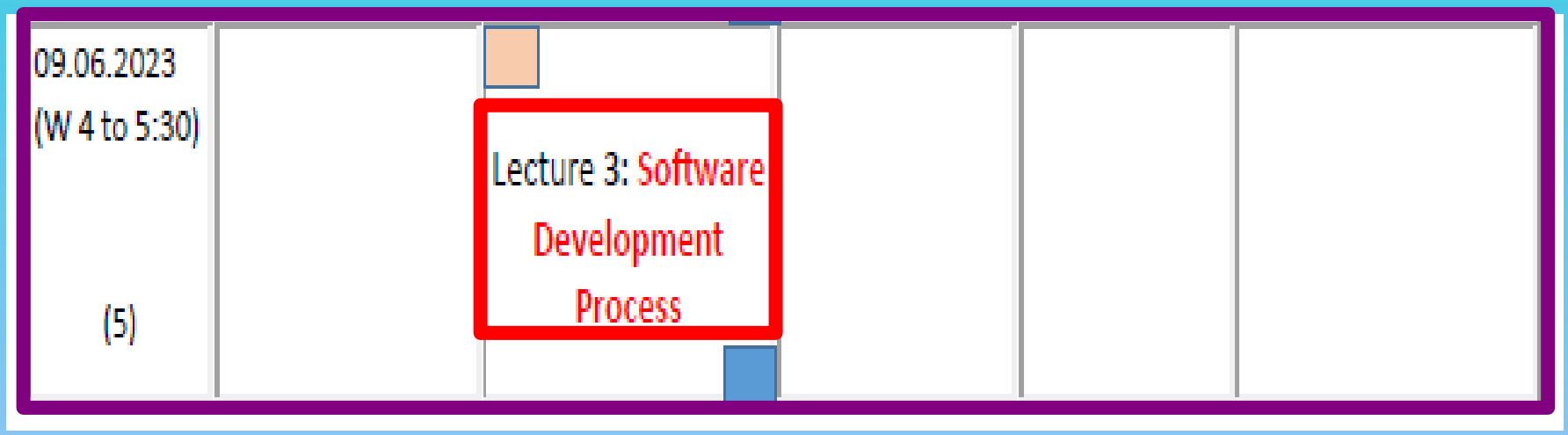


PASSWORD: I AM IN TEAMS



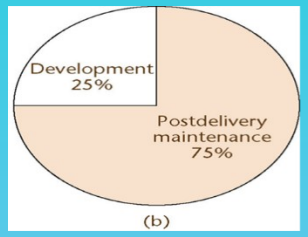
VH, publish.

From 4:10 to 5:00 – 50 minutes.

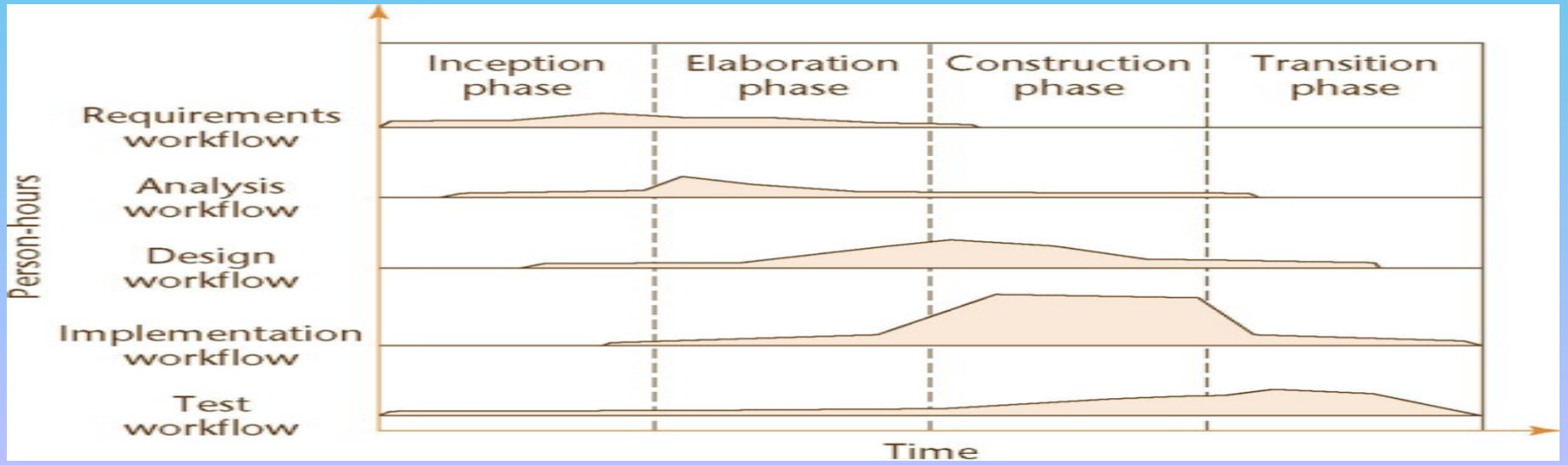


Outline

- **Ethical aspects**
- Historical **aspects**
- Economic **aspects**
- Maintenance **aspects**
-

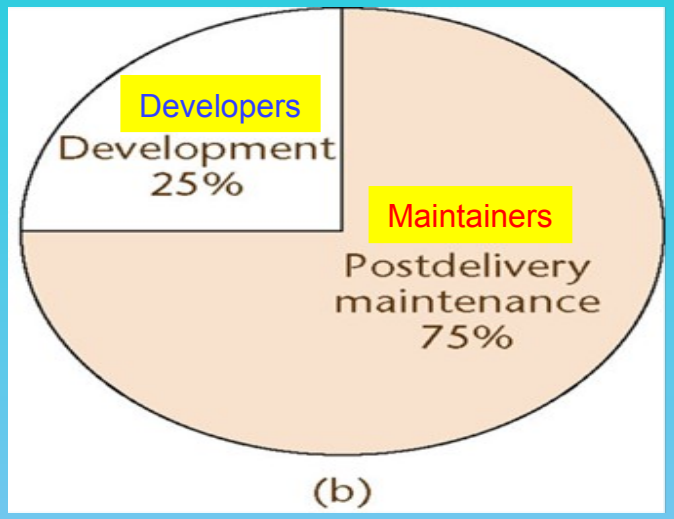


- **Requirements**, **Analysis**, and **Design** **aspects**

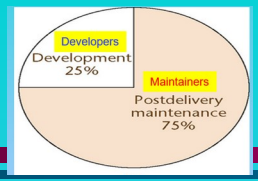


Ethical **aspects**

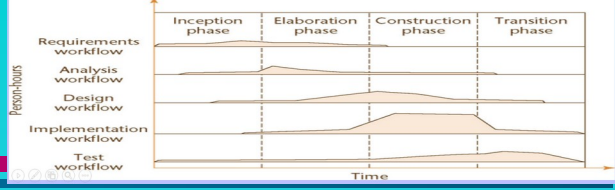
- Developers and Maintainers need to be
  - Hard working
  - Intelligent
  - Sensible
  - Up to date and, above all,
  - Ethical



- IEEE-CS ACM Software Engineering **Code of Ethics** and Professional Practice
  www.acm.org/serving/se/code.htm



Ethical Issues



Software engineers shall commit themselves to making the analysis, design, development, testing and maintenance of **software** a beneficial and respected profession. In accordance with their commitment to the health, safety and welfare of the public, **software engineers** shall adhere to the following Eight Principles:

1. PUBLIC - **software engineers** shall act consistently with the public interest.

2. CLIENT AND EMPLOYER - **software engineers** shall act in a manner that is in the best interests of their client and employer consistent with the public interest.

3. PRODUCT - **software engineers** shall ensure that their **products** and related modifications meet the highest professional standards possible.

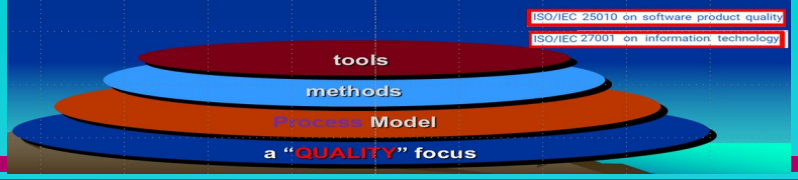
4. JUDGMENT - **software engineers** shall maintain integrity and independence in their professional judgment.

5. MANAGEMENT - **software engineering** managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance.

6. PROFESSION - **software engineers** shall advance the integrity and reputation of the profession consistent with the public interest.

7. COLLEAGUES - **software engineers** shall be fair to and supportive of their colleagues.

8. SELF - **software engineers** shall participate in lifelong learning regarding the practice of their profession and shall promote an **ethical** approach to the practice of the profession.



Today, in many firms **software development** is still viewed as a **creative task**, rather than an **engineering task**.

Most obvious is the ***lack of adherence to standards, principles, and processes*** which have been proven beneficial in similar settings. As a result, **practical software development** does not benefit from the lessons learned in the **Software Engineering** community. This is one important reason for the huge gap between state-of-the-practice and state-of-the-art in **Software Engineering**.

Some of the widely learned lessons ("**Laws**") which *should be adhered to* in any **software development project** – except for justifiable reasons. However, justification of non-adherence to such laws implies **personal responsibility for the consequences** (e.g., Ariane-5 accident!).
http://www.youtube.com/watch?v=kYUrqdUyEpI

The list of "**Laws**" substantiated by documented and analyzed industry experience is large and originated many years back in many cases. ***The more surprising is the wide-spread ignorance***.

Historical **aspects**

- **1968** NATO Conference, Garmisch, Germany
  - Coined the term ***software engineering***
- Aim: To solve the ***software crisis***
- **Software** is delivered
  - **Late**
  - **Over budget**
  - **With residual faults**

Cutter Consortium Data

- 2002 survey of information technology organizations
  - 78% have been involved in disputes ending in litigation
- For the organizations that entered into litigation:
  - In 67% of the disputes, the functionality of the information system as delivered did not meet up to the claims of the developers
  - In 56% of the disputes, the promised delivery date slipped several times
  - In 45% of the disputes, the defects were so severe that the information system was unusable

78% failed

Standish Group Data

- Data on 9236 projects completed in 2004

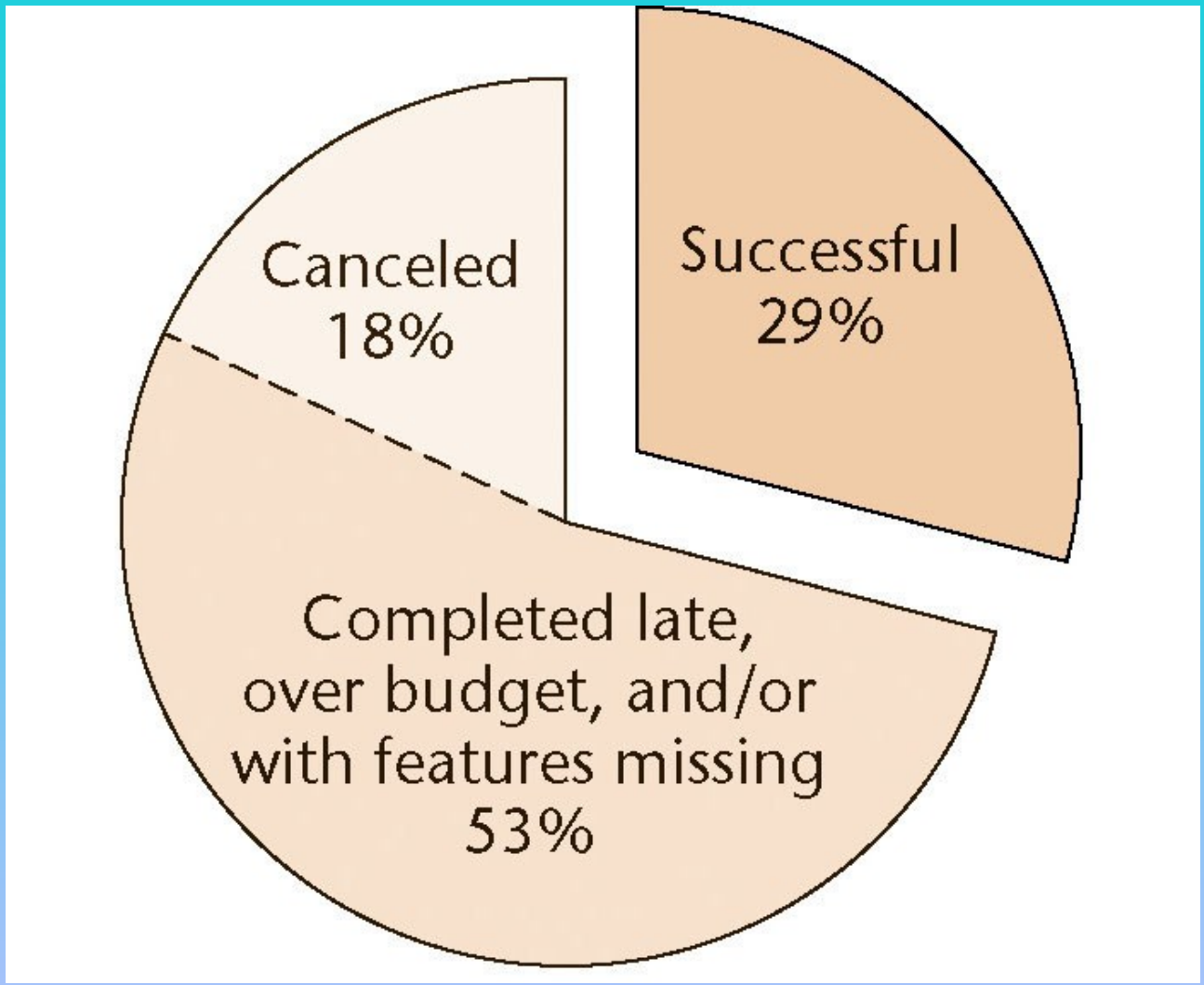


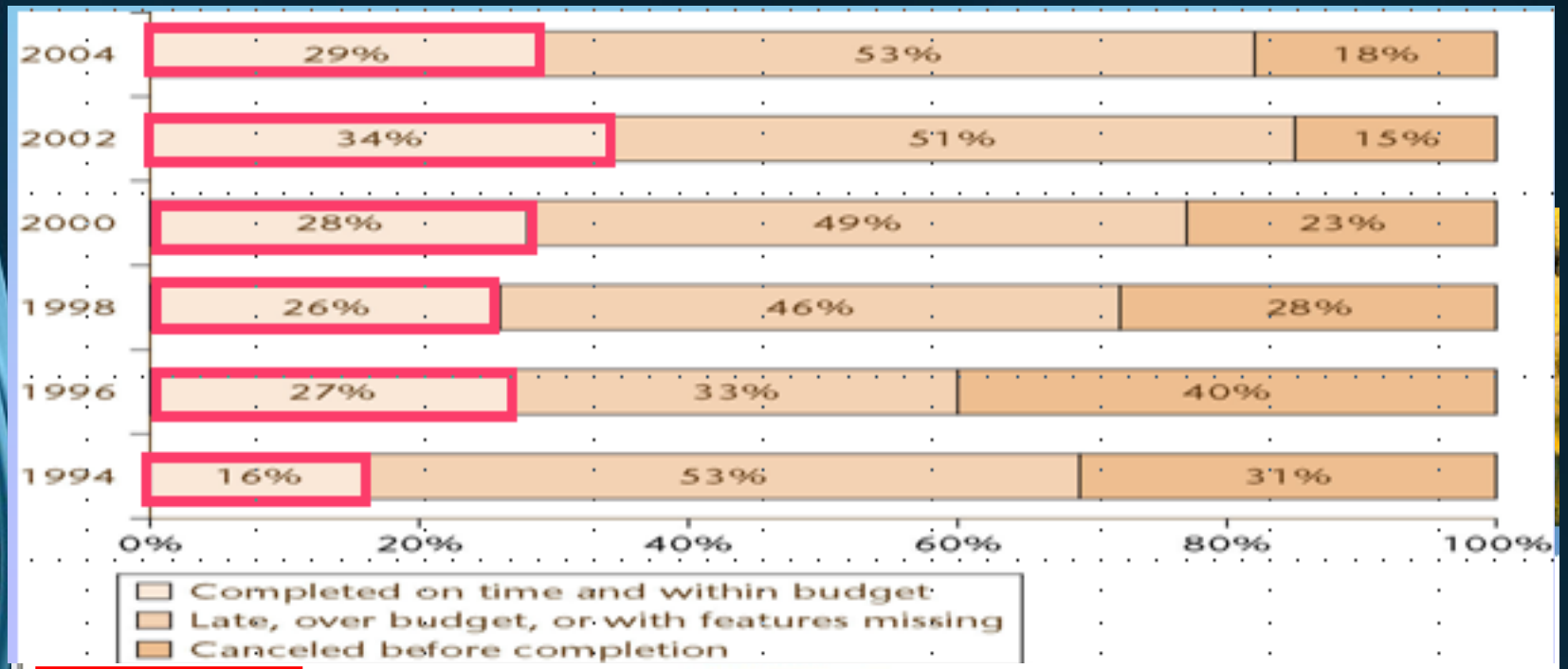
Figure 1.1

71% failed

Myths, Statistics

Project Management Institute (PMI)

http://leadinganswers.typepad.com/leading_answers/2011/04/lies-damn-lies-and-statistics.html

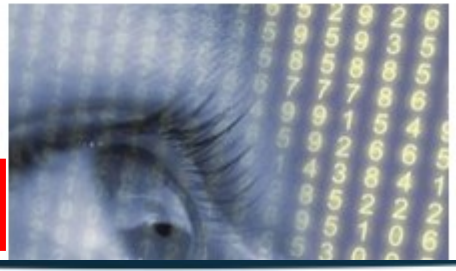


April 29, 2011

Lies, Damn Lies, and Statistics

"Calling Hallelujah Always Offends Someone"

I am glad the PMI is finally recognizing agile methods. Ken Schwaber recently posted about the PMI Agile Certification, saying that he *"...welcomes this and looks forward to PMI shifting from its previous approach to an agile approach. The test of this will be, of course, the success of the projects that adhere to its principles. In the past, the success (or yield) of their predictive approach has been less than 50% of projects (on time, on date, with the desired functionality.)"*



http://www.versionone.com/assets/img/files/ChaosManifesto2013.pdf

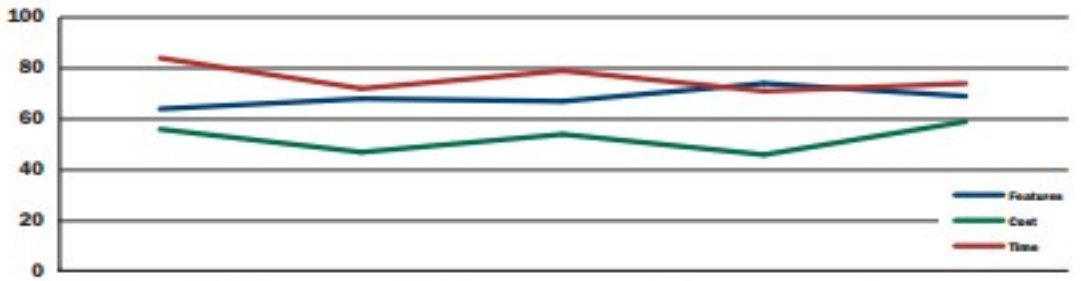
RESOLUTION

	2004	2006	2008	2010	2012
Successful	29%	35%	32%	37%	39%
Failed	18%	19%	24%	21%	18%
Challenged	53%	46%	44%	42%	43%
	71% failed	65% failed	68% failed	63% failed	61% failed

Project resolution results from CHAOS research for years 2004 to 2012.

OVERRUNS AND FEATURES

Time and cost overruns, plus percentage of features delivered from CHAOS research for the years 2004 to 2012.

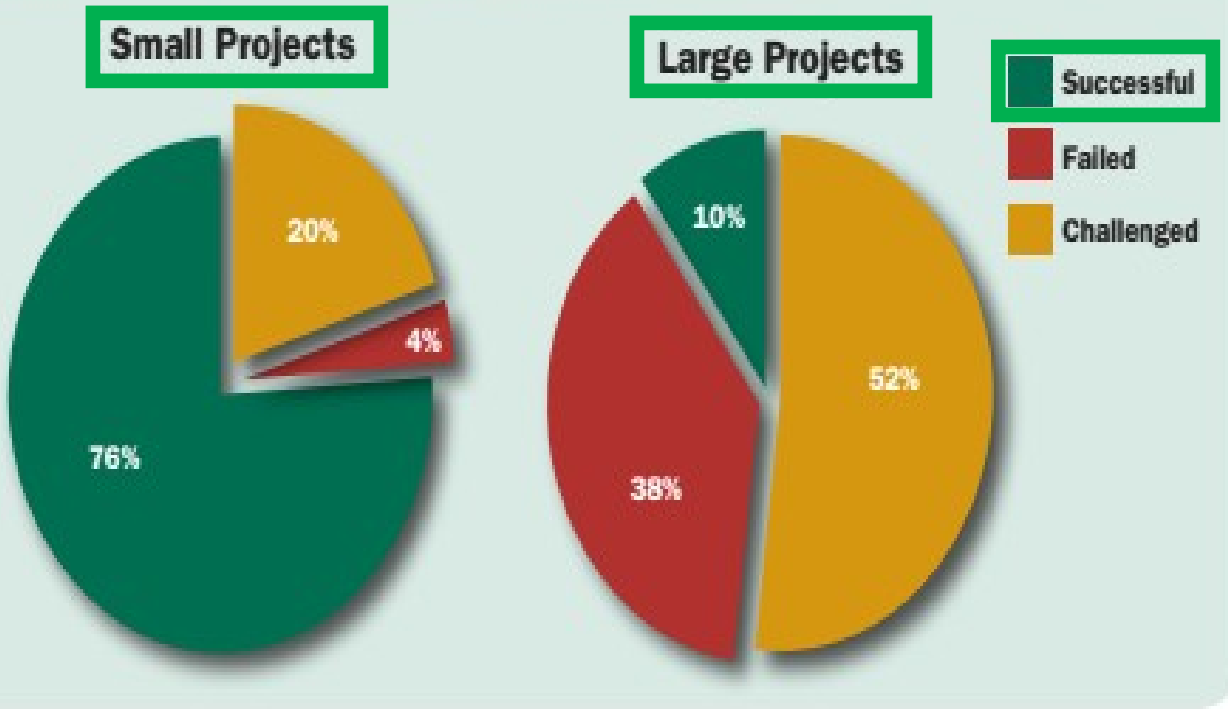


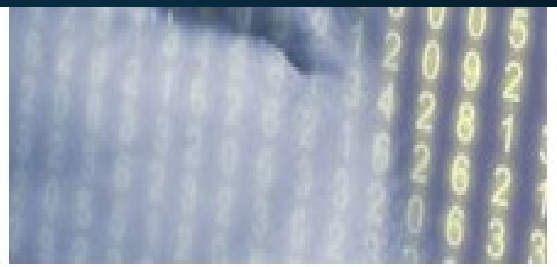
	2004	2006	2008	2010	2012
TIME	84%	72%	79%	71%	74%
COST	56%	47%	54%	46%	59%
FEATURES	64%	68%	67%	74%	69%

http://www.versionone.com/assets/img/files/ChaosManifesto2013.pdf

CHAOS RESOLUTION BY LARGE AND SMALL PROJECTS

Project resolution for the calendar year 2012 in the new CHAOS database. Small projects are defined as projects with less than $1 million in labor content and large projects are considered projects with more than $10 million in labor content.



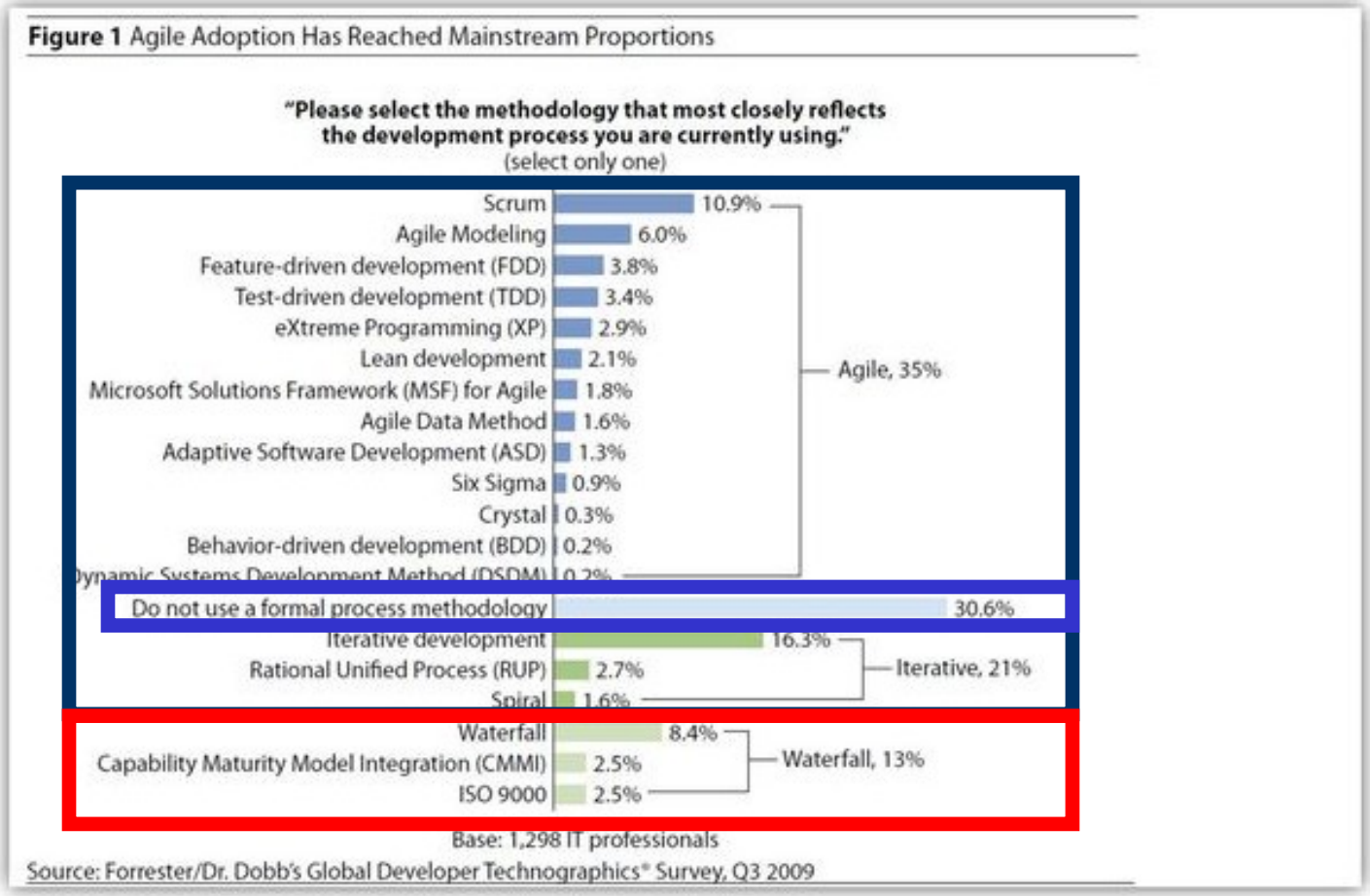


He was quoting from the Standish CHAOS Report that comes out every couple of years and documents the success and failure rates of IT projects. The CHAOS reports have been published since 1994, the same year DSDM appeared and when many agile methods were getting going. Each year the results vary slightly, but the general theme is that many IT projects are challenged and results like the following are typical:

* 32% Successful (On Time, On Budget, Fully Functional)
* 44% Challenged (Late, Over Budget, And/Or Less than Promised Functionality)
* 24% Failed (Cancelled or never used)
* 61% Feature complete

It is interesting then that Ken attributes the poor success rates of IT projects since the start of agile to be a PMI problem. You would think that with the rise of agile methods and the success of all these Scrum, XP, FDD, and DSDM projects we hear about, that these statistics would have turned right around!

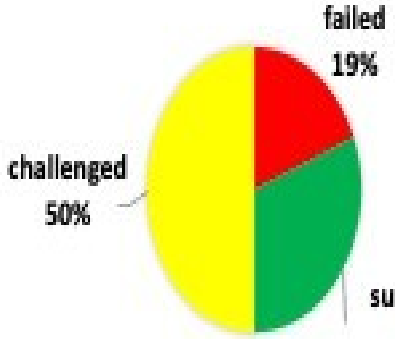
control project managers? Well, not if you believe Forester Research and Dr Dobb's. They indicate that agile methods are now used more than any other approach for IT projects. With 35% of companies using agile and a further 21% using some kind of iterative approach and only 13% using a waterfall approach.



So if 56% of projects, back in 2009, were already getting the adaptive feedback benefits of an iterative approach why are the CHAOS report findings not getting any better? Also the increase in agile continues and last year Gartner predicted *"By 2012 agile development methods will be utilized in 80% of all software development projects"*. So why the poor results, and tell me again why the failures we see are PMI problems caused by "their predictive approach" when it appears these approaches are already in the minority?

Project Success
Quick Reference Card

Based on CHAOS 2020: Beyond Infinity Overview. January 2021, QRC by Henny Portman



Modern measurement
(software projects)

31% successful

Good Sponsor, Good Team, and Good Place are the only things we need to improve and build on to improve project performance.

2022

The Good Place is where the sponsor and team work to create the product. It's made up of the people who support both sponsor and team. These people can be helpful or destructive. It's imperative that the organization work to improve their skills if a project is to succeed. This area is the hardest to mitigate, since each project is touched by so many people. Principles for a Good Place are:

- The Decision Latency Principle
- The Emotional Maturity Principle
- The Communication Principle
- The User Involvement Principle
- The Five Deadly Sins Principle
- The Negotiation Principle
- The Competency Principle
- The Optimization Principle
- The Rapid Execution Principle
- The Enterprise Architecture Principle

Successful project Resolution by Good Place Maturity Level:

highly mature	50%
mature	34%
moderately mature	23%
not mature	23%

The Good Team is the project's workhorse. They do the heavy lifting. The sponsor breathes life into the project, but the team takes that breath and uses it to create a viable product that the organization can use and from which it derives value. Since we recommend small teams, this is the second easiest area to improve. Principles for a Good Team are:

- The Influential Principle
- The Mindfulness Principle
- The Five Deadly Sins Principle
- The Problem-Solver Principle
- The Communication Principle
- The Acceptance Principle
- The Respectfulness Principle
- The Confrontationist Principle
- The Civility Principle
- The Driven Principle

Successful project Resolution by Good Team Maturity Level:

highly mature	66%
mature	46%
moderately mature	21%
not mature	1%

The Good Sponsor is the soul of the project. The sponsor breathes life into a project, and without the sponsor there is no project. Improving the skills of the project sponsor is the number-one factor of success – and also the easiest to improve upon, since each project has only one. Principles for a Good Sponsor are:

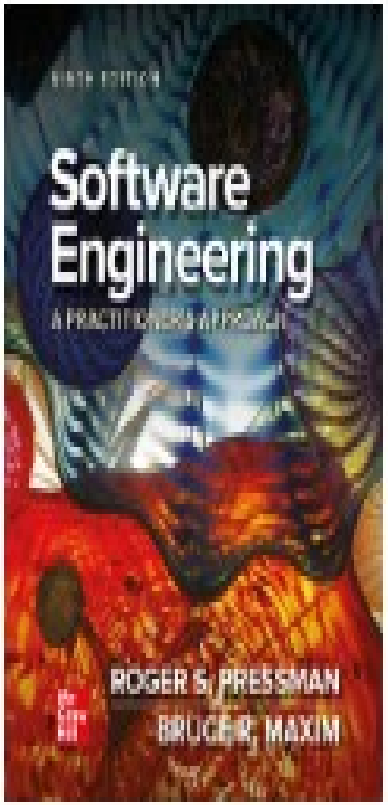
- The Decision Latency principle
- The Vision Principle
- The Work Smart Principle
- The Daydream Principle
- The Influence Principle
- The Passionate Principle
- The People Principle
- The Tension Principle
- The Torque Principle
- The Progress Principle

Successful project Resolution by Good Sponsor Maturity Level:

highly mature	67%
mature	33%
moderately mature	21%
not mature	18%

Conclusion

- The **software crisis** has not been solved
- Perhaps it should be called the ***software depression***
  - **Long duration**
  - **Poor prognosis**



Product: Software Engineering: A Practitioner's Approach

Edition: 9th

Author: Roger Pressman,Bruce Maxim

ISBN10: 1260915050

29.2 SOFTWARE ENGINEERING AS A DISCIPLINE

For almost 50 years, many academic researchers and industry professionals have clamored for a true engineering discipline for software. In an important follow-on to her classic 1990 paper on the subject, Mary Shaw [Sha09] comments on this continuing quest:

> Engineering disciplines typically evolve from craft practices of a technology, sufficient for local or ad hoc use. When the technology becomes economically significant, it requires stable production techniques and management control. The resulting commercial market is based on experience, rather than a deep understanding of the technology . . . an engineering profession emerges when . . . science becomes sufficiently mature to support purposeful practice and design evolution with predictable outcomes.

We would argue that the industry has achieved "purposeful practice," but that "predictable outcomes" have remained elusive.

As mobility begins to dominate the software landscape, Shaw identifies challenges that "emerge from the deep interdependencies between very complex systems and their users" [Sha09]. She argues that the knowledge base that leads to "purposeful practice" has been democratized by the specialized social networks that now populate the Web. For example, rather than referencing a centrally controlled software engineering handbook, a software developer can pose a problem on an appropriate forum and obtain a crowd-sourced solution that draws from the experience of many other developers. The proposed solution if often critiqued in real time, with alternatives and adaptations offered as options.

But this is not the level of discipline that many demand. As Shaw states: "[P]roblems facing software engineers are increasingly situated in complex social contexts and delineating the problem's boundaries is increasingly difficult" [Sha09]. As a consequence, isolating the scientific underpinnings of a discipline remains a challenge. At this point in the history of our field, it is reasonable to state that "the discovery of new software engineering ideas is, by now, naturally incremental and evolutionary" [Erd10].

29.3 Observing Software Engineering Trends

Barry Boehm [Boe08] suggests that "software engineers [will] face the formidable challenges of dealing with rapid change, uncertainty and emergence, dependability, diversity, and interdependence, but they also have opportunities to make significant contributions that will make a difference for the better." But what are the trends that will enable you to face these challenges in the years ahead?

We sat in a development area at one of the most successful game developers on the planet. Over the years, earlier generations of the gam he demoed sold over 50 million copies and generated billions of dollars in revenue.

"So, when will this version be on the market?" I asked.

He shrugged. "In about five months, and we've still got a lot of work to do."

He had responsibility for game play and artificial intelligence functionality in an application that encompassed more than three million lines of code.

"Do you guys use any software engineering techniques?" I asked, half-expecting that he'd laugh and shake his head.

He paused and thought for a moment. Then he slowly nodded. "We adapt them to our needs, but sure, we use them."

"Where?" I asked, probing.

"Our problem is often translating the requirements the creatives give us."

"The creatives?" I interrupted.

"You know, the guys who design the story, the characters, all the stuff that make the game a hit. We have to take what they give us and produce a set of technical requirements that allow us to build the game."

"And after the requirements are established?"

He shrugged. "We have to extend and adapt the architecture of the previous version of the game and create a new product. We have to create code from the requirements, test the code with daily builds, and do lots of things that your book recommends."

"You know my book?" I was honestly surprised.

"Sure, used it in school. There's a lot there."

"I've talked to some of your buddies here, and they're more skeptical about the stuff in my book."

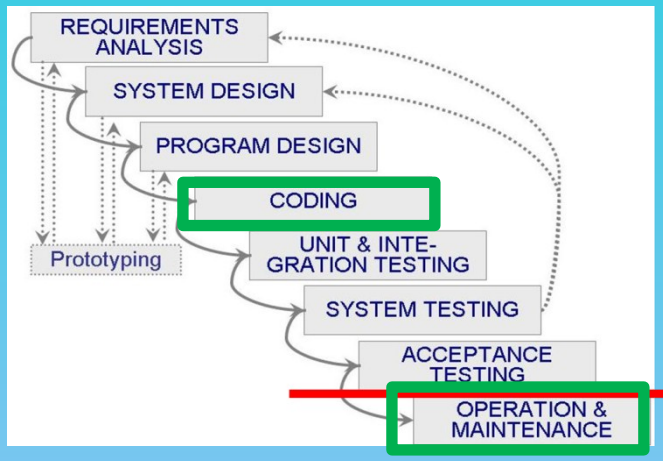
He frowned. "Look, we're not an IT department or an aerospace company, so we have to customize what you advocate. But the bottom line is the same—we need to produce a high-quality product, and the only way we can accomplish that in a repeatable fashion is to adapt our own subset of software engineering techniques."

"And how will your subset change as the years pass?"

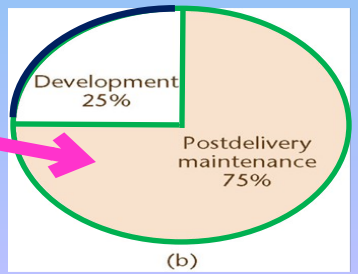
He paused as if to ponder the future. "Games will become bigger and more complex, that's for sure. And our development timelines will shrink as more competition emerges. Slowly, the games themselves will force us to apply a bit more development discipline. If we don't, we're dead."

Economic **aspects**

- Coding method CM_{new} is 10% faster than currently used method CM_{old}. **Should it be used?**



- Common sense answer **?????**
  - Of course!
- **Software Engineering** answer
  - Consider the cost of training
  - Consider the impact of introducing a new technology
  - Consider the effect on **Maintenance**

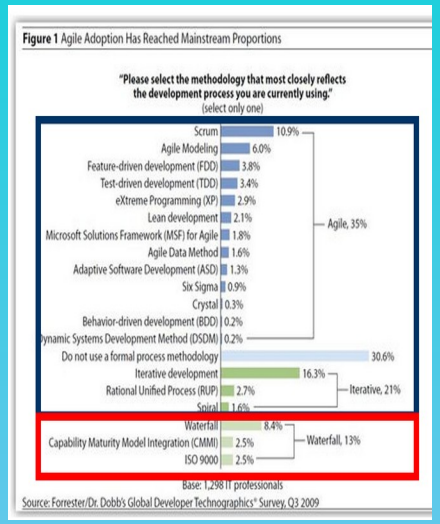


Maintenance aspects

State of art

Life cycle **Model**

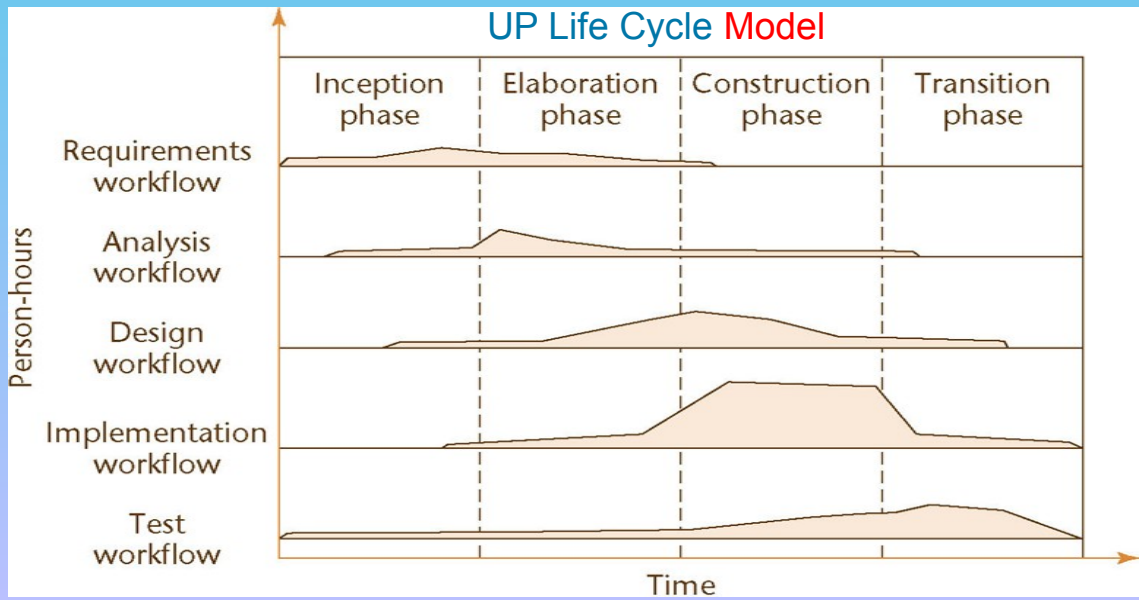
- The steps (workflows & *phases*) to follow when building **software products**
- A theoretical description of what should be done



State of practice

Life cycle

- The actual steps performed on a **specific product**

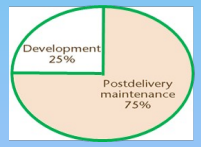


Waterfall Life Cycle Model

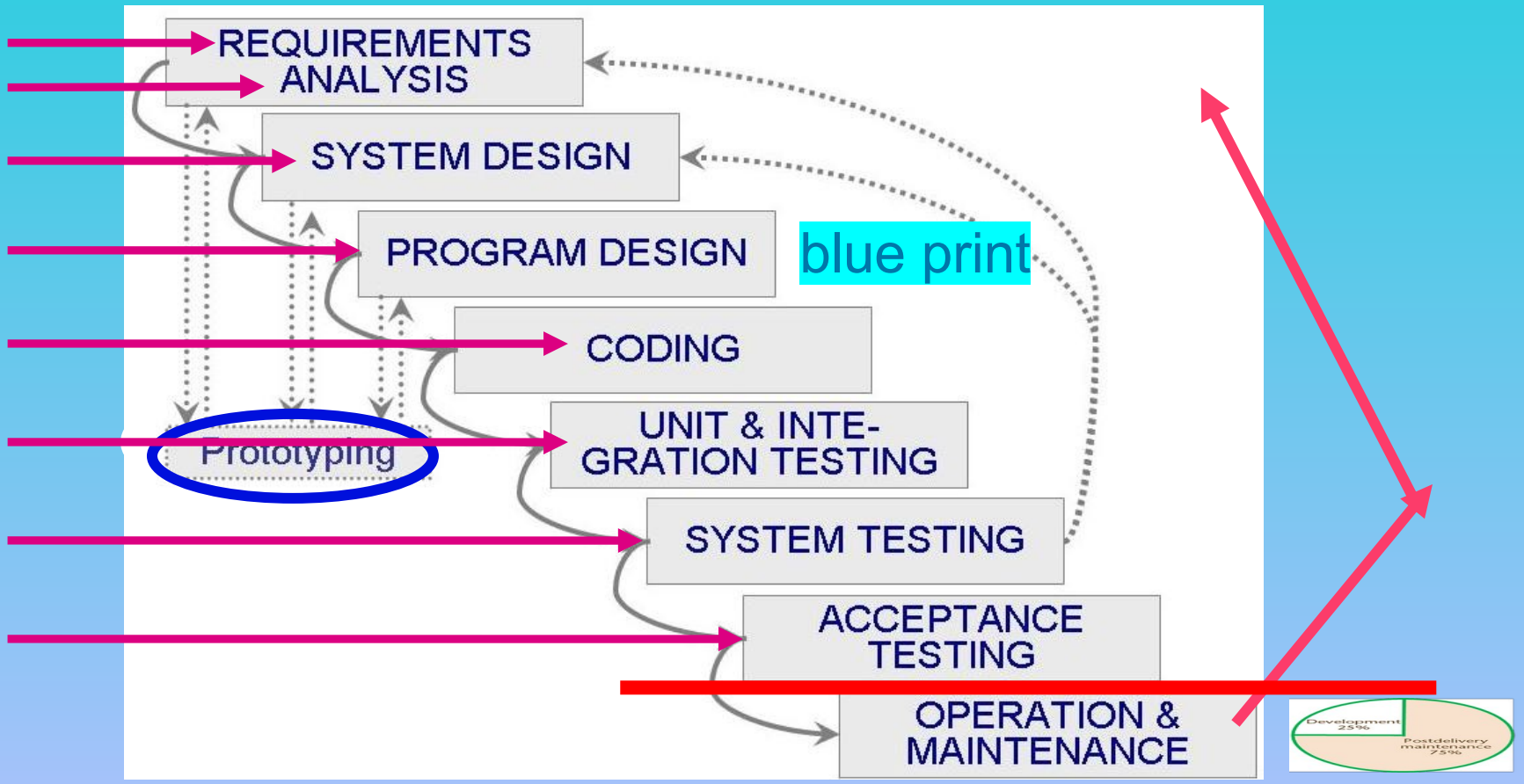
- **Classical Model (1970) Structured Paradigm ("C")**

1. Requirements phase
2. Analysis (specification) phase
3. Design phase
4. Implementation phase

5. Postdelivery maintenance
6. Retirement



Waterfall Life Cycle **Model** with Prototyping



Existing Laws

Requirements

Analysis

Design

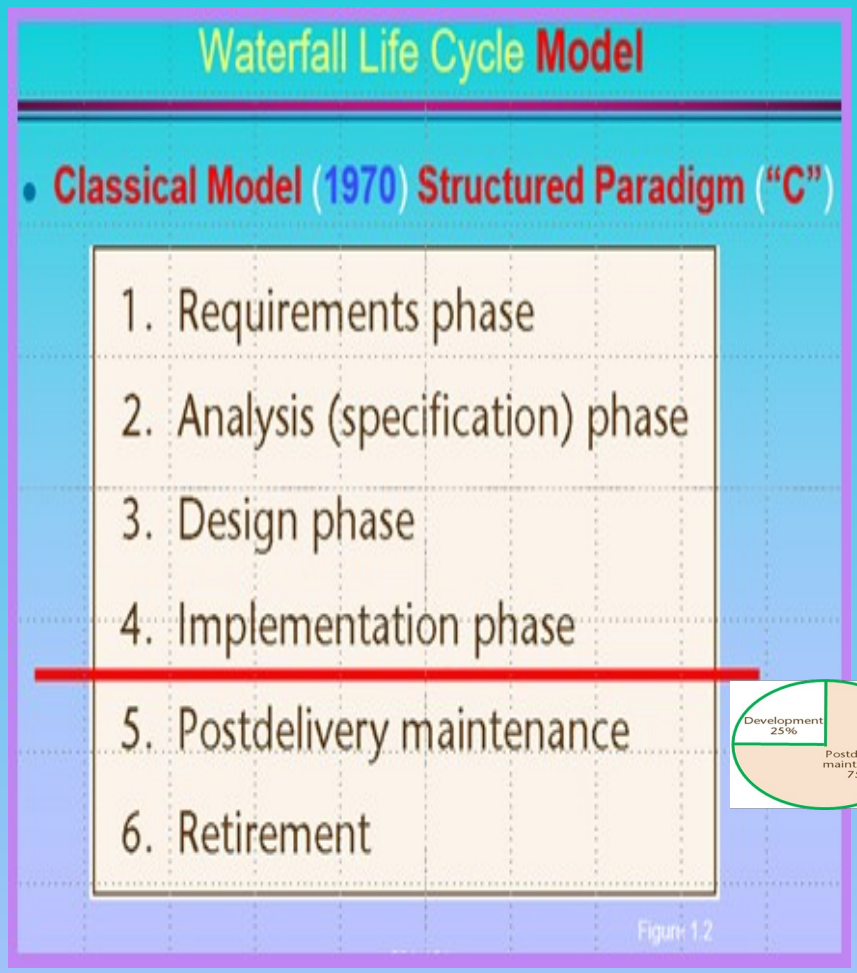
Implementation

Verification

Validation

Evolution

Project management



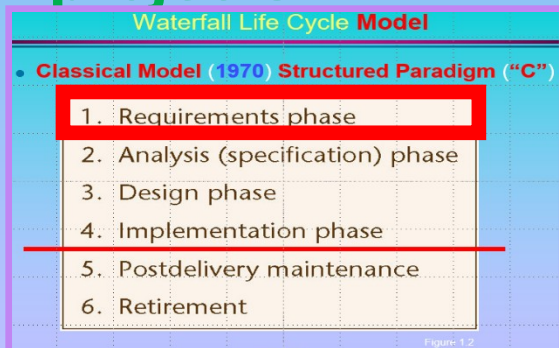
Laws

Requirements deficiencies are the prime source of **project** failures (L1)

- Source: Robert Glass [Glas 98] et al
- Most defects (> 50%) stem from **Requirements**
- **Requirements** defects (if not removed quickly) trigger follow-up defects in late activities

Possible solutions:

- early inspections
- **formal** specifications & validation early on
- other forms of **Prototyping** & validation early on
- **Reuse** of **Requirements** documentation from **similar projects**



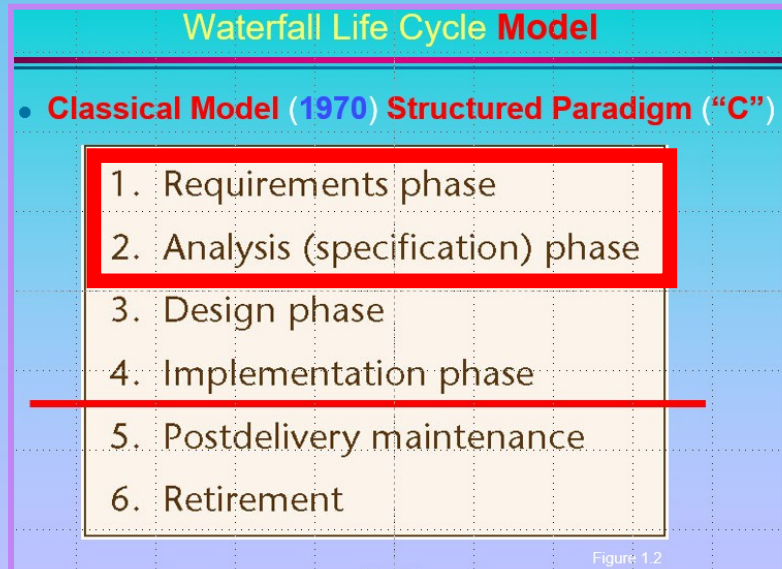
Laws

Defects are most frequent during **Requirements** and **Analysis** activities and are more expensive the later they are removed (L2)

- Source: Barry Boehm [Boeh 75] et al
- >80% of defects are caused up-stream (**Requirements**, **Analysis**)
- Removal delay is expensive (e.g., factor 10 per phase delay)

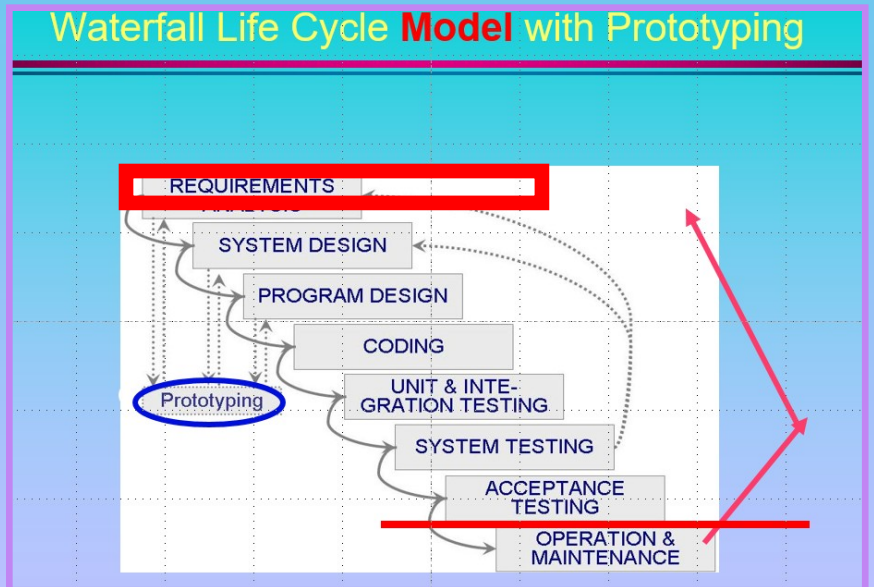
HealthCare.gov October 21, 2013

For large software projects, failure is generally determined early in the process, because failures almost exclusively have to do with planning: the failure to create a workable plan, to stick to it, or both. Healthcare.gov reportedly involved over fifty-five contractors, managed by a human-services agency that lacked deep experience in software engineering or project management. The final product had to be powerful enough to navigate any American through a complex array of different insurance offerings, secure enough to hold sensitive private data, and robust enough to withstand peak traffic in the hundreds of thousands, if not millions, of concurrent users. It also had to be simple enough so that anyone who can open a Web browser could use it. In complexity, this is a project on par with the F.B.I.'s V.C.F. or Sentinel. The number and variety of systems to be connected may not be quite as large, but the interface had to be usable by anyone, without special training. And, unlike V.C.F., Healthcare.gov was given only twenty-two months from contract award to launch—less than two years for a project similar to one that took the F.B.I. more than ten years and over twice the budget.



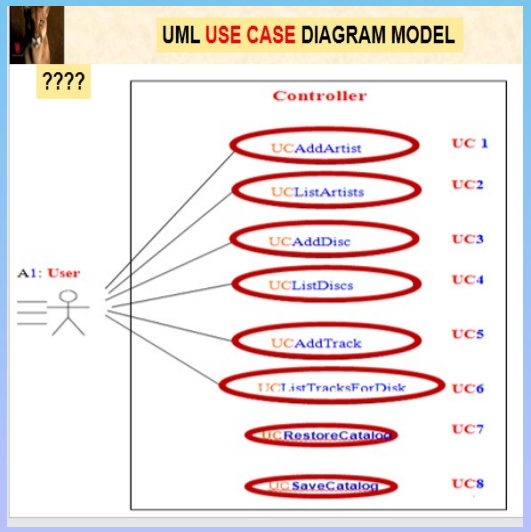
Laws

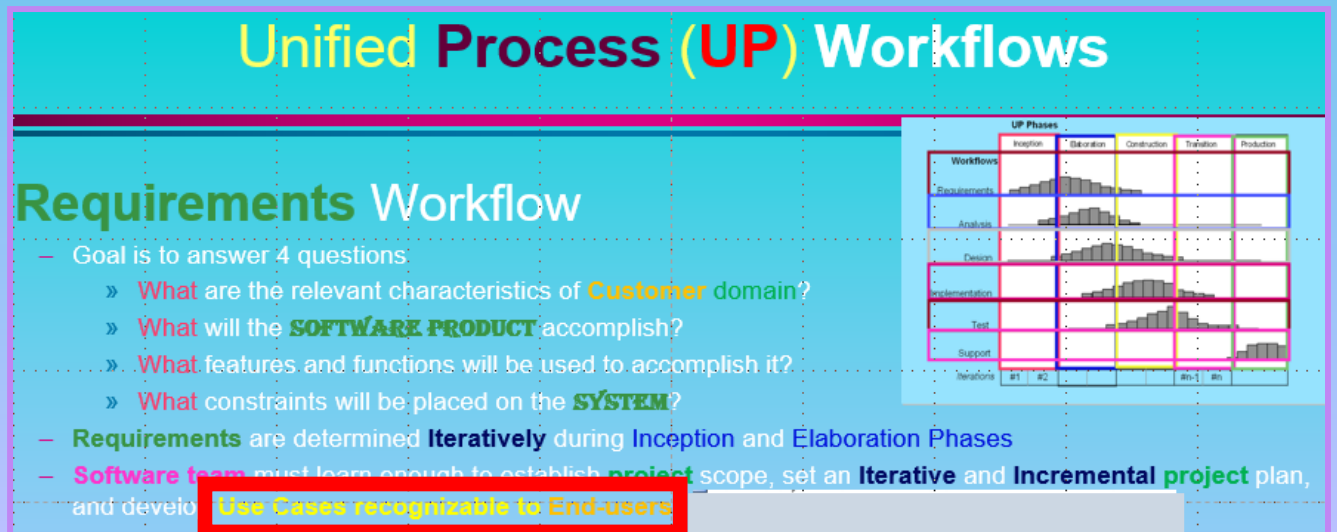
- **Prototyping** (significantly) reduces **Requirements** and **Design** defects, especially those related to the **U**ser **I**nterface (L3)
  - Source: Barry Boehm [Boeh 84a]
  - See: Prototype Life cycle Model



Laws

- The value of a **Model** depends on the view taken, but none is best for all purposes (L4)
  - Source: Alan Davis [Davi 90]
  - **Requirements Model** suitable for stakeholders decreases the likelihood of inconsistencies and incompleteness





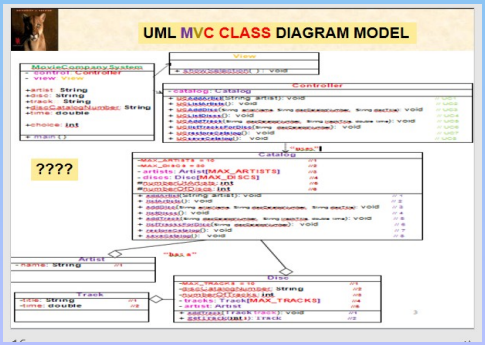
Laws

Good **Designs** require deep application **Domain knowledge** (L5)

- Source: Bill Curtis et al [Curt 88, Curt 90]
- "Goodness" is defined as stable and locally changeable (diagonalized requirements x component matrix)
- Key principle: **information hiding**
- **Domain knowledge** allows prediction of possible changes/variations

Hierarchical structures reduce complexity (L6)

- Examples: large mathematical functions, operating systems (layers), books (chapter structure), ….



Laws

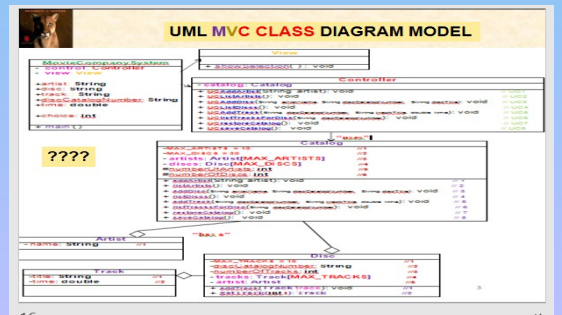
Incremental Processes reduce complexity (L6a)

- Source: Harlan Mills (Cleanroom) [MIL 87]
- Large tasks need to be refined in a number of comprehensible tasks
- Examples: **Iterative Life cycle Model**, **Incremental** verification & inspection

Laws

- A **structure** is stable if **cohesion** is strong & **coupling** is low (L7)
  - Source: Stevens, Myers, and Constantine [Stev 74]
  - High **Cohesion** allows changes (to one issue) locally
  - Low **Coupling** avoids spill-over or so-called ripple effects

- Only what is hidden can be changed without risk (L8)
  - Source: David Parnas [Parn 72]
  - **Information Hiding** applied properly leads to strong Cohesion/low Coupling
  - See: Y2K-Problem



Laws

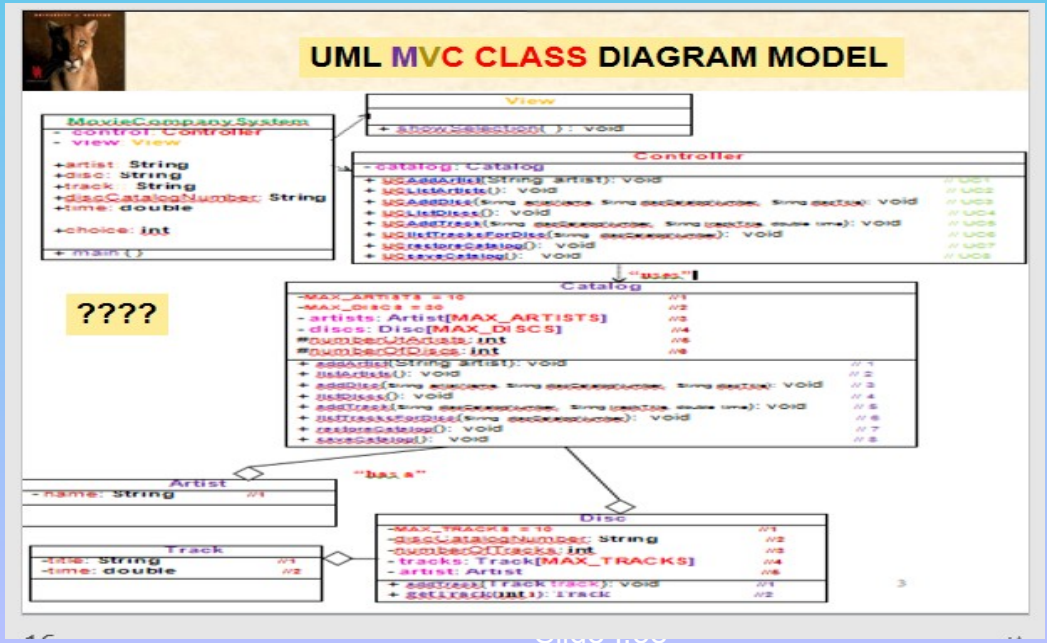
- **Well-structured programs** have fewer defects and are easier to **Maintain** (L13)
  - Source: Edsger Dijkstra [Dijk 69], Harlan Mills [Mil 71], and Niklaus Wirth [Wirt 71]
  - e.g., well-structured imperative **programs** use for control flow sequence, alternative & iteration only

- **Software Reuse** reduces **cycle time** and increases **productivity** and **quality** (L15)
  - Source: Doug McIlroy, in 1968 Garmisch Conference[Naur 69b]
  - **Reuse** of proven **software** **avoids** defects and **saves** development time
  - **Reuse** is only possible if it is well understood and trusted ´(a) what its services are, (b) what its degree of verification & validation is, and (c) what its integration constraints are

Laws

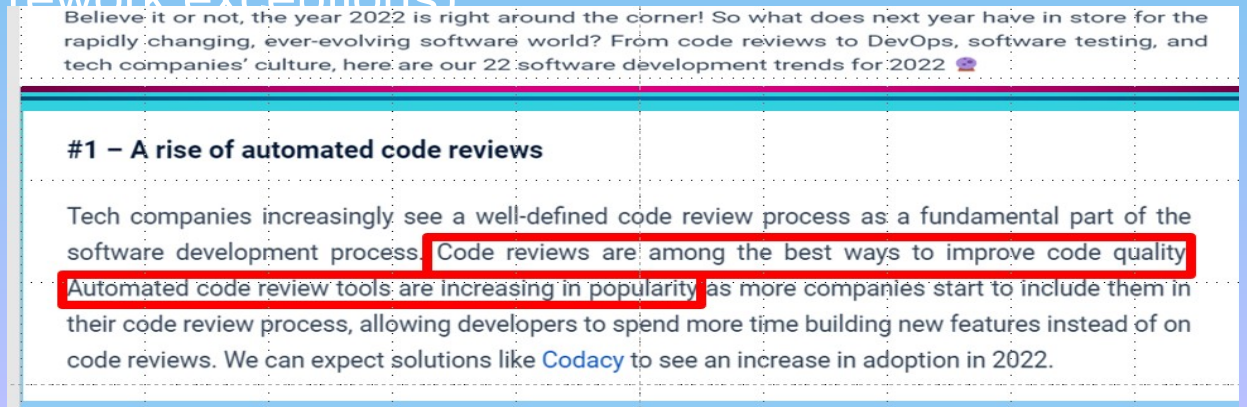
- **O**bject **O**riented **P**aradigm reduces **defects** and encourages **Reuse** (L17)
  - Source: Ole-Johan Dahl [Dahl 67], Adele Goldberg [Gold 89]
  - First languages: Simula 67, Smalltalk, Java
  - Based on **Information Hiding** via **Classes** & increased **Reuse** potential

MVC



Laws

- **Inspections** significantly increase productivity, quality and project stability (L17)
  - Source: Mike Fagan [Faga 76, Faga 86]
  - Early defect detection increases productivity (less rework, lower cost per defect)
  - Early defect detection increases quality (no follow-up defects, testing of clean code at the end ☾ quality certification)
  - Early defect detection increases project stability (better plannable due to fewer rework exceptions)



Believe it or not, the year 2022 is right around the corner! So what does next year have in store for the rapidly changing, ever-evolving software world? From code reviews to DevOps, software testing, and tech companies' culture, here are our 22 software development trends for 2022

#1 – A rise of automated code reviews

Tech companies increasingly see a well-defined code review process as a fundamental part of the software development process. Code reviews are among the best ways to improve code quality. Automated code review tools are increasing in popularity as more companies start to include them in their code review process, allowing developers to spend more time building new features instead of on code reviews. We can expect solutions like Codacy to see an increase in adoption in 2022.

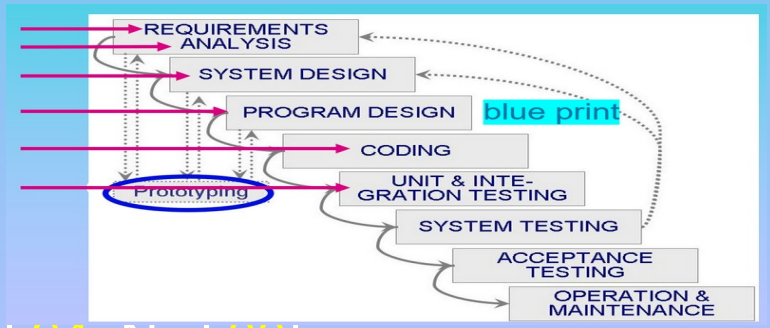
Laws

Testing can show the presence but not the absence of defects (L22)

- Source: Edsger Dijkstra [Dijk 70]
- by definition (as a sampling technique)
- Empirical data shows that in large system **testing** covers only a fraction of possible usages (**less than 25%**)

A **developer is unsuited** to test his or her code (L23)

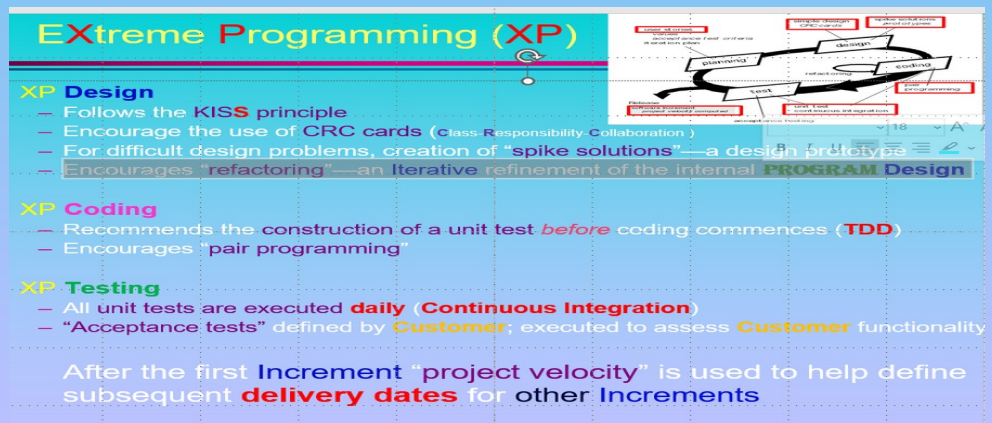
- Source: Weinberg [Wein 71]
- Developer can devise **UNIT test cases**
- See: Cleanroom
- Usability is quantifiable (L26)
- Source: Jacob Nielsen, Doug Norman [Niel 94, Niel 00]



Laws

- A **system** that is used **will be changed** (L27)
  - Source: Many Lehman [Lehm 80]
  - **IBM OS/360**
    - » grew from 1 MLoC to 8 MLOC in 3 years
    - » Induced **2 defects** for **any 1 removed**

- **An evolving system** increases **complexity** unless work is done to reduce it (L28)
  - Source: Many Lehman [Lehm 80]
  - evolving **systems** must be re-engineered (**refactor**) in regular intervals



Laws

Individual developer productivity varies considerably (L31)

- Source: Sackmann [Sack 68]

Development effort is a (non-linear) function of product size (L33)

- Source: Barry Boehm [Boeh 81, Boeh 00c]
- See: COCOMO **Model**

Adding resources to a late project makes it later (L36)

- Source: Fred Brooks [Broo 75]

1975

HealthCare.gov October 21, 2013

"The Mythical Man-Month," Fred Brooks writes that "adding manpower to a late software project makes it later." This is known as Brooks's Law, and it is taken as gospel by programmers because it is usually true: it takes so much time for new coders to comprehend the system that they're supposed to be fixing that typically it would have been faster not to include them at all.

Typical Classical phases

Requirements phase

- Explore the concept
- Elicit the Client's Requirements

"WHAT the product is supposed to do"

- Start the IEEE SRS
- Start the IEEE DRS

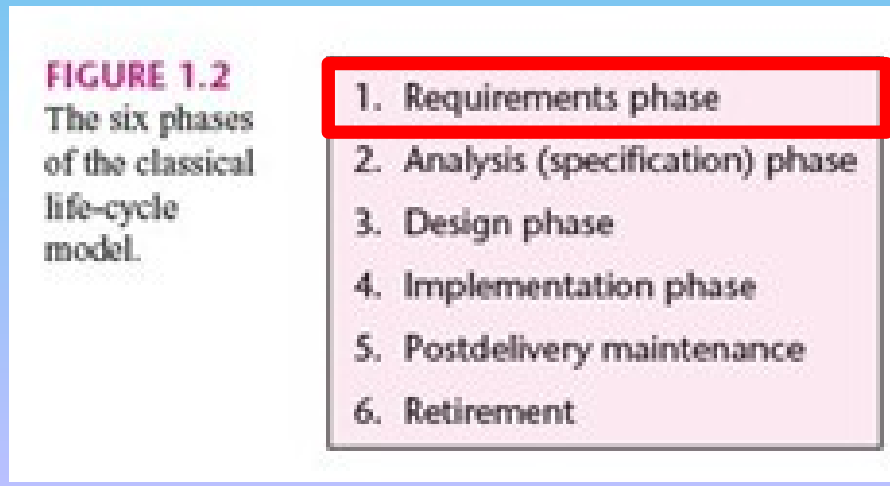


FIGURE 1.2
The six phases of the classical life-cycle model.

Typical Classical phases

Analysis (Specification) phase

- Analyze the Client's Requirements
- Draw up the Specification document Software Requirements Specification (IEEE SRS)
- Draw up the Data Specification document Data Requirements Specification (IEEE DRS)
- Draw the Black Box Test Cases (BBTC)
- Draw up the Software Project Management Plan (IEEE SPMP)

"WHAT the product is supposed to do

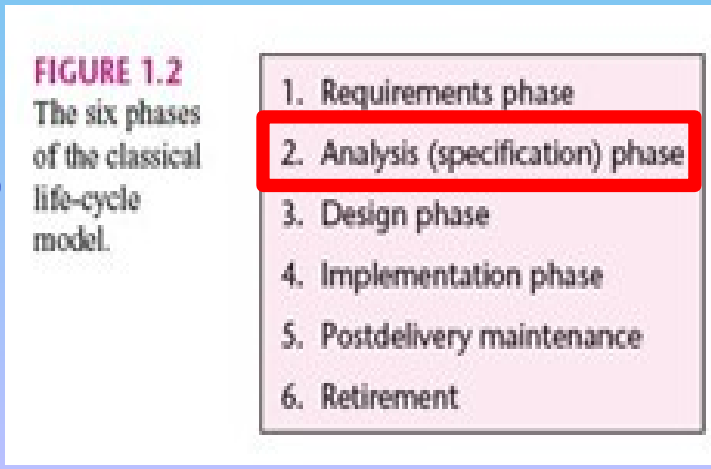


FIGURE 1.2
The six phases of the classical life-cycle model.

Typical Classical phases

- **Design phase**
  - Architectural design, followed by
  - Detailed design
  - "**How** the **product** does it"
- **Implementation phase**
  - Coding
  - Unit Testing
  - Integration Testing
  - System Testing
  - **Acceptance Testing**

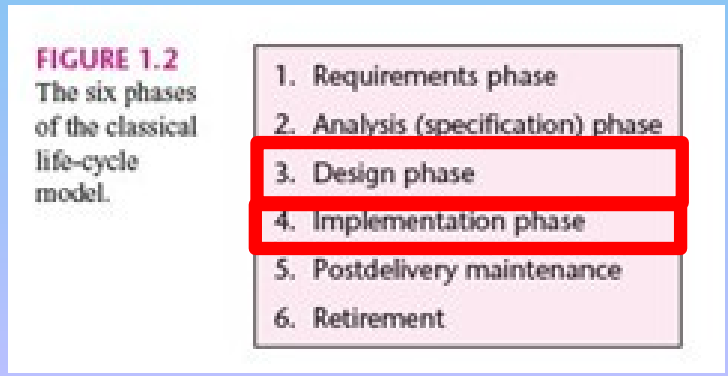
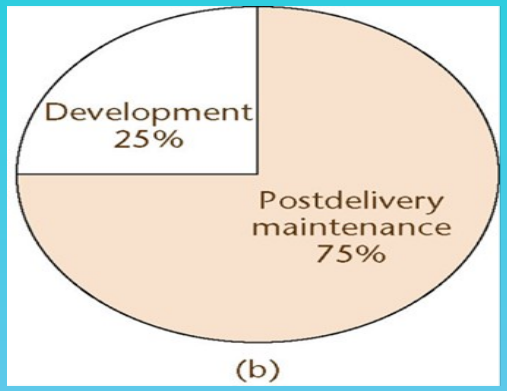


FIGURE 1.2 The six phases of the classical life-cycle model.

Typical Classical phases

- **Postdelivery Maintenance phase**
  - Corrective Maintenance
  - Perfective Maintenance
  - Adaptive Maintenance



- **Retirement phase**

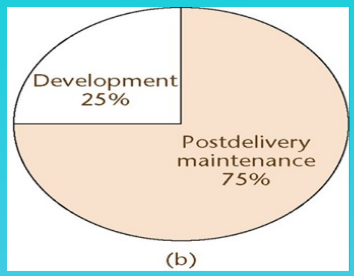
"**product** discarded"

FIGURE 1.2
The six phases of the classical life-cycle model.

1. Requirements phase
2. Analysis (specification) phase
3. Design phase
4. Implementation phase
5. Postdelivery maintenance
6. Retirement

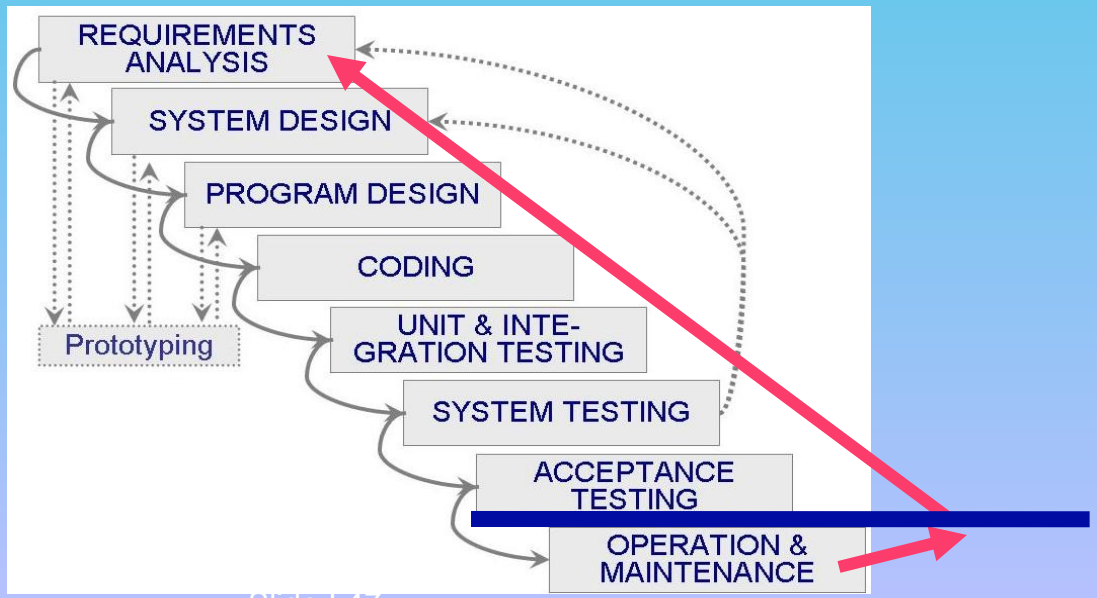
The Importance of **Postdelivery Maintenance**

Bad **software** is discarded

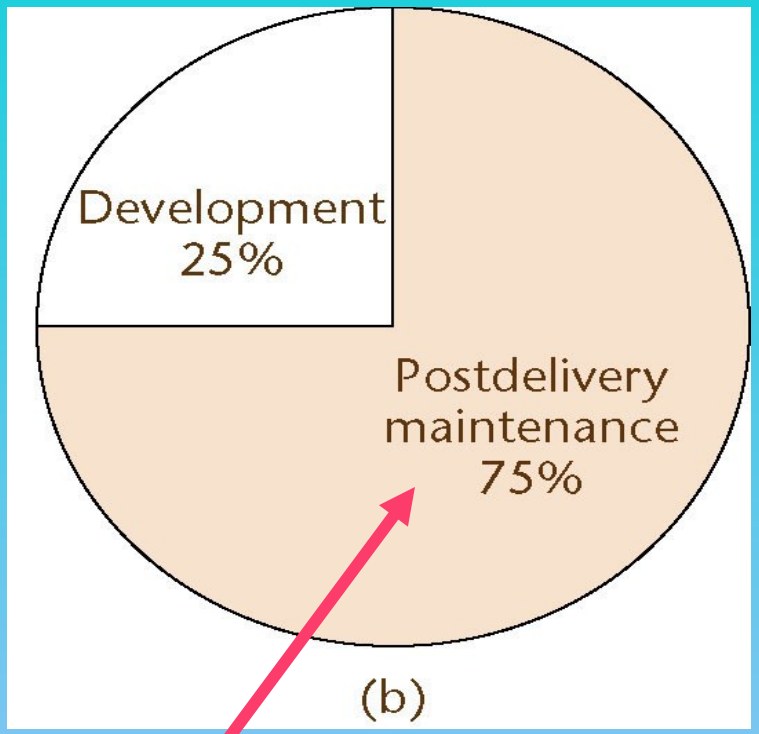


Good **software** is maintained, for 10, 20 years or more

Software is a **Model** of reality, which is **constantly changing**



Time (= Cost) of Postdelivery Maintenance



You are in charge of automating a multi- site architectural practice. The cost of developing the software has been estimated to be $ 5,000,000. Approximately how much **additional money** will be needed for **postdelivery maintenance of the software**?

$15,000,000

What would you say?

The **Costs** of the Classical **phases**

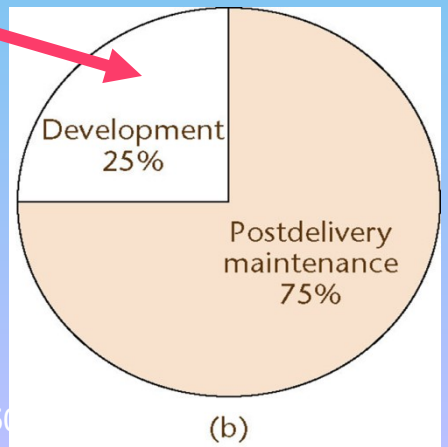
- Surprisingly, the **costs** of the Classical **phases** have hardly changed

	Various Projects between 1976 and 1981	132 More Recent Hewlett-Packard Projects
Requirements and analysis (specification) phases	21%	18%
Design phase	18	19
Implementation phase		
Coding (including unit testing)	36	34
Integration	24	29

Coding method CM_{new} is 10% faster than currently used method CM_{old}. **Should it be used?**

FIGURE 1.2
The six phases of the classical life-cycle model.

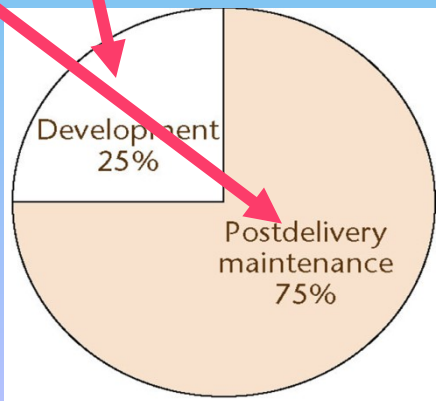
1. Requirements phase
2. Analysis (specification) phase
3. Design phase
4. Implementation phase



Consequence of Relative Costs of phases

- Return to CT_{old} and CT_{new}
- Reducing the coding **cost** by 10% yields at most a 0.85% reduction in total **costs** (10% X (34% of 25%) = 0.85%)
  - Consider the expenses and disruption incurred
- Reducing Postdelivery Maintenance cost by 10% yields a 7.5% reduction in overall costs (10% X (75% of 100%) = 7.5%)

	132 More Recent Hewlett-Packard Projects
Requirements and analysis (specification) phases	18%
Design phase	19
Implementation phase	
Coding (including unit testing)	34
Integration	29



(b)

Requirements, Analysis, and Design Aspects

The **earlier we detect and correct** a **fault**, the less it **costs** us

Requirements, Analysis, and Design Aspects

- The **cost** of detecting and correcting a **fault** at each phase

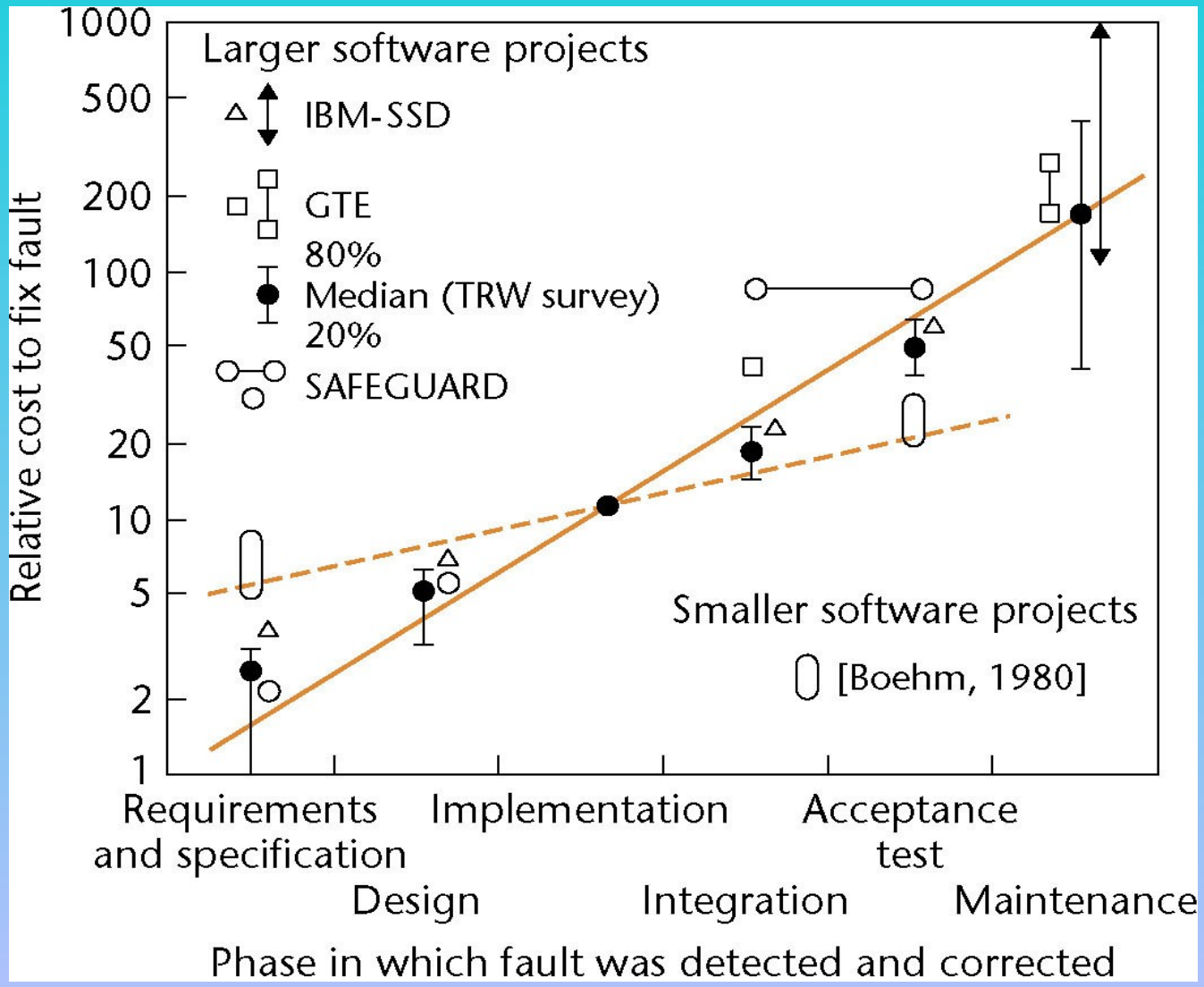


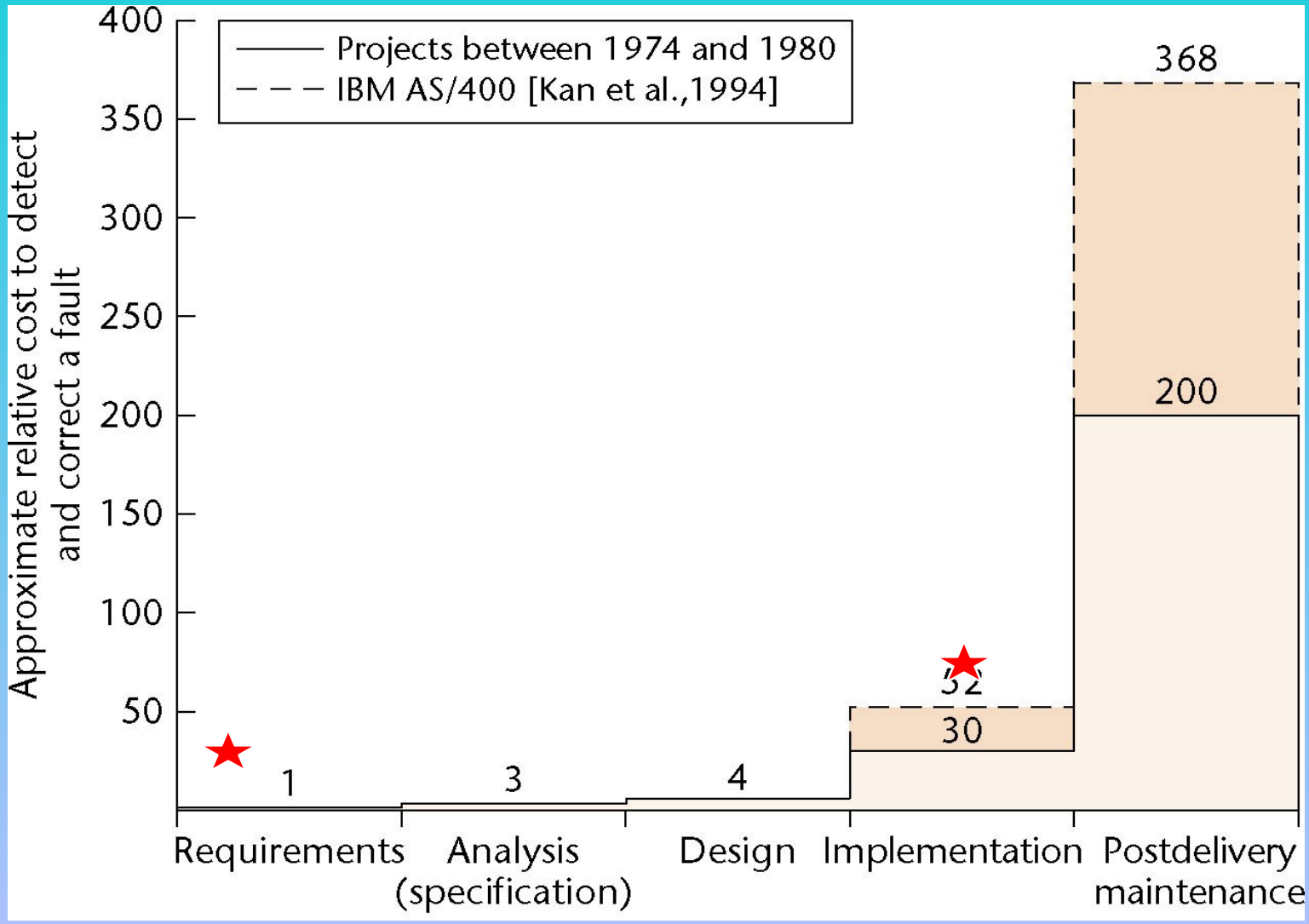
FIGURE 1.2
The six phases of the classical life-cycle model.

1. Requirements phase
2. Analysis (specification) phase
3. Design phase
4. Implementation phase
5. Postdelivery maintenance

Figure 1.5

Requirements, Analysis, and Design Aspects

The previous figure **redrawn** on a linear scale

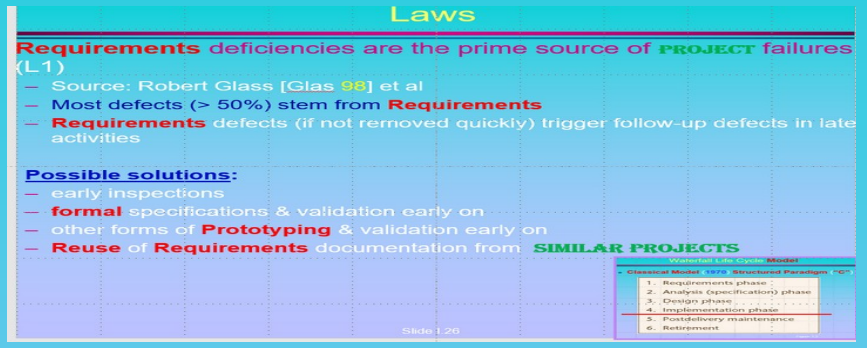


Major FAULT in the UML **USE CASE** MODEL

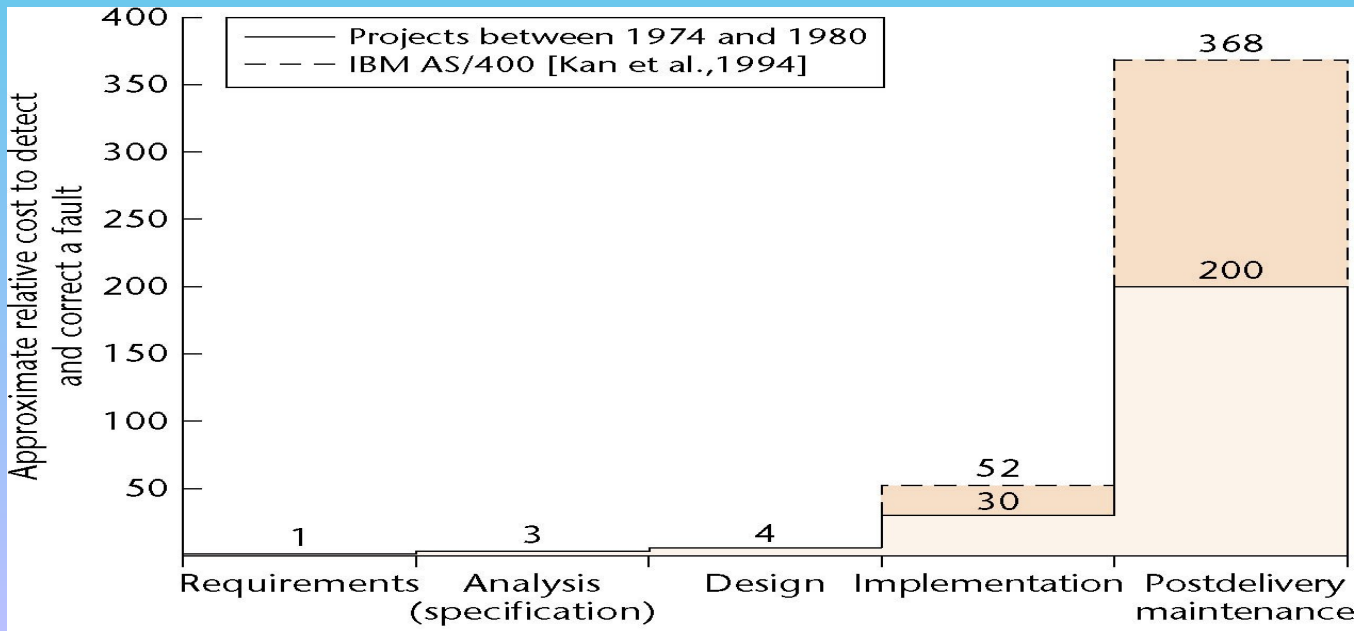
Figure 1.6

Conclusion

- It is vital to improve our Requirements, Analysis, and Design **techniques**
  - To find **faults** as early as possible
  - To reduce the overall number of **faults** (and, hence, the overall **cost**)



1. Requirements phase
2. Analysis (specification) phase
3. Design phase



The Object Oriented Paradigm aspects

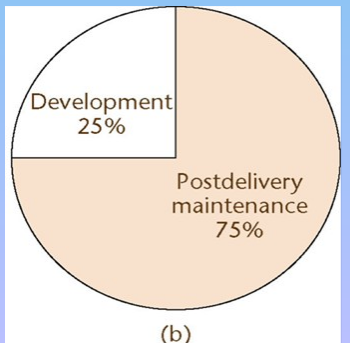
The **Structured Paradigm (C)** was successful (**?**) initially

- It started to fail with **larger products** (> 50,000 LOC)

Postdelivery **Maintenance** problems (today, 70 to 80% of total effort)

Reason: **Structured** methods are

- **Action** oriented (e.g., finite state machines, Data Flow Diagrams); or
- **Data** oriented (e.g., Entity Relationship Diagrams, Jackson's method);
- But not both



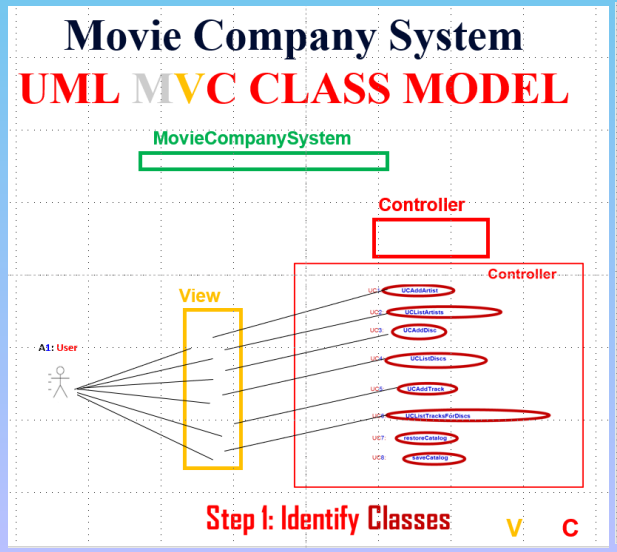
Strengths of the Object Oriented Paradigm

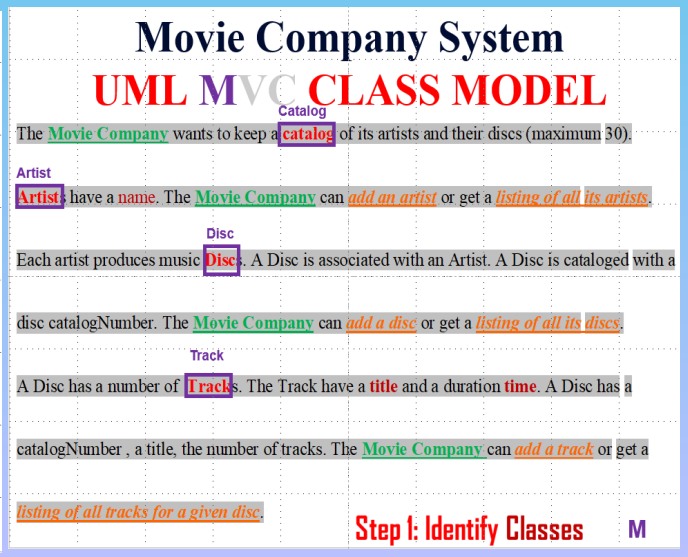
With **Information Hiding**, postdelivery **Maintenance** is safer

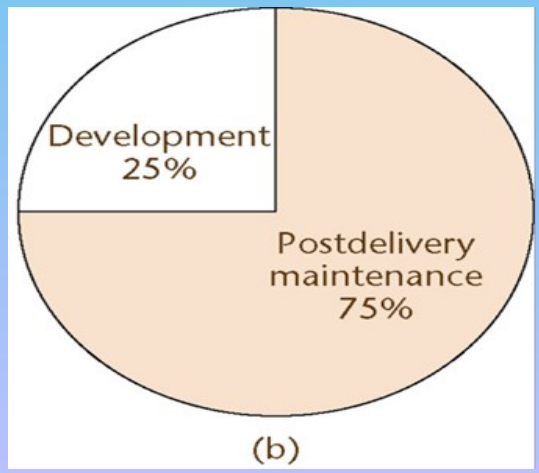
- The chances of a **regression fault** are reduced

Development is easier

- **Objects** generally have physical counterparts
- This simplifies **Modeling** (a key aspect of the Object Oriented Paradigm)



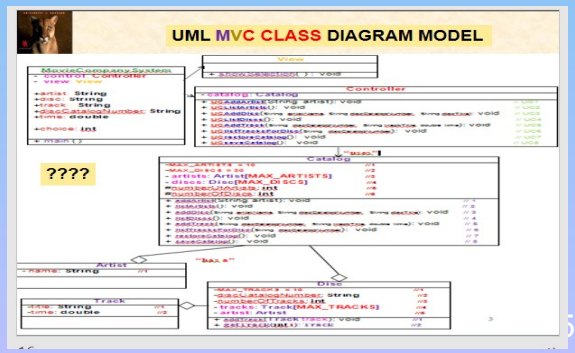


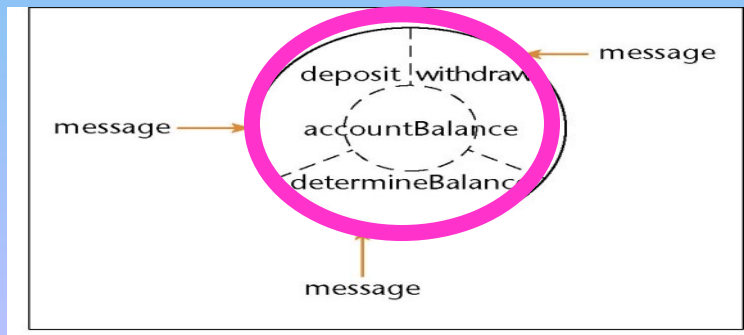


Strengths of the Object Oriented Paradigm

Well-designed **objects** are **independent** units (high **COHESION** / low **COUPLING** - **MVC**)

- Everything that relates to the real-world item being modeled is in the corresponding **object** — ***encapsulation***
- Communication is by sending *messages*
- This **independence** is enhanced by ***Responsibility-Driven Design***





(b)

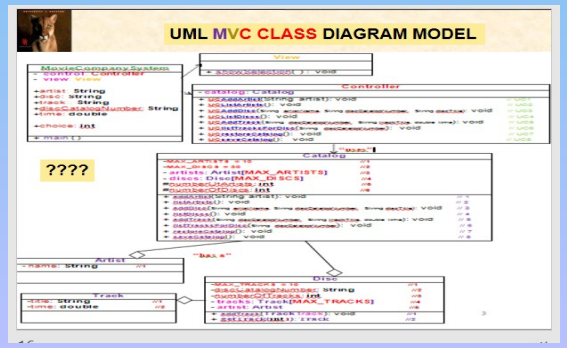
Strengths of the Object Oriented Paradigm

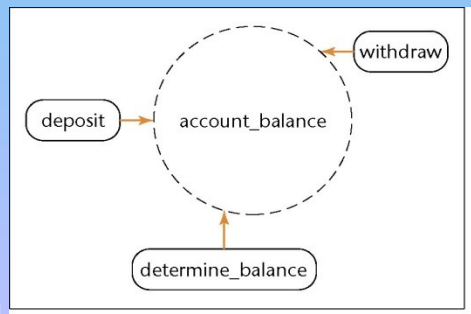
A **Classical product** conceptually consists of a single unit (although it is implemented as a set of **modules**)

The **O**bject **O**riented **P**aradigm reduces complexity because the **product** generally consists of **independent** units

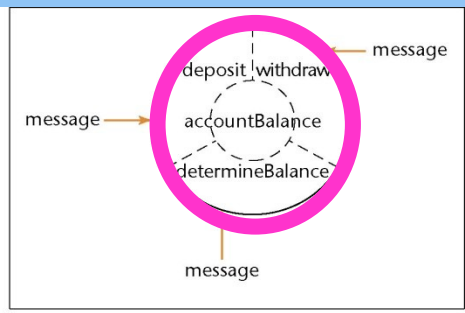
The **O**bject **O**riented **P**aradigm promotes **Reuse**

- **objects** are **independent** entities





(a)

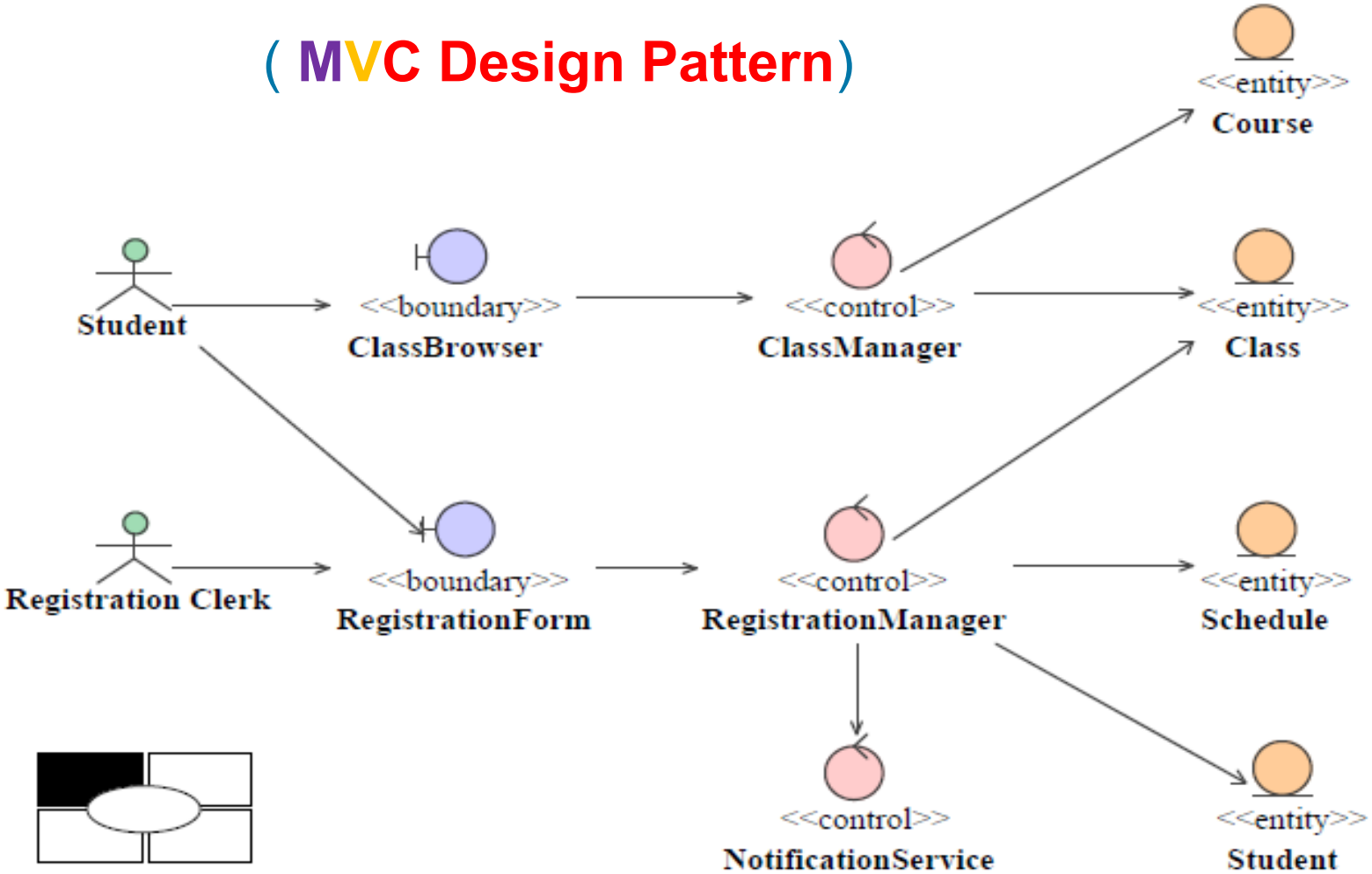


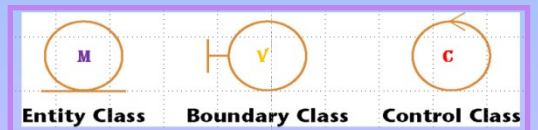
(b)

UML OOA Robustness Diagram

Extending UML – Robustness Diagram

(MVC Design Pattern)





Responsibility driven Design

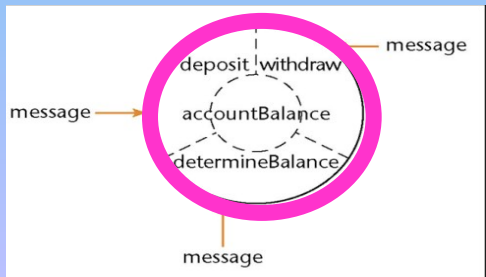
Also called ***Design by Contract***

Send flowers to your mother in Chicago

- Call `1-800-flowers`
- Where is `1-800-flowers`?
- Which Chicago florist does the delivery?

- **Information Hiding**

- Send a message to a **method** [**action**] of an **object** without knowing the internal structure of the **object**



Responsibility driven Design

Also called ***Design by Contract***

- **Information Hiding**
- Send a message to a **method** [**action**] of an **object** without knowing the internal structure of the **object**

Movie Company System
UML MVC CLASS MODEL

```
Create the instance of the Catalog class
Catalog catalog = new Catalog();

Since the catalog is private we need to call a public method in the catalog to
operate on the private instance variables of the catalog

public void UCAddArtist(String artist){
   catalog.addArtist(artist);
}                                          // UC1

public void UCListArtists(){
   catalog.listArtists();
}                                          // UC2

public void UCAddDisc(String artistName, String discCatalogNumber, String
discTitle){
   catalog.addDisc(artistName, discCatalogNumber, discTitle);
}                                          // UC3

public void UCListDiscs(){
   catalog.listDiscs();
}                                          // UC4

public void UCAddTrack(String discCatalogNumber, String trackTitle, double time){
   catalog.addTrack(discCatalogNumber, trackTitle, time);
}                                          // UC5

public void UClistTracksForDisc(String discCatalogNumber){
   catalog.listTracksForDisc(discCatalogNumber);
}                                          // UC6

public void UCrestoreCatalog(){
   catalog.restoreCatalog();
}                                          // UC7

public void UCsaveCatalog(){
   catalog.saveCatalog();
```

Controller	
- catalog: Catalog	
+ UCAddArtist(String artist): void	// UC1
+ UCListArtists(): void	// UC2
+ UCAddDisc(String artistName, String discCatalogNumber, String discTitle): void	// UC3
+ UCListDiscs(): void	// UC4
+ UCAddTrack(String discCatalogNumber, String trackTitle, double time): void	// UC5
+ UClistTracksForDisc(String discCatalogNumber): void	// UC6
+ UCrestoreCatalog(): void	// UC7
+ UCsaveCatalog(): void	// UC8

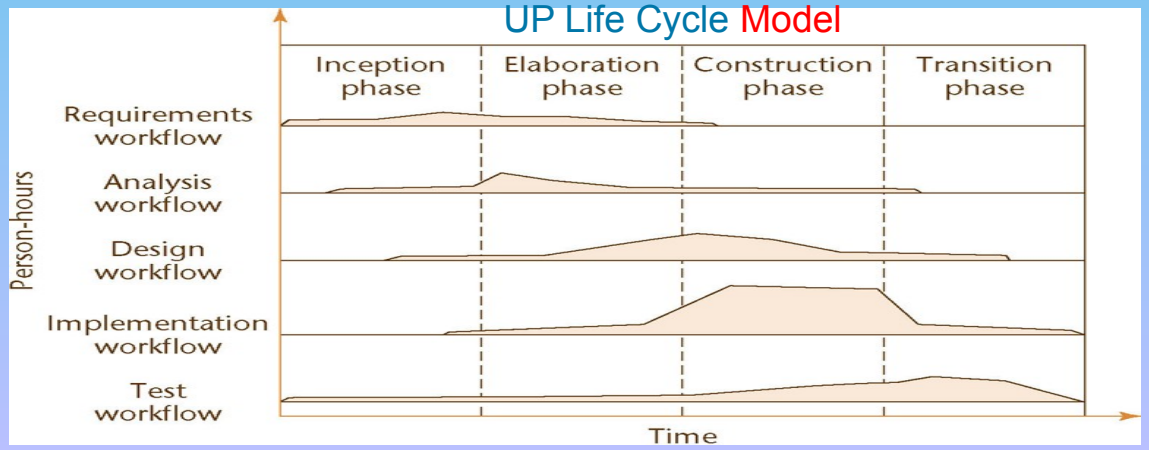
Step 5: Detailed Design Pseudocode for **methods**

Classical phases vs Object Oriented Workflows

Classical Paradigm	Object-Oriented Paradigm
1. Requirements phase	1. Requirements workflow
2. Analysis (specification) phase	2′. Object-oriented analysis workflow
3. Design phase	3′. Object-oriented design workflow
4. Implementation phase	4′. Object-oriented implementation workflow
5. Postdelivery maintenance	5. Postdelivery maintenance
6. Retirement	6. Retirement

Figure 1.8

- There is no correspondence between phases and Workflows



Analysis/Design "Hump"

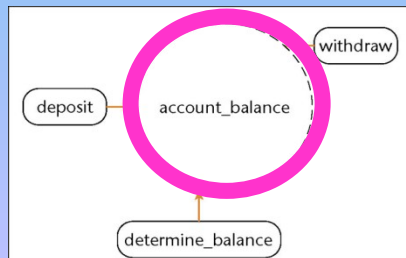
- **Structured** Paradigm:
  - There is a jolt between **Analysis** (**what**) and **Design** (**how**)

- **O**bject **O**riented **P**aradigm:
  - **Objects** enter from the very beginning
    (**UML OOA MVC CLASS DIAGRAM Model**)

Classical Paradigm	Object-Oriented Paradigm
1. Requirements phase	1. Requirements workflow
2. Analysis (specification) phase	2′. Object-oriented analysis workflow
3. Design phase	3′. Object-oriented design workflow
4. Implementation phase	4′. Object-oriented implementation workflow
5. Postdelivery maintenance	5. Postdelivery maintenance
6. Retirement	6. Retirement

Analysis/Design "Hump"

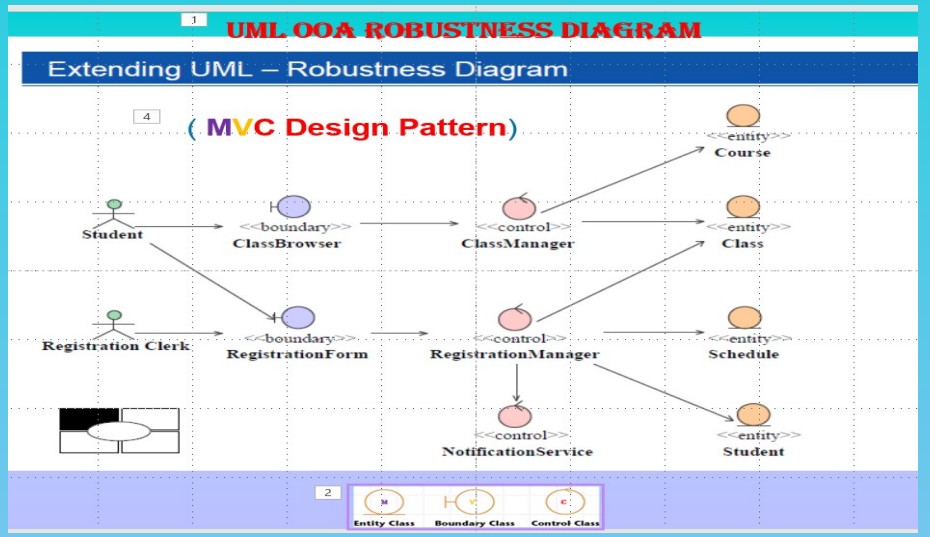
- In the **Classical** Paradigm
  - **Classical** Analysis
    - » Determine **what** has to be done
  - **Classical** Design
    - » Determine **how** to do it
    - » Architectural design — determine the modules
    - » Detailed design — design each module (pseudocode)



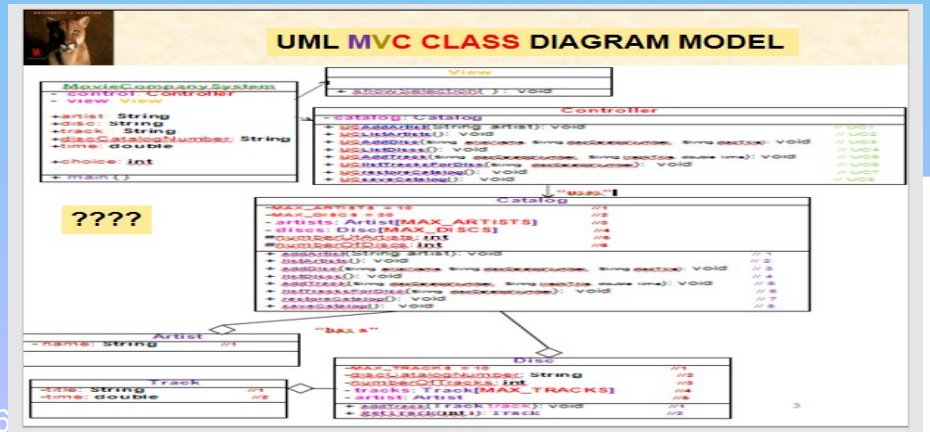
Removing the "Hump"

In the Object Oriented Paradigm

- Object Oriented Analysis
  - » Determine **what** has to be done
  - » Determine the objects



- Object Oriented Design
  - » Determine **how** to do it
  - » Design the objects



In More Detail

Classical Paradigm	Object-Oriented Paradigm
2. Analysis (specification) phase • Determine what the product is to do	2′. Object-oriented analysis workflow • Determine what the product is to do • Extract the classes
3. Design phase • Architectural design (extract the modules) • Detailed design	3′. Object-oriented design workflow • Detailed design
4. Implementation phase • Code the modules in an appropriate programming language • Integrate	4′. Object-oriented implementation workflow • Code the classes in an appropriate object-oriented programming language • Integrate

Figure 1.9

- **objects** enter here

(**UML OOA MVC CLASS DIAGRAM Model**)

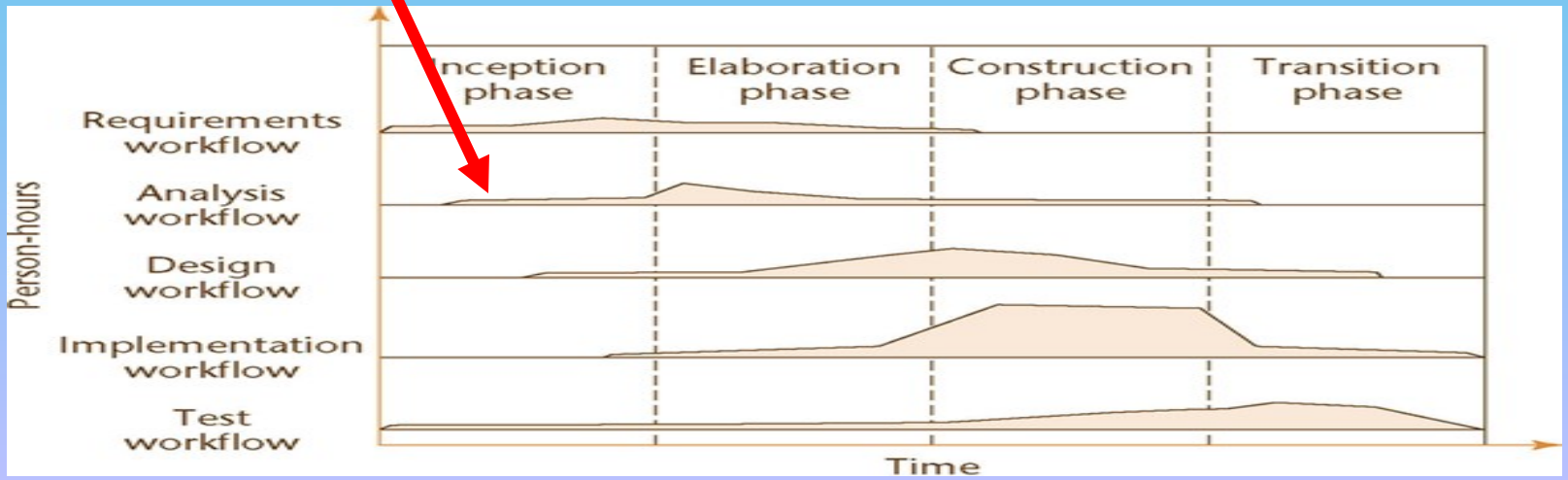
Object Oriented Paradigm

objects are introduced as early as the **O**bject **O**riented **A**nalysis **workflow**

- This ensures a **smooth transition** from the **OO** Analysis workflow to the **OO** Design workflow

The **objects** are then coded during the **Implementation** workflow

- Again, the **transition is smooth**



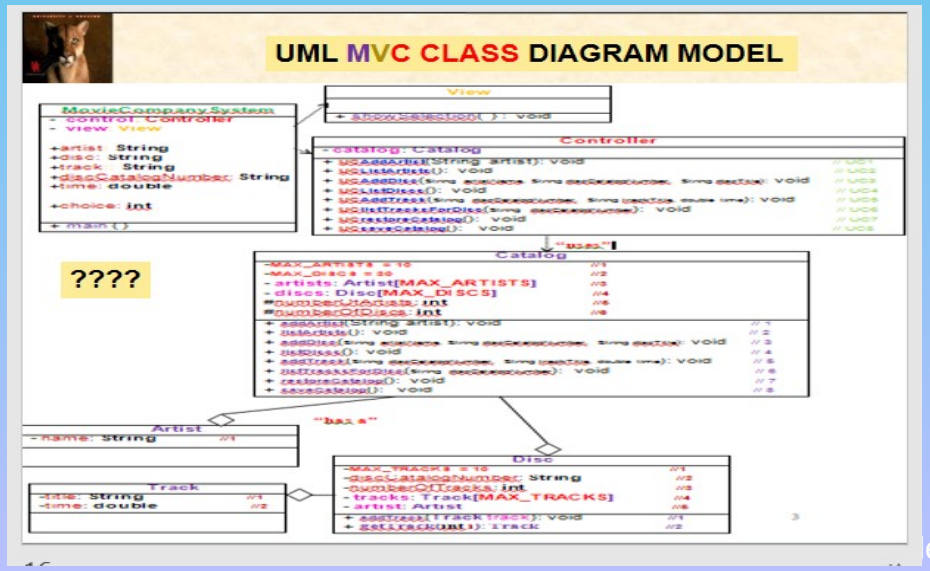
The Object Oriented Paradigm in Perspective

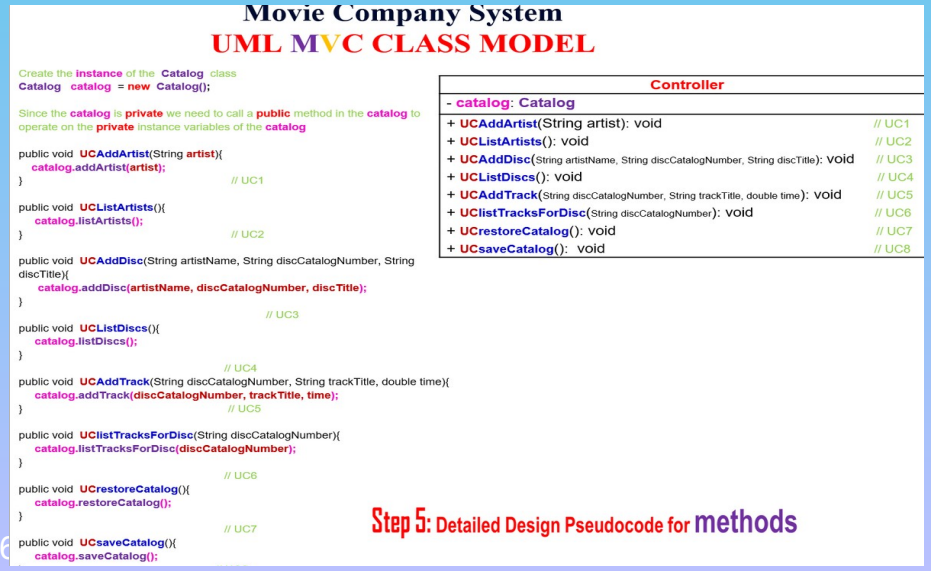
The Object Oriented Paradigm **has to be** used correctly

- All Paradigms are easy to misuse

When used correctly, the Object Oriented Paradigm can solve some (but not all) of the problems of the **Classical** Paradigm

MVC





The Object Oriented Paradigm in Perspective

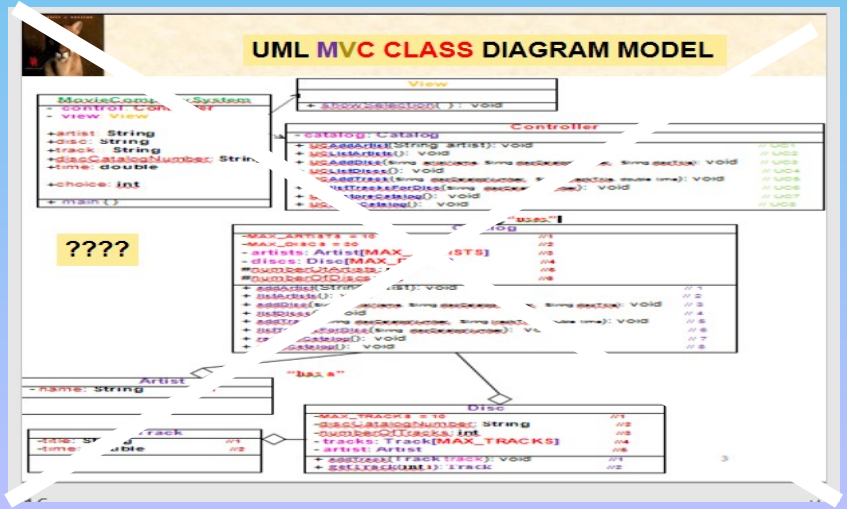
The Object Oriented Paradigm has problems of its own

lack of high **cohesion** and low **coupling**

The Object Oriented Paradigm is the best alternative available today (???)

- However, **it is certain to be superseded** by something better in the future

MVC



A STRUCTURE is stable if **cohesion** is strong & **coupling** is low (L7)

- Source: Stevens, Myers, and Constantine [Stev 74]
- High **Cohesion** allows changes (to one issue) locally
- Low **Coupling** avoids spill-over or so-called ripple effects

Only what is hidden can be changed without risk (L8)

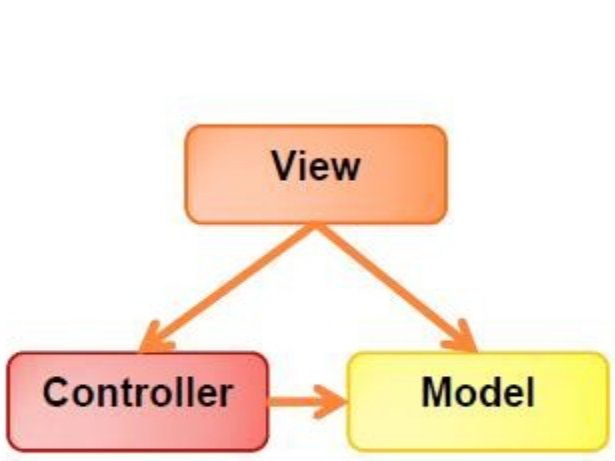
- Source: David Parnas [Parn 72]
- **Information Hiding** applied properly leads to strong Cohesion low Coupling

Laws

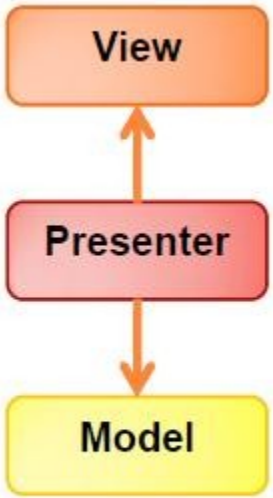
Object **O**riented **P**aradigm reduces **defects** and encourages **Reuse** (L17)

- Source: Ole-Johan Dahl [Dahl 67], Adele Goldberg [Gold 89]
- First languages: Simula 67, Smalltalk, Java
- Based on **Information Hiding** via **Classes** & increased **Reuse** potential

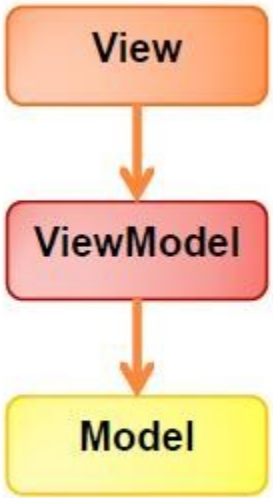
(**MVC MVP MVVM Design Patterns**)



MVC



MVP

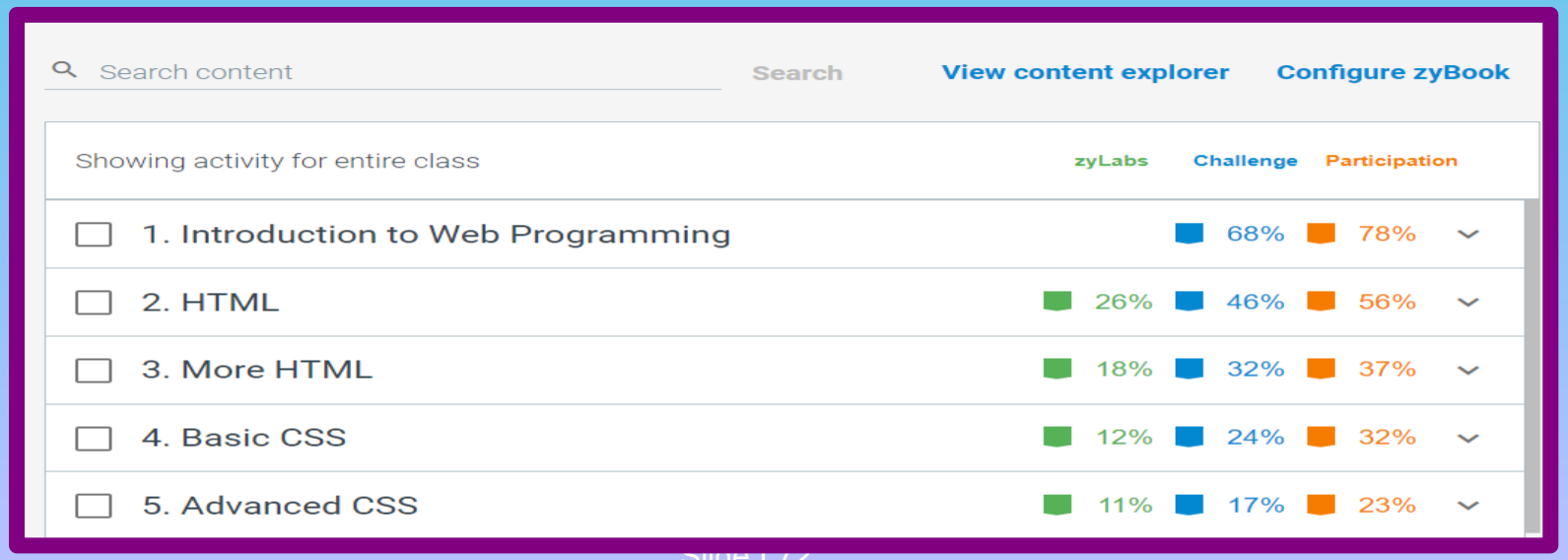


MVVM

At 5:00 PM.

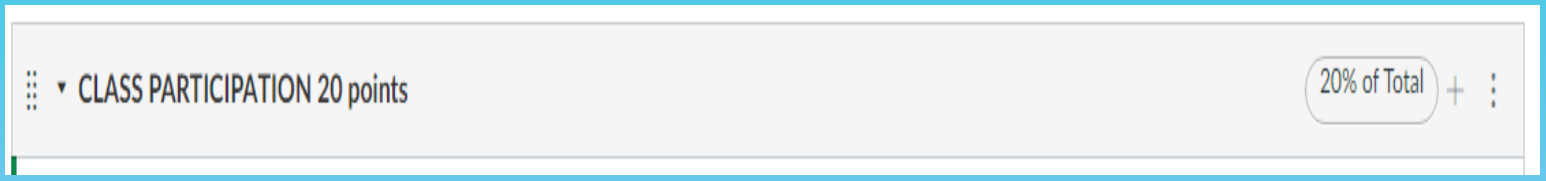
VH, next EXAM 1 Review

09.11.2023 (M 4 to 5:30) (6)		EXAM 1 REVIEW (CANVAS) (ZyBook)	Download ZyBook: Sections 1-5		

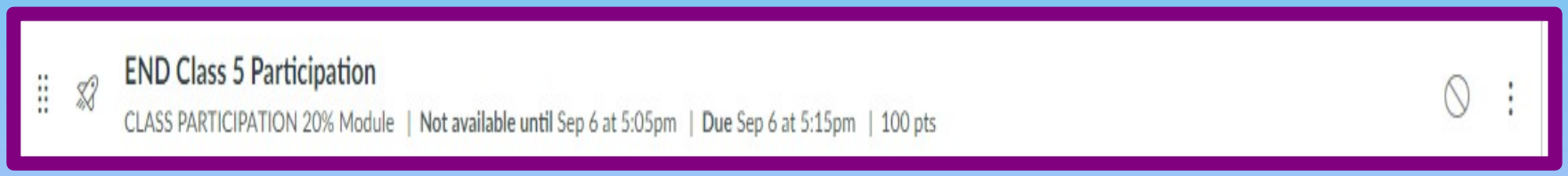


From 5:05 to 5:15 – 10 minutes.

09.06.2023 (W 4 to 5:30) (5)		Lecture 3: Software Development Process			



PASSWORD: I AM IN TEAMS



VH, publish.

At 5:15 PM.

End Class 5

VH, upload Class 5 to CANVAS.

VH, Download Attendance Report
Rename it:
9.06.2023 Attendance Report FINAL