# Chapter 1

- **Software & Software Engineering**

*Slide Set to accompany*

*Software Engineering: A Practitioner's Approach, 7/e*

**by Roger S. Pressman**

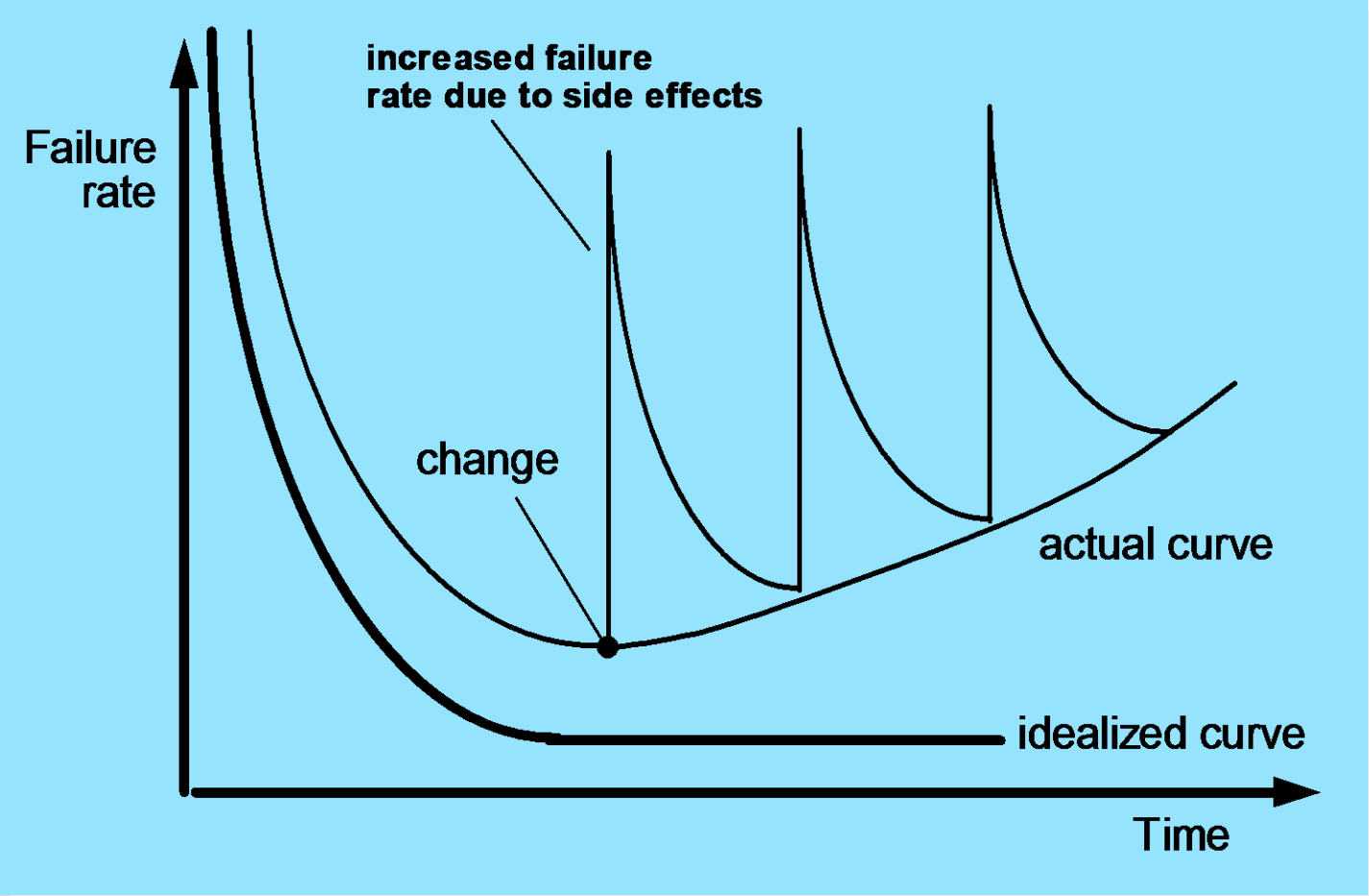
# What is Software?

*Software is: (1) instructions (computer programs) that when executed provide desired features, function, and performance; (2) data structures that enable the programs to adequately manipulate information and (3) documentation that describes the operation and use of the programs.*

# What is Software?

- ***Software is developed or engineered, it is not manufactured in the classical sense.***
- ***Software doesn't "wear out."***
- ***Although the industry is moving toward component-based construction, most software continues to be custom-built.***

# Wear vs. Deterioration



# Software Applications

- system software
- application software
- engineering/scientific software
- embedded software
- product-line software
- WebApps (Web applications)
- AI software

# Software—New Categories

- Open world computing—pervasive, distributed computing
- Ubiquitous computing—wireless networks
- Netsourcing—the Web as a computing engine
- Open source—”free” source code open to the computing community (a blessing, but also a potential curse!)
- Also … (see Chapter 31)
  - Data mining
  - Grid computing
  - Cognitive machines
  - Software for nanotechnologies

# Legacy Software

## *Why must it change?*

- software must be adapted to meet the needs of new computing environments or technology.
- software must be enhanced to implement new business requirements.
- software must be extended to make it interoperable with other more modern systems or databases.
- software must be re-architected to make it viable within a network environment.

# Characteristics of WebApps - I

- **Network intensiveness.** A WebApp resides on a network and must serve the needs of a diverse community of clients.
- **Concurrency.** A large number of users may access the WebApp at one time.
- **Unpredictable load.** The number of users of the WebApp may vary by orders of magnitude from day to day.
- **Performance.** If a WebApp user must wait too long (for access, for server-side processing, for client-side formatting and display), he or she may decide to go elsewhere.
- **Availability.** Although expectation of 100 percent availability is unreasonable, users of popular WebApps often demand access on a “24/7/365” basis.

# Characteristics of WebApps - II

- **Data driven.** The primary function of many WebApps is to use hypermedia to present text, graphics, audio, and video content to the end-user.
- **Content sensitive.** The quality and aesthetic nature of content remains an important determinant of the quality of a WebApp.
- **Continuous evolution.** Unlike conventional application software that evolves over a series of planned, chronologically-spaced releases, Web applications evolve continuously.
- **Immediacy.** Although *immediacy*—the compelling need to get software to market quickly—is a characteristic of many application domains, WebApps often exhibit a time to market that can be a matter of a few days or weeks.
- **Security.** Because WebApps are available via network access, it is difficult, if not impossible, to limit the population of end-users who may access the application.
- **Aesthetics.** An undeniable part of the appeal of a WebApp is its look and feel.

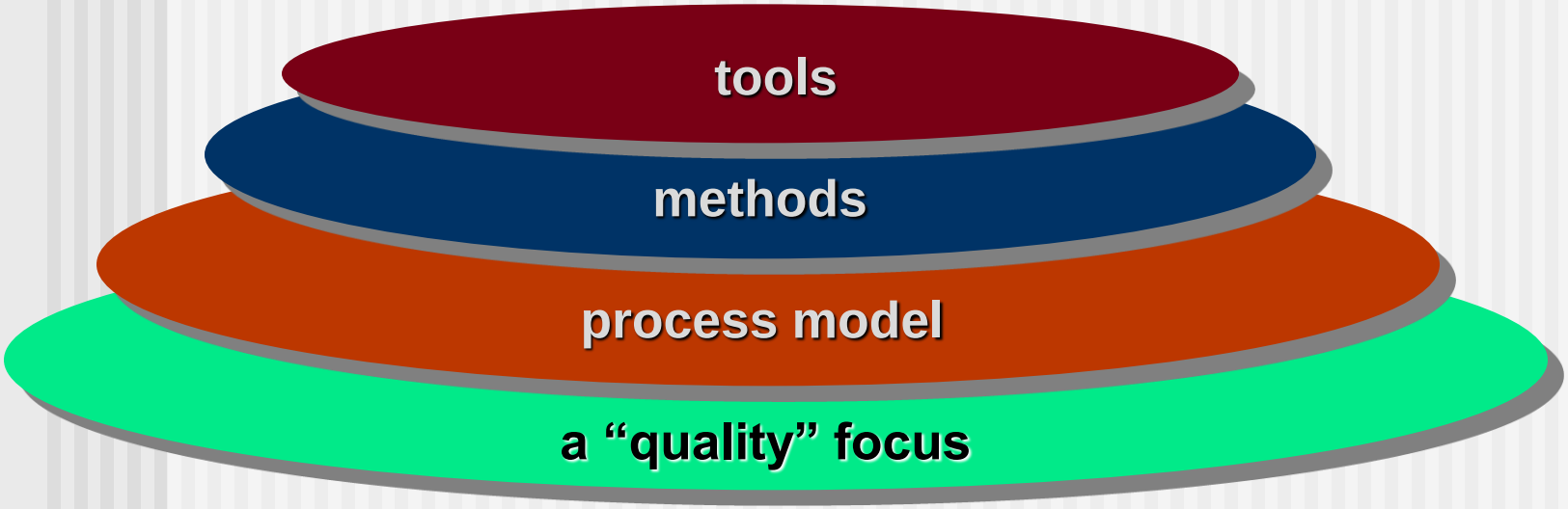
# Software Engineering

- Some realities:
  - *a concerted effort should be made to understand the problem before a software solution is developed*
  - *design becomes a pivotal activity*
  - *software should exhibit high quality*
  - *software should be maintainable*
- The seminal definition:
  - *[Software engineering is] the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines.*

# Software Engineering

- The IEEE definition:
  - *Software Engineering: (1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. (2) The study of approaches as in (1).*

# A Layered Technology




***Software Engineering***

# A Process Framework

**Process framework**

**Framework activities**

work tasks
work products
milestones & deliverables
QA checkpoints

**Umbrella Activities**

# Framework Activities

- Communication
- Planning
- Modeling
  - Analysis of requirements
  - Design
- Construction
  - Code generation
  - Testing
- Deployment

# Umbrella Activities

- Software project management
- Formal technical reviews
- Software quality assurance
- Software configuration management
- Work product preparation and production
- Reusability management
- Measurement
- Risk management

# Adapting a Process Model

- the overall flow of activities, actions, and tasks and the interdependencies among them
- the degree to which actions and tasks are defined within each framework activity
- the degree to which work products are identified and required
- the manner which quality assurance activities are applied
- the manner in which project tracking and control activities are applied
- the overall degree of detail and rigor with which the process is described
- the degree to which the customer and other stakeholders are involved with the project
- the level of autonomy given to the software team
- the degree to which team organization and roles are prescribed

# The Essence of Practice

- Polya suggests:
  1. *Understand the problem* (communication and analysis).
  2. *Plan a solution* (modeling and software design).
  3. *Carry out the plan* (code generation).
  4. *Examine the result for accuracy* (testing and quality assurance).

# Understand the Problem

- *Who has a stake in the solution to the problem?* That is, who are the stakeholders?
- *What are the unknowns?* What data, functions, and features are required to properly solve the problem?
- *Can the problem be compartmentalized?* Is it possible to represent smaller problems that may be easier to understand?
- *Can the problem be represented graphically?* Can an analysis model be created?

# Plan the Solution

- *Have you seen similar problems before?* Are there patterns that are recognizable in a potential solution? Is there existing software that implements the data, functions, and features that are required?
- *Has a similar problem been solved?* If so, are elements of the solution reusable?
- *Can subproblems be defined?* If so, are solutions readily apparent for the subproblems?
- *Can you represent a solution in a manner that leads to effective implementation?* Can a design model be created?

# Carry Out the Plan

- *Does the solution conform to the plan?* Is source code traceable to the design model?
- *Is each component part of the solution provably correct?* Has the design and code been reviewed, or better, have correctness proofs been applied to algorithm?

# Examine the Result

- *Is it possible to test each component part of the solution?* Has a reasonable testing strategy been implemented?
- *Does the solution produce results that conform to the data, functions, and features that are required?* Has the software been validated against all stakeholder requirements?

# Hooker's General Principles

- 1: *The Reason It All Exists*
- 2: *KISS (Keep It Simple, Stupid!)*
- 3: *Maintain the Vision*
- 4: *What You Produce, Others Will Consume*
- 5: *Be Open to the Future*
- 6: *Plan Ahead for Reuse*
- 7*: Think!*

# Software Myths

- Affect managers, customers (and other non-technical stakeholders) and practitioners
- Are believable because they often have elements of truth,

*but …*

- Invariably lead to bad decisions,

*therefore …*

- Insist on reality as you navigate your way through software engineering

# How It all Starts

- *SafeHome:*
  - Every software project is precipitated by some business need—
    - the need to correct a defect in an existing application;
    - the need to the need to adapt a ‘legacy system’ to a changing business environment;
    - the need to extend the functions and features of an existing application, or
    - the need to create a new product, service, or system.

# Chapter 2

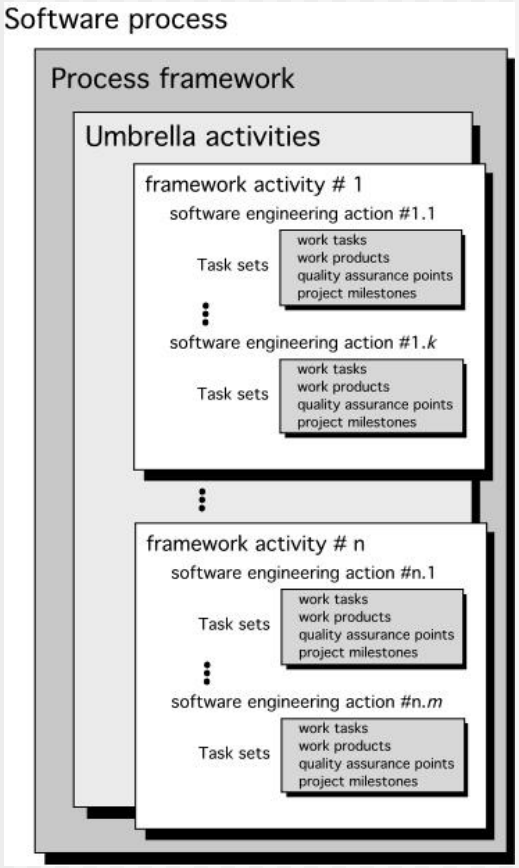
- **Process Models**

*Slide Set to accompany*

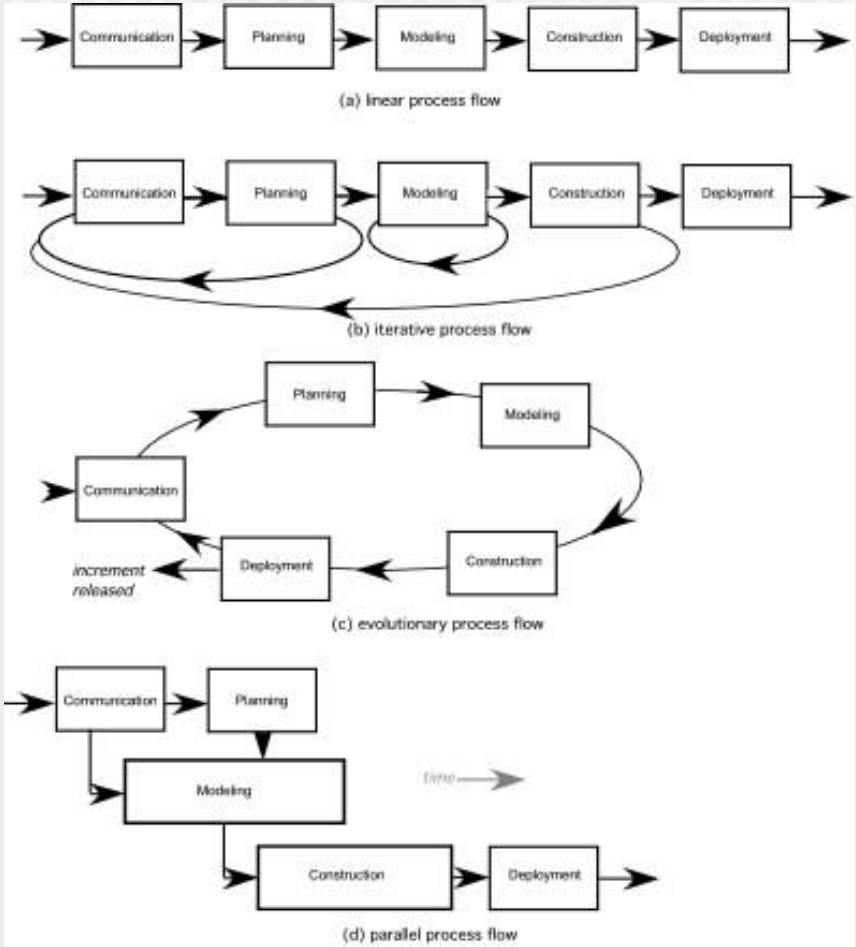
*Software Engineering: A Practitioner's Approach, 7/e*

**by Roger S. Pressman**

# A Generic Process Model



# Process Flow



# Identifying a Task Set

- A task set defines the actual work to be done to accomplish the objectives of a software engineering action.
  - A list of the task to be accomplished
  - A list of the work products to be produced
  - A list of the quality assurance filters to be applied

# Process Patterns

- A *process pattern*
  - describes a process-related problem that is encountered during software engineering work,
  - identifies the environment in which the problem has been encountered, and
  - suggests one or more proven solutions to the problem.
- Stated in more general terms, a process pattern provides you with a *template* [Amb98]—a consistent method for describing problem solutions within the context of the software process.

# Process Pattern Types

- *Stage patterns*—defines a problem associated with a framework activity for the process.
- *Task patterns*—defines a problem associated with a software engineering action or work task and relevant to successful software engineering practice
- *Phase patterns*—define the sequence of framework activities that occur with the process, even when the overall flow of activities is iterative in nature.

# Process Assessment and Improvement

- **Standard CMMI Assessment Method for Process Improvement (SCAMPI)** — provides a five step process assessment model that incorporates five phases: initiating, diagnosing, establishing, acting and learning.
- **CMM-Based Appraisal for Internal Process Improvement (CBA IPI)**—provides a diagnostic technique for assessing the relative maturity of a software organization; uses the SEI CMM as the basis for the assessment [Dun01]
- **SPICE—The SPICE (ISO/IEC15504)** standard defines a set of requirements for software process assessment. The intent of the standard is to assist organizations in developing an objective evaluation of the efficacy of any defined software process. [ISO08]
- **ISO 9001:2000 for Software—**a generic standard that applies to any organization that wants to improve the overall quality of the products, systems, or services that it provides. Therefore, the standard is directly applicable to software organizations and companies. [Ant06]

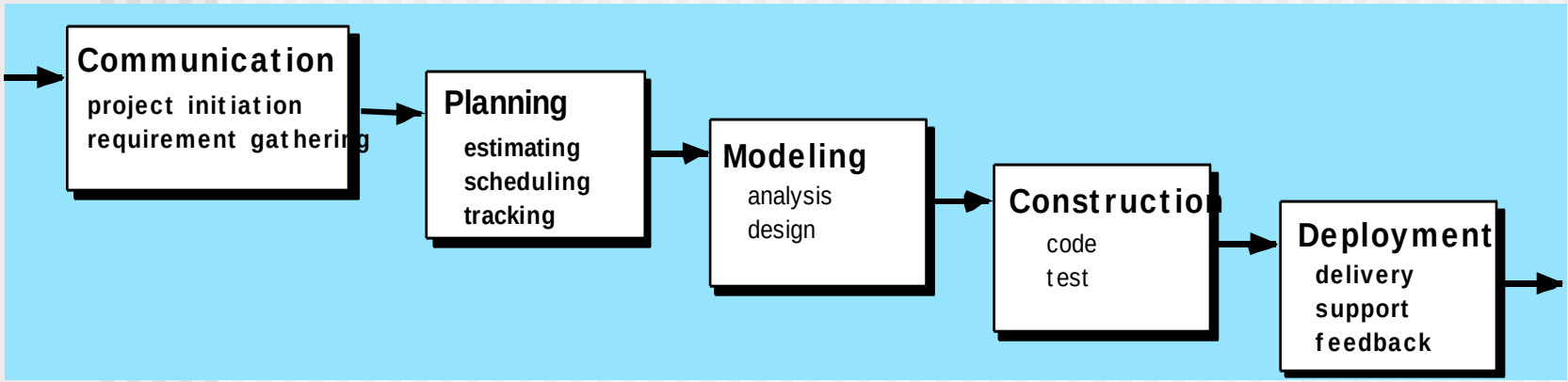
# Prescriptive Models

- Prescriptive process models advocate an orderly approach to software engineering

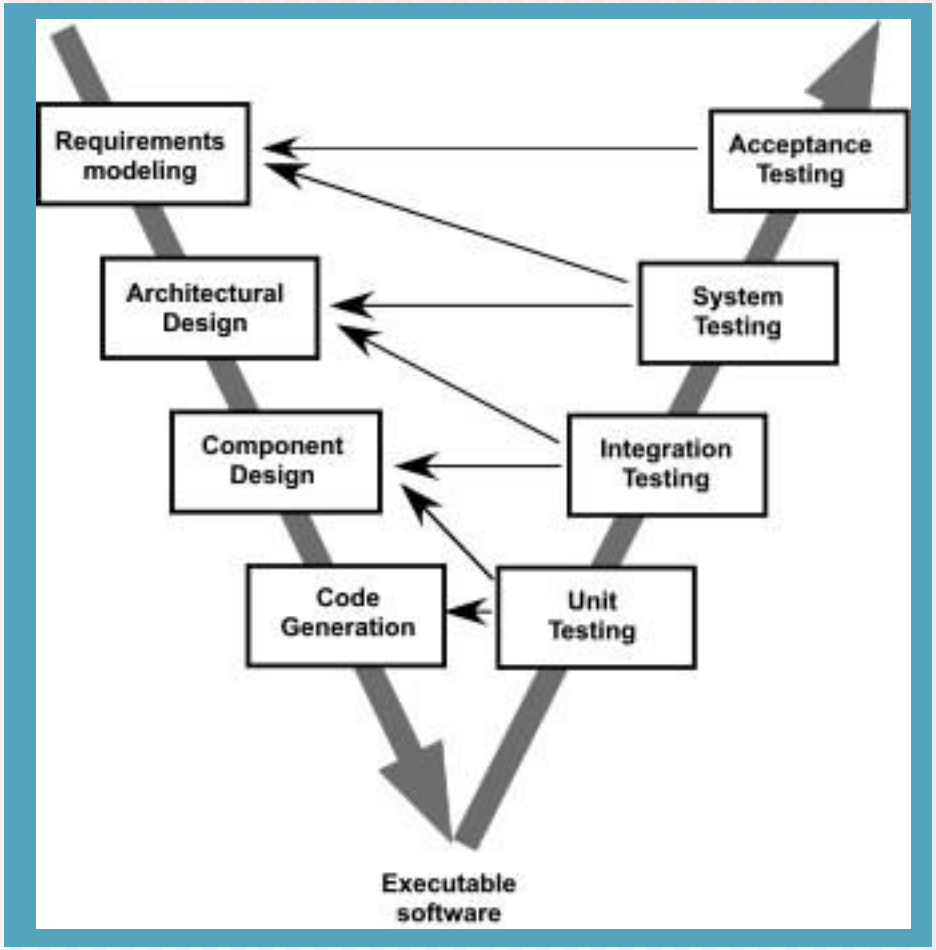
*That leads to a few questions …*

- If prescriptive process models strive for structure and order, are they inappropriate for a software world that thrives on change?
- Yet, if we reject traditional process models (and the order they imply) and replace them with something less structured, do we make it impossible to achieve coordination and coherence in software work?

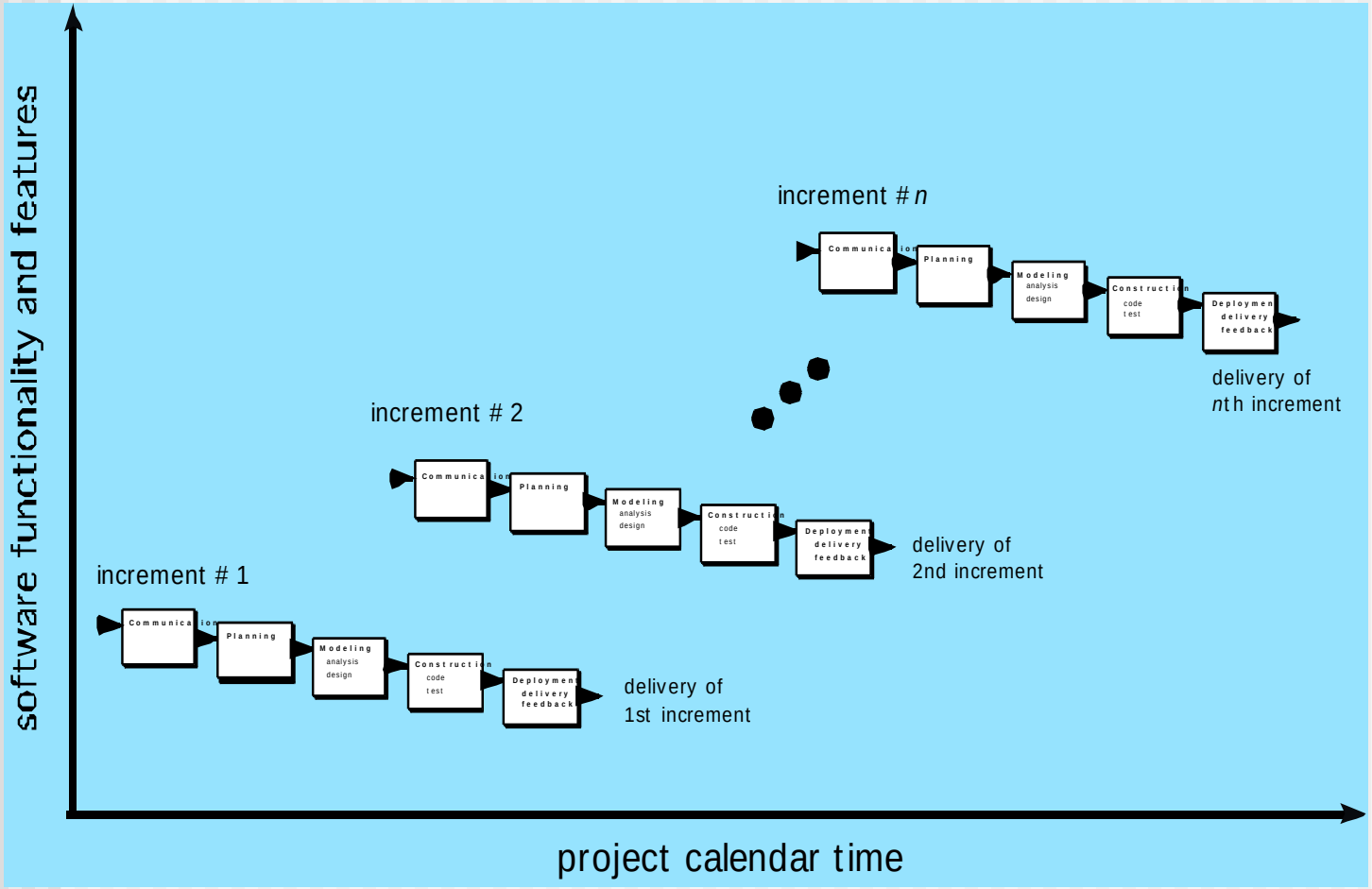
# The Waterfall Model



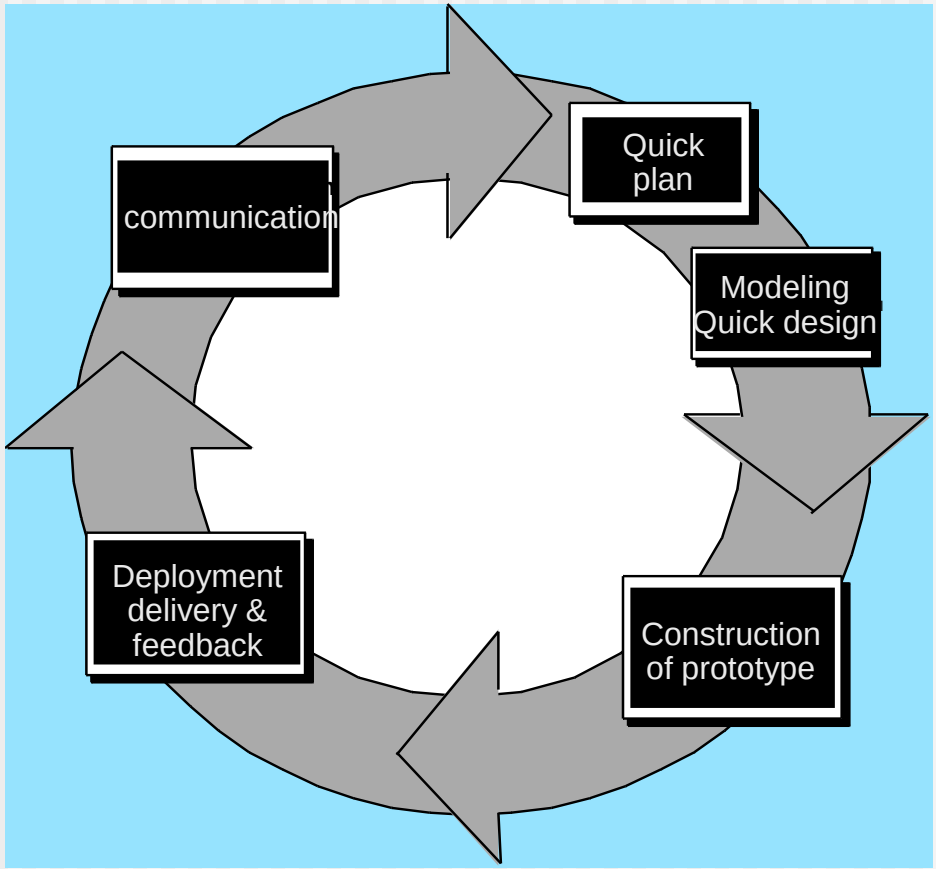
# The V-Model



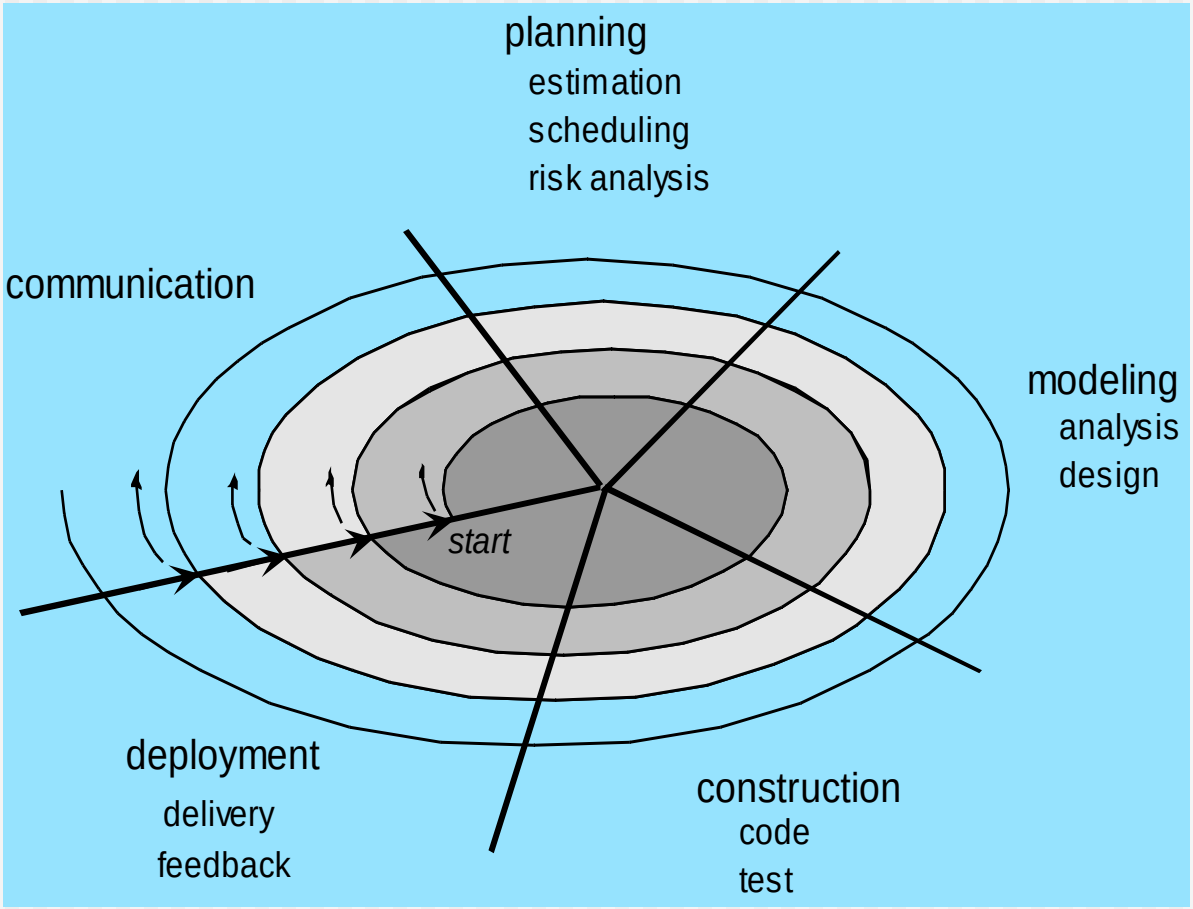
# The Incremental Model



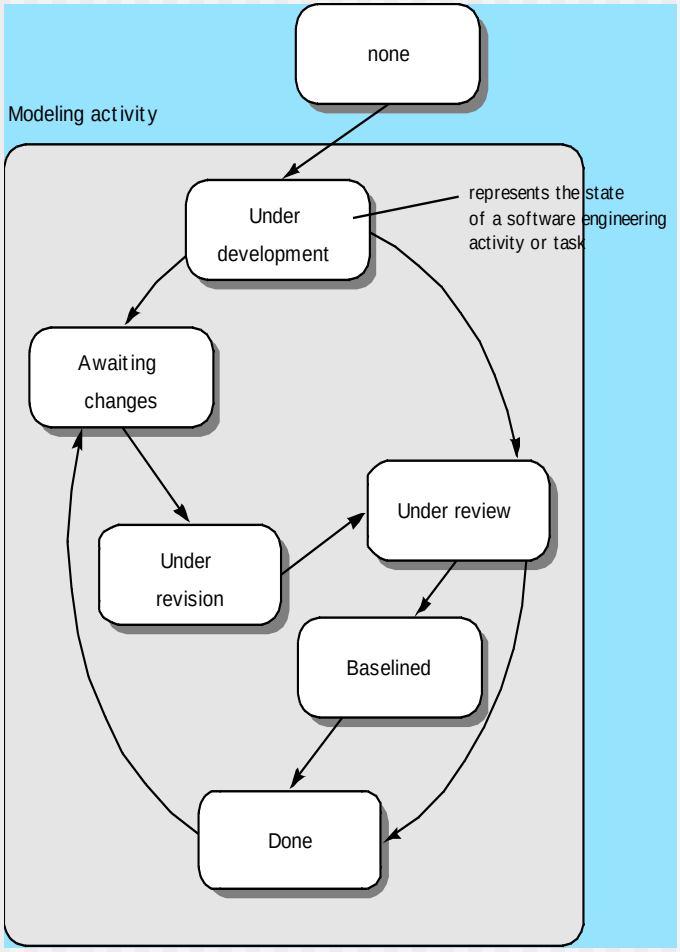
# Evolutionary Models: Prototyping



# Evolutionary Models: The Spiral



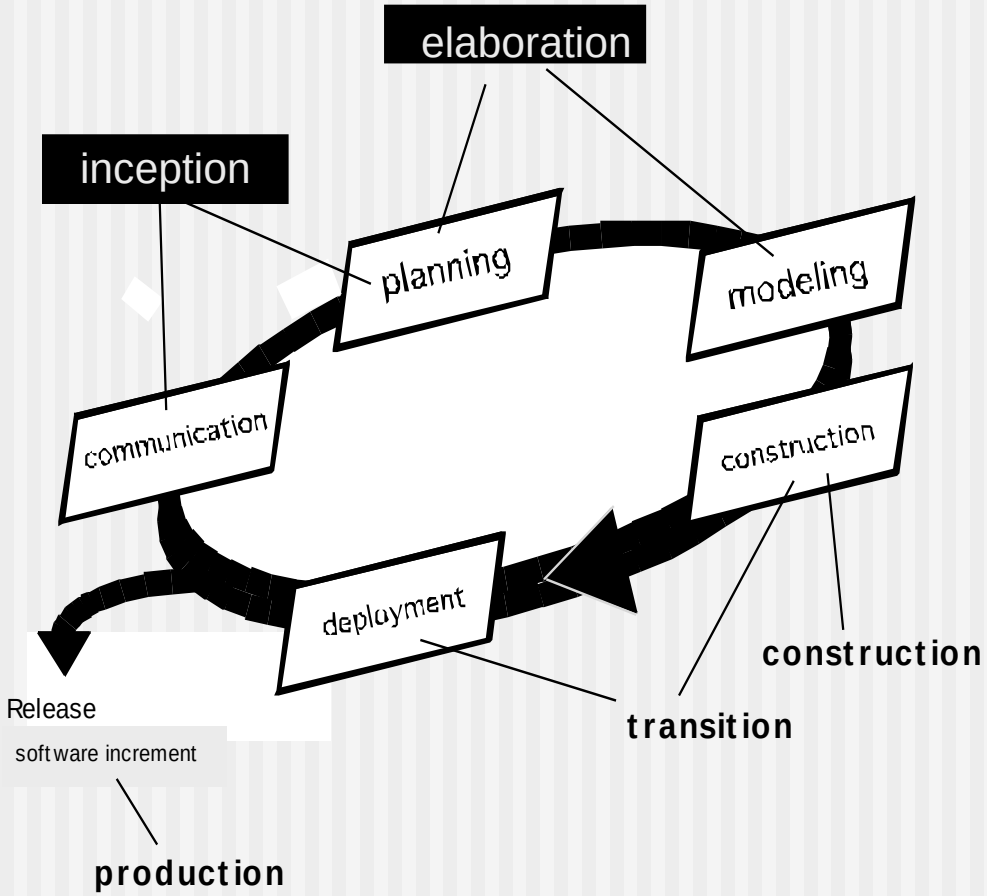
# Evolutionary Models: Concurrent



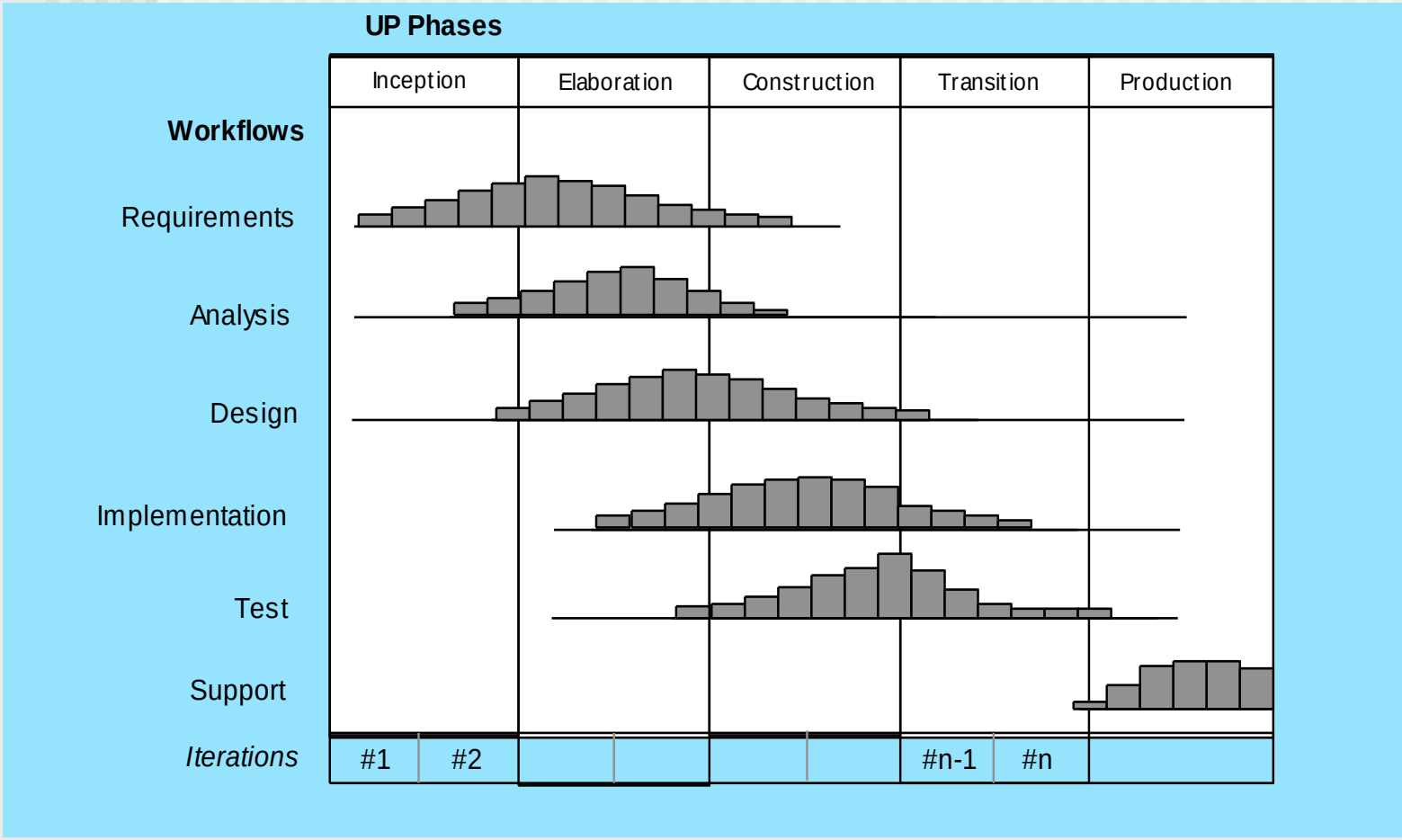
# Still Other Process Models

- Component based development—the process to apply when reuse is a development objective
- Formal methods—emphasizes the mathematical specification of requirements
- AOSD—provides a process and methodological approach for defining, specifying, designing, and constructing *aspects*
- Unified Process—a "use-case driven, architecture-centric, iterative and incremental" software process closely aligned with the Unified Modeling Language (UML)

# The Unified Process (UP)



# UP Phases



# UP Work Products

**Inception phase**

Vision document
Initial use-case model
Initial project glossary
Initial business case
Initial risk assessment.
Project plan,
  phases and iterations.
Business model,
  if necessary.
One or more prototypes

**Elaboration phase**

Use-case model
Supplementary requirements
  including non-functional
Analysis model
Software architecture
  Description.
Executable architectural
  prototype.
Preliminary design model
Revised risk list
Project plan including
  iteration plan
  adapted workflows
  milestones
  technical work products
Preliminary user manual

**Construction phase**

Design model
Software components
Integrated software
  increment
Test plan and procedure
Test cases
Support documentation
  user manuals
  installation manuals
  description of current
    increment

**Transition phase**

Delivered software increment
Beta test reports
General user feedback

# Personal Software Process (PSP)

- **Planning.** This activity isolates requirements and develops both size and resource estimates. In addition, a defect estimate (the number of defects projected for the work) is made. All metrics are recorded on worksheets or templates. Finally, development tasks are identified and a project schedule is created.
- **High-level design.** External specifications for each component to be constructed are developed and a component design is created. Prototypes are built when uncertainty exists. All issues are recorded and tracked.
- **High-level design review.** Formal verification methods (Chapter 21) are applied to uncover errors in the design. Metrics are maintained for all important tasks and work results.
- **Development.** The component level design is refined and reviewed. Code is generated, reviewed, compiled, and tested. Metrics are maintained for all important tasks and work results.
- **Postmortem.** Using the measures and metrics collected (this is a substantial amount of data that should be analyzed statistically), the effectiveness of the process is determined. Measures and metrics should provide guidance for modifying the process to improve its effectiveness.

# Team Software Process (TSP)

- Build self-directed teams that plan and track their work, establish goals, and own their processes and plans. These can be pure software teams or integrated product teams (IPT) of three to about 20 engineers.
- Show managers how to coach and motivate their teams and how to help them sustain peak performance.
- Accelerate software process improvement by making CMM Level 5 behavior normal and expected.
  - The Capability Maturity Model (CMM), a measure of the effectiveness of a software process, is discussed in Chapter 30.
- Provide improvement guidance to high-maturity organizations.
- Facilitate university teaching of industrial-grade team skills.

# REST

Feliz Gouveia
fribeiro@ufp.edu.pt
UFP, 2014

# What is REST ?

- REpresentational State Transfer
- A term and architectural approach for the web coined in Roy Fielding's thesis (2000)
- It proposes that accesses to a web site be represented uniformly
- **State Transfer** because a user goes from one page to another by following links
- It helps to design "good" URIs

# REST

- Uses a combination of URI and HTTP methods
  - GET means “read”
  - PUT means “edit”
  - DELETE means “delete”
  - POST means “create”

# What is good URI design ?

- Uniform and consistent
  - Is /listproduct.php?id=2 or /show_product.jsp?pid=5 consistent?
- Readable
- Use nouns, not verbs
- Meaningful: users can guess what the resource is
- Stackable: remove a piece from the url and should also get a resource
- Query args (after the ?) should be used for queries

# Tips for good URIs

- Do not mix lower and upper case (prefer lower case)
- Use hyphens rather than underscores (more readable, but that’s your choice)
- Do not use extensions (php, aspx, jsp, pl, py,…), be technology independent
- Avoid spaces and “illegal” URI characters, because they get URL-encoded (with lots of %)
- Do not change an URI over time

# Some examples

- /students  GET lists students
- /students  POST creates a student
- /students/25  GET lists, PUT edits it, DELETE
- /students/25/courses  GET
- /students/25/courses/3  GET, PUT, DELETE
- /students?name=john  GET does a search

- The same URI can have different operations depending on the HTTP method

# More examples

- Getting forms to edit/create
  - GET / students/25/edit
  - GET / students/new
- Alternative way
  - GET /students/form/33/data/25 the edit form
  - GET /students/form/33 same, but empty
- Sorting data
  - GET /students?sort=ascending&page=2

# Specifying data formats

- When the client needs a specific format it must specify it
- /students/23.json
- /students/23.xml
- Useful for web services, Ajax calls
- A default format can be HTML

# Responses

- Response to a REST request should specify a HTTP response code
  - 200 OK
  - …

# HTTP basics

Programming Web applications

Feliz Gouveia

fribeiro@ufp.edu.pt

# Timeline

- 1975: ARPANET
- 1983: TCP/IP is published
- 1990: Internet  replaces ARPANET
- 1991: HTTP (Hypertext Transfer Protocol) is invented
  - At CERN (Geneva), from the need to share tons of information between several groups

# HTTP: Hyper Text Transfer Protocol

- First defined in 1991 (version 1.1 in 1999)
  - www.ietf.org/rfc/rfc1945.txt
- Client – server protocol
- Uses TCP/IP
  - Port 80
- Basic sequence:
  - Client sends message
  - Server answers
  - Server does not keep state between requests

# HTTP

- Basic idea of HTTP is to apply a set of methods to resources
- Resources are identified by a URI (Uniform Resource Identifier)
- A URI can be:
  - URL (Uniform Resource Locator)
  - URN (Uniform Resource Name)
- Most of the time a URI is what we call a web address

# URI (Universal Resource Identifier)

- Is of the form:
  - schema : ( absoluteURI | relativeURI ) [ "#" fragment ]
- It specifies an address where to find a resource (URL) or a name of a resource (URN)

# URN

- Defined in RFC 2141
- A URN does not define how to find a resource
  - We may need a plugin to do that
- A URN specifies a name, for example:
  - urn:isbn:123456789
  - swift:+351-223456789
- List of registered URNs
  - http://www.iana.org/assignments/urn-namespaces/urn-namespaces.xhtml

# URL

- Defined in RFC 1738
- Can be absoute or relative (to some base path)
- Example:
  - http://www.ufp.pt
  - http://www.ufp.pt:80
  - http://uniformjs.com/#intro
  - /js/stylesheet.js (relative URL)
  - http://example.pt/pub?**q**=2&**a**=23
    - Parameters **q** and **a**, separated by **&**
    - The part after **?** is the query string

# URL

- Some characters are not allowed in the URL, so they must be “URL encoded” (also known as percent-encoded): %hex hex
- Try http://meyerweb.com/eric/tools/dencoder/
- Common cases are spaces (%20), slashes (%2F), question marks (%3F)

# HTTP Control parameters

- Version: the protocol version (current 1.1)
- Content coding: coding applied to the resource
  - deflate/zlib
  - gzip
  - compress
- Compression reduces the size of traffic, and some HTTP servers do it automatically for certain types of files
- More next

# Media types

- Allow the server to express how the resource should be processed
  - For example a PDF file should be opened in a PDF viewer
- Media types are registered at:
  - http://www.iana.org/assignments/media-types/media-types.xhtml
- An unknown media type gets a fairly general treatment

# Character encoding

- Specifies how characters are encoded for transmission
- Initially ISO-8859-1 was most used
  - Single byte encoding, limited set of chars (191)
- We should  now (always) specify UTF-8
  - One to four byte encoding
- Source of frequent headaches
  - Some software seems to stick to ISO encoding
  - As streams of chars go through several applications, they can suffer unexpected transformations

# HTTP requests

- Request are sent by the client (user agent)
- A method:
  - GET, POST, HEAD
- A resource identified by a URI
- A head
- A body

- A GET is the result of clicking a hyperlink or writting the address on a browser
- A POST is the result of submitting a form

# The methods

- GET is expected to be read-only, has no effect on the state of the server
  - Simply: GET is a READ
- POST is expected to change the state of the server
  - Simply: POST is a WRITE
  - Try to resubmit a form; most often the browser will warn that you are resubmiting
- HEAD is a GET but the server only returns headers (no body)
  - Useful to test the modification date of a resource and avoid getting it if not newer than cached version

# HTTP responses

- Sent by the server, they specify:
  - A code (see next)
  - A header
  - A body

# Response code families (RFC 1945)

- 1xx: Informational - Not used, but reserved for future use
- 2xx: Success - The action was successfully received, understood, and accepted
- 3xx: Redirection - Further action must be taken in order to complete the request
- 4xx: Client Error - The request contains bad syntax or cannot be fulfilled
- 5xx: Server Error - The server failed to fulfill an apparently valid request

# Example

- request:
  - GET http://www.w3.org/pub/WWW/info.html
  - Accept: text/html
  - If-Modified-Since: Saturday, 10-July-2010 10:00:00 GMT
  - User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-GB; rv:1.9.2.10) Gecko/20100914 Firefox/3.6.10 ( .NET CLR 3.5.30729)
- response:
  - "200"  ; OK

# User agent caching

- The user agent (typically a browser) keeps a local copy of resources in cache and avoids downloading them from the server
- Static content (like image files, scripts, stylesheets) is cached
- Dynamic content is fetched from the server each time it is needed
- The server can also specify if content is to be cached or not but the browsers typically ignore

# Other optimization

- Manage the number of requests and the size of requests
- Besides caching, optimization includes resource size and rendering, and images
- Good reading at:
  - https://developers.google.com/web/fundamentals/performance/optimizing-content-efficiency/

# Recommended tools

- For Firefox
  - *Firebug* (http://getfirebug.com/)
  - *HTML Validator* / Tidy
  - Optionaly *Live HTTP Headers*
- In Chrome and IE9+ use their Developper Tools
- A text editor (Notepad++)
- An IDE (Eclipse, Netbeans,…)

# For more information

- W3C: the web standards organization
  - http://www.w3.org
- IETF: The Internet Engineering Task Force
  - http://www.ietf.org/
  - To search for RFC (Request for Comments) use http://www.ietf.org/download/rfc-index.txt
- IANA: for DNS
  - http://www.iana.org/

# UML

Feliz Gouveia
UFP

# Introducion

- Unified Modeling Language
- International standard for software analysis and design
- Currently a huge standard, covering most of the documentation needs
- 14 UML diagram types

# UML diagrams

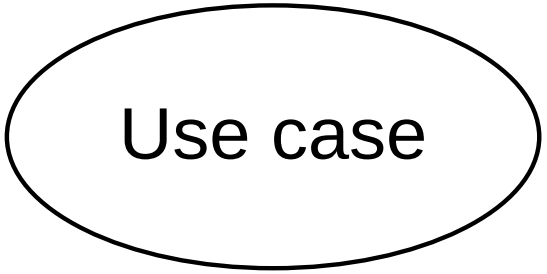
- Class diagram
- Component diagram
- Deployment diagram
- Object diagram
- Package diagram
- Profile diagram
- Composite structure diagram
- Use case diagram
- Activity diagram
- State machine diagram
- Sequence diagram
- Communication diagram
- Interaction overview diagram
- Timing diagram

# Use case diagrams

- Are used during requirements elicitation and analysis as a graphical means of representing the functional requirements of the system.
- Use cases are very helpful for writing acceptance test cases.
- Consist of 4 objects:
  - Use cases
  - Actors
  - System
  - Package

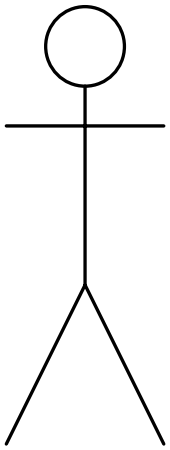
# Use case diagrams: use case

- A use case typically represents a major piece of functionality that is complete from beginning to end.
- Represents a function or an action within the system.



# Use case diagrams: actors

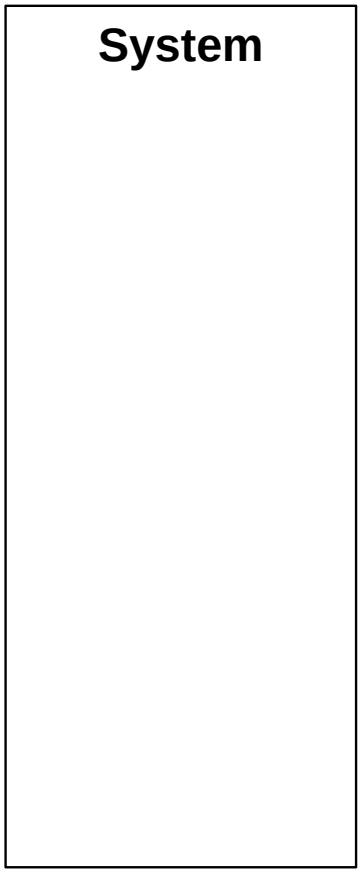
- An actor represents whoever or whatever (person, machine, or other) interacts with the system.
- The actor is not part of the system itself and represents anyone or anything that must interact with the system to:
  - Input information to the system;
  - Receive information from the system; or
  - Both input information to and receive information from the system.



Actor

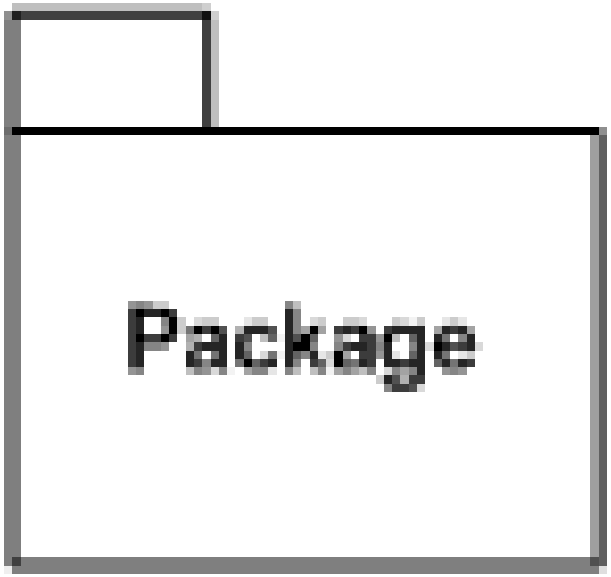
# Use case diagrams: system

- System is used to **define the scope of the use case** and drawn as a rectangle.



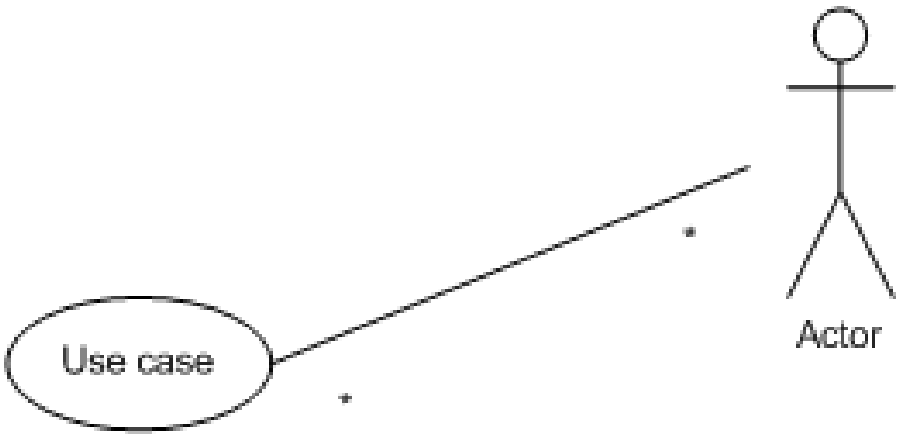
# Use case diagrams: package

- Package is another optional element that is extremely useful in complex diagrams. Similar to class diagrams, packages are **used to group together use cases**.



# Use cases: notation

- Actor comunicates with the system



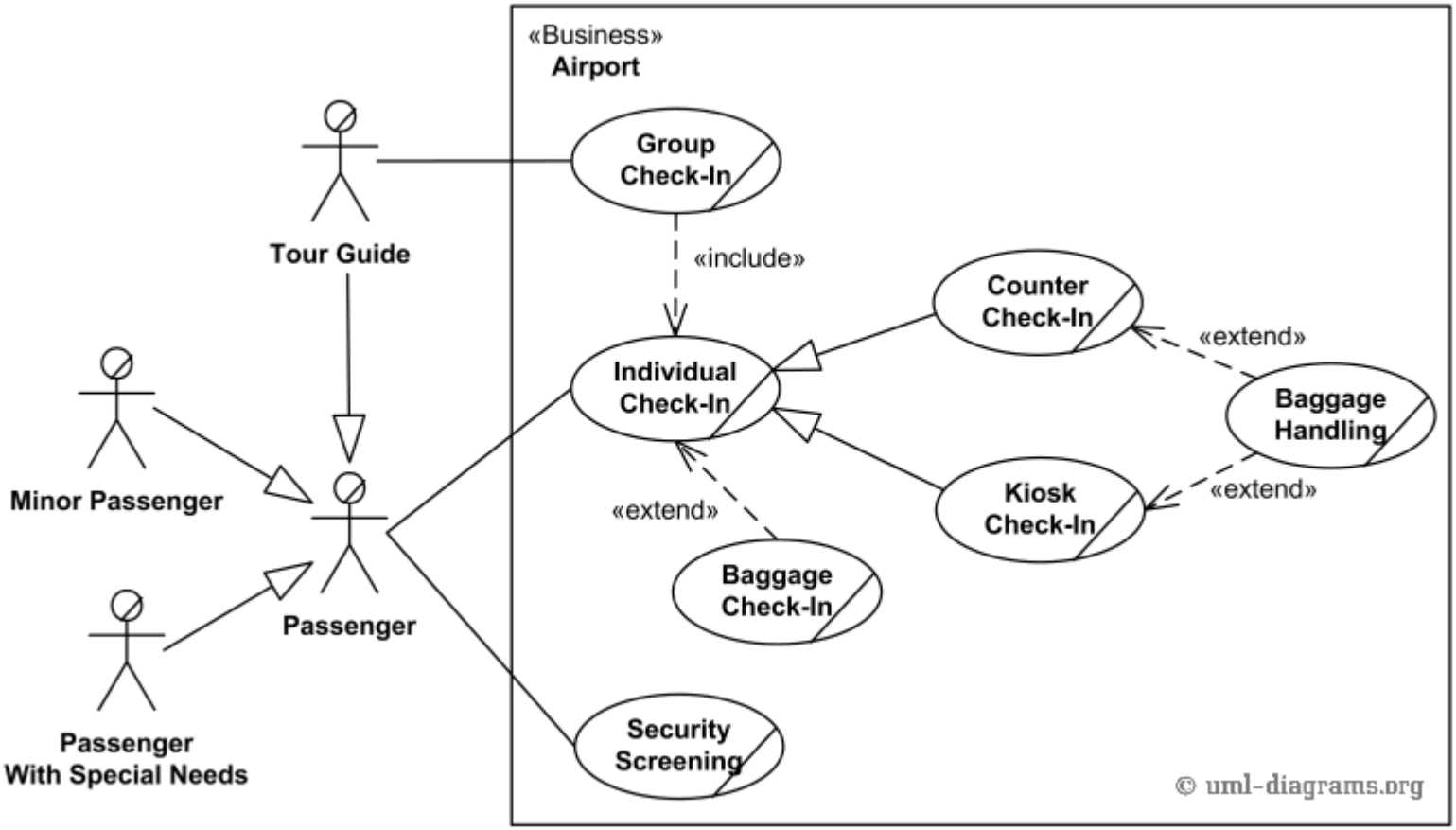
# Use case diagrams: relationships

- There are five types of relationships in a use case diagram. They are:
  - Association between an actor and a use case
  - Generalization of an actor
  - Extend relationship between two use cases
  - Include relationship between two use cases
  - Generalization of a use case

# Use case: relationships

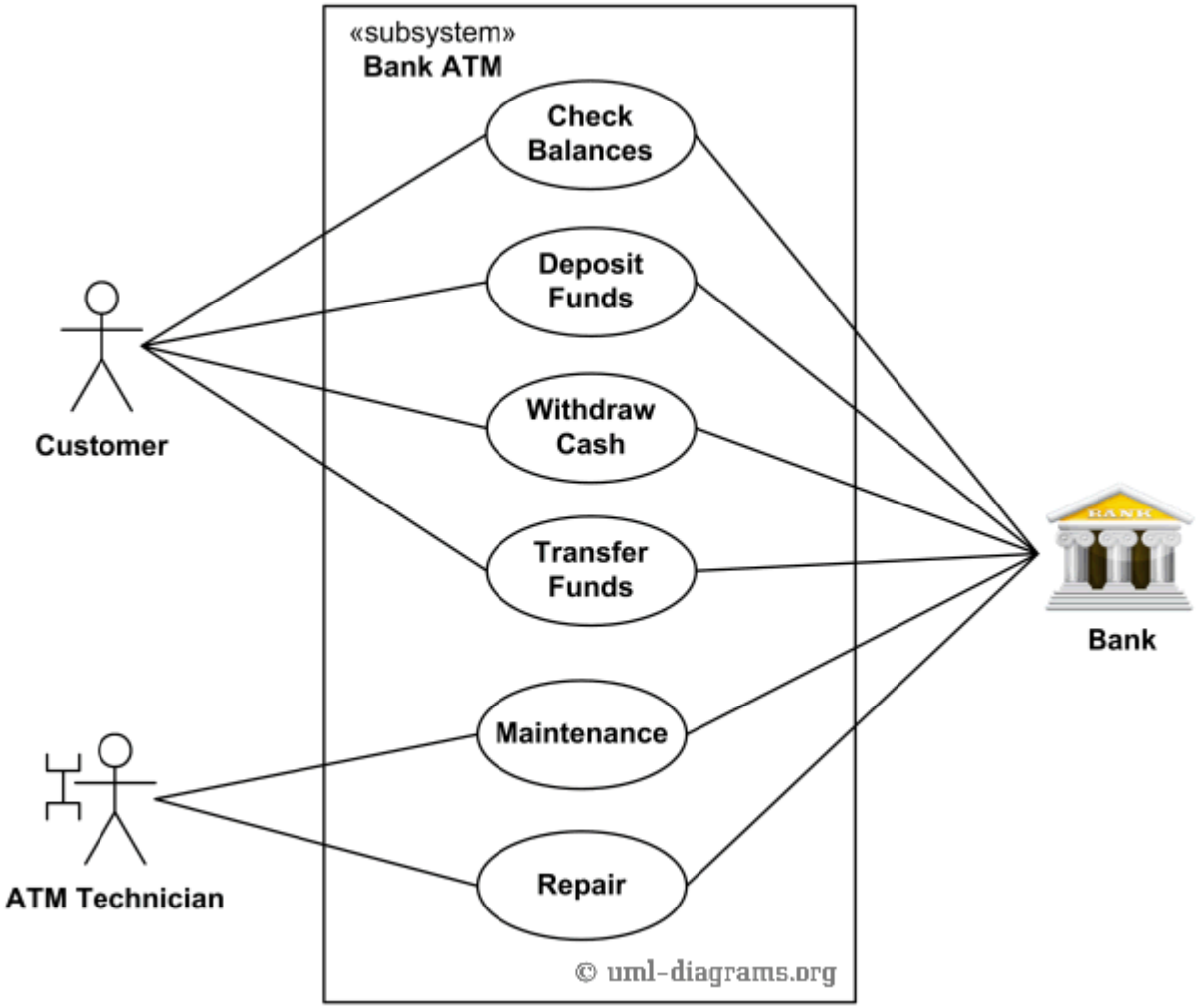
- A use case can “include” another use case; this is useful to avoid repeating the description of a behavior; e.g. if several use cases require a login, it’s best to include the login use case;
- A use case can “extend” another use case; this can be used for variations on a use case, that only happen under certain circunstances;
- Note that an include use case is always completed, but an exclude use case is sometimes completed

# Use cases: example

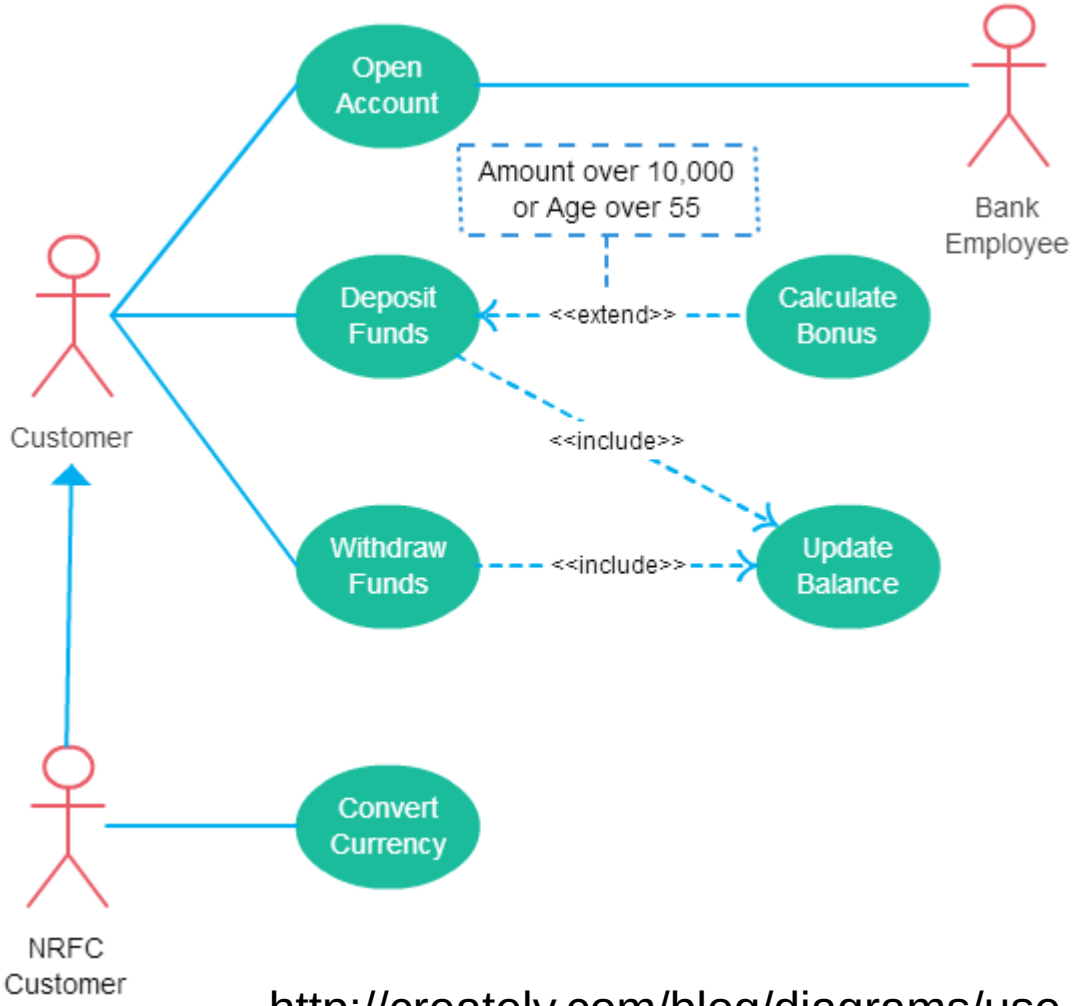



http://www.uml-diagrams.org/

# Use cases: example



# Use cases: example




http://creately.com/blog/diagrams/use-case-diagram-relationships/

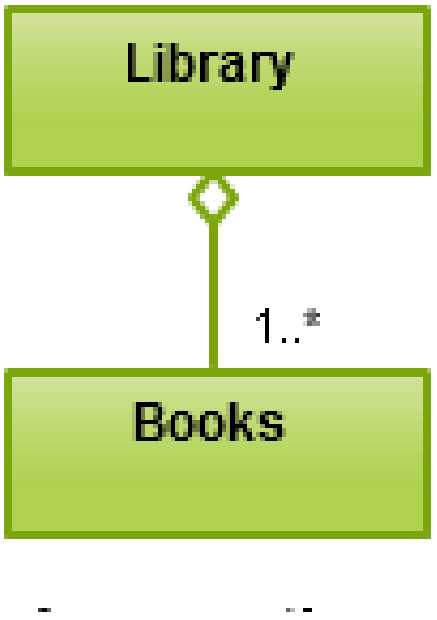
# Class diagram

- Class diagrams are used in both the analysis and the design phases.
- They provide detailed information about the structure and the behavior of the classes
- There are two main types of relationships between classes:
  - Inheritance
  - Association

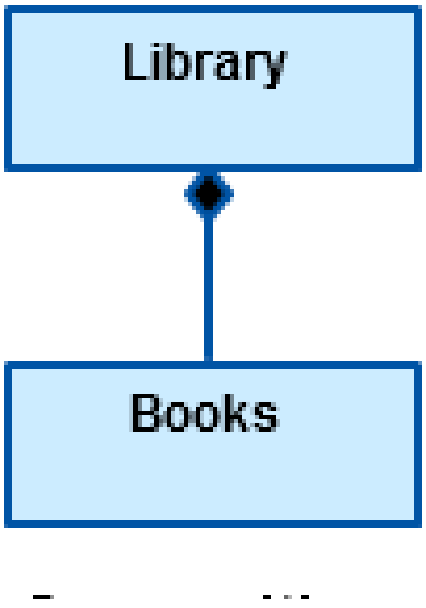
# Class relationships

- Inheritance: one class is more general than others
- Association:
  - Aggregation: one object is part of another object but their lifetimes are independent (a department has employees)
  - Composition: one object is composed of other objects (an invoice is composed of invoice lines) and those can only exist as long as the container object exists
  - Association has multiplicity

# Class relationships



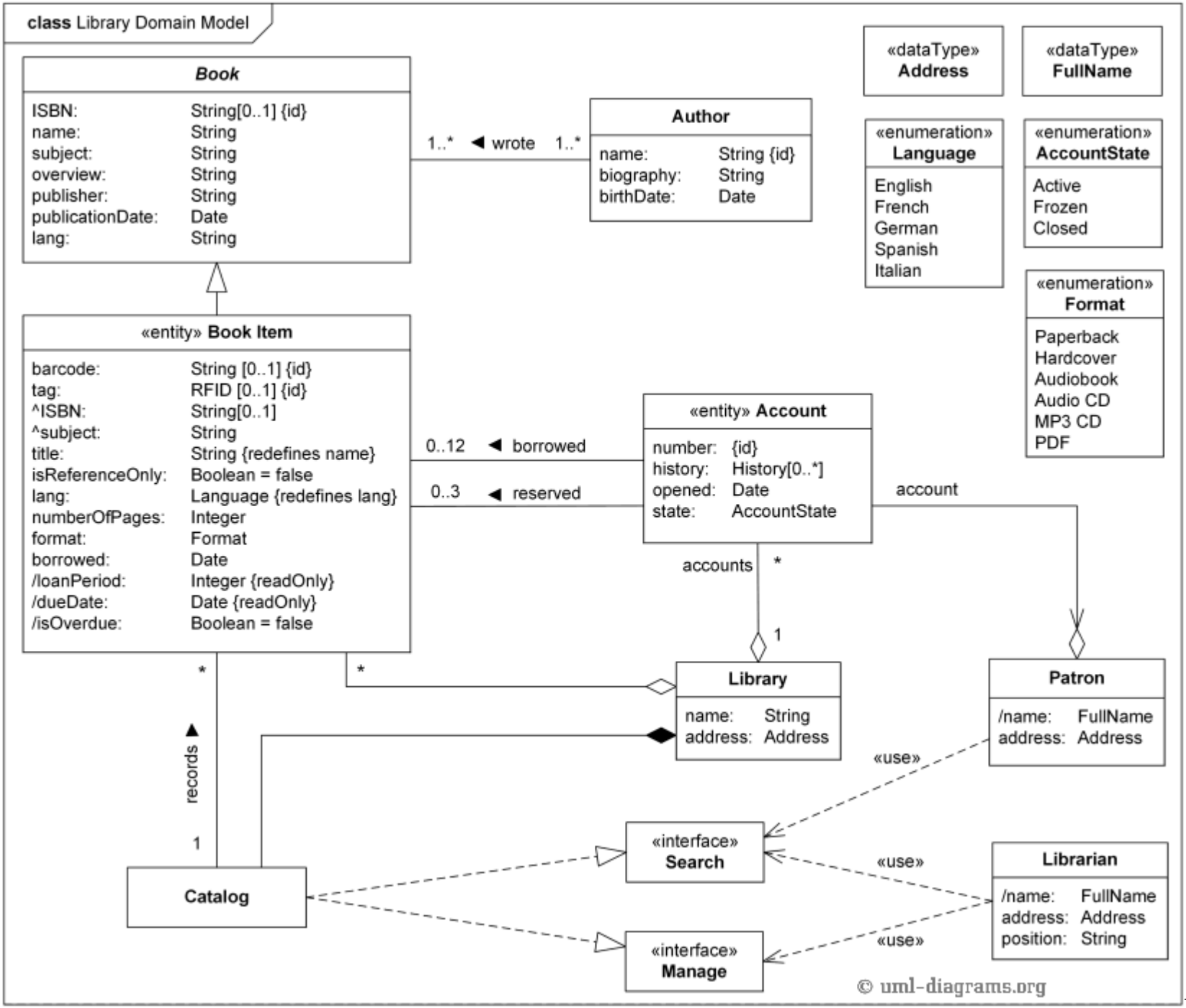

aggregation




composition

# Class relationships: multiplicity

- One-to-one and mandatory:  1
- One-to-one and optional:  0..1
- One-to-many and mandatory:  1..*
- One-to-many and optional:  *
- With lower bound l and upper bound u:  l..u
- With lower bound l and no upper bound:  l..*


class Library Domain Model
Book
ISBN: String[0..1] {id}
name: String
subject: String
overview: String
publisher: String
publicationDate: Date
lang: String
1..* wrote 1..*
Author
name: String {id}
biography: String
birthDate: Date
«dataType» Address
«dataType» FullName
«enumeration» Language
English
French
German
Spanish
Italian
«enumeration» AccountState
Active
Frozen
Closed
«enumeration» Format
Paperback
Hardcover
Audiobook
Audio CD
MP3 CD
PDF
«entity» Book Item
barcode: String [0..1] {id}
tag: RFID [0..1] {id}
^ISBN: String[0..1]
^subject: String
title: String {redefines name}
isReferenceOnly: Boolean = false
lang: Language {redefines lang}
numberOfPages: Integer
format: Format
borrowed: Date
/loanPeriod: Integer {readOnly}
/dueDate: Date {readOnly}
/isOverdue: Boolean = false
0..12 borrowed
0..3 reserved
«entity» Account
number: {id}
history: History[0..*]
opened: Date
state: AccountState
account
accounts
*
1
*
*
records
1
Library
name: String
address: Address
Patron
/name: FullName
address: Address
«use»
«interface» Search
«use»
Librarian
/name: FullName
address: Address
position: String
Catalog
«interface» Manage
«use»
© uml-diagrams.org

# DESIGN PATTERNS

FELIZ GOUVEIA

UFP

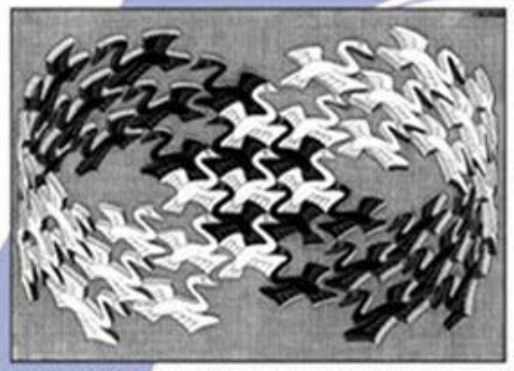
# WHAT ARE THEY?

- **“**In software engineering, a **design pattern** is a general repeatable solution to a commonly occurring problem in software design. A design pattern isn't a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations.”
- *Design Patterns: Elements of Reusable Object-Oriented Software*

# Design Patterns

## Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides





Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

# PATTERN TYPES

- Creational design patterns
  - About class instantiation
- Structural design patterns
  - About class composition
- Behavioral design patterns
  - About class communication

# SINGLETON PATTERN

- For situations where we know we only need one instance:
  - Filesystem
  - Printer controller
  - UI controller
  - Some services (database pool)
- Single point of access to the object
- Possibility of creating more instances in the future

# SINGLETON PATTERN

```
public class ClassicSingleton {

        private static ClassicSingleton instance = null;
        protected ClassicSingleton() { // Exists only to defeat
                                       instantiation. }

        public static ClassicSingleton getInstance() {
                if(instance == null){

                        instance = new ClassicSingleton();

                }
                return instance;
        }
}
```

# SINGLETON PATTERN (1)

Prevent multi-threading issues:

```
public static Singleton getInstance() {
        if(singleton == null) {
                synchronized(Singleton.class) {
                        singleton = new Singleton();
                }
        }
        return singleton;
}
```

# FAÇADE PATTERN

“Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use.”

- Hides complexity and detail
- Heavy work is done by the façade’s code, not by the developer
- Less business objects need to be exposed, which increases flexibility

# FAÇADE PATTERN (1)

Example from the Java Pet Store. A façade centralizes services common to all shopping actions

```
public interface ShoppingClientFacadeLocal extends
EJBLocalObject {
        public ShoppingCartLocal getShoppingCart();
        public void setUserId(String userId);
        public String getUserId();
        public CustomerLocal getCustomer() throws
                                                FinderException;
        public CustomerLocal createCustomer(String userId); }
```

# FACTORY PATTERN

- A superclass specifies all standard and generic behavior and then delegates the creation details to subclasses that are supplied by the client.
- Lets a class defer instantiation to subclasses

# FACTORY PATTERN (1)

```
public class SimpleFactory {
        public Toy createToy(String toyName) {
            if (“car”.equals(toyName)){
                return new Car();
            } else if (“helicopter”.equals(toyName)){
                return new Helicopter();
            else
                return null;
        }
}
```

# FACTORY PATTERN (2)

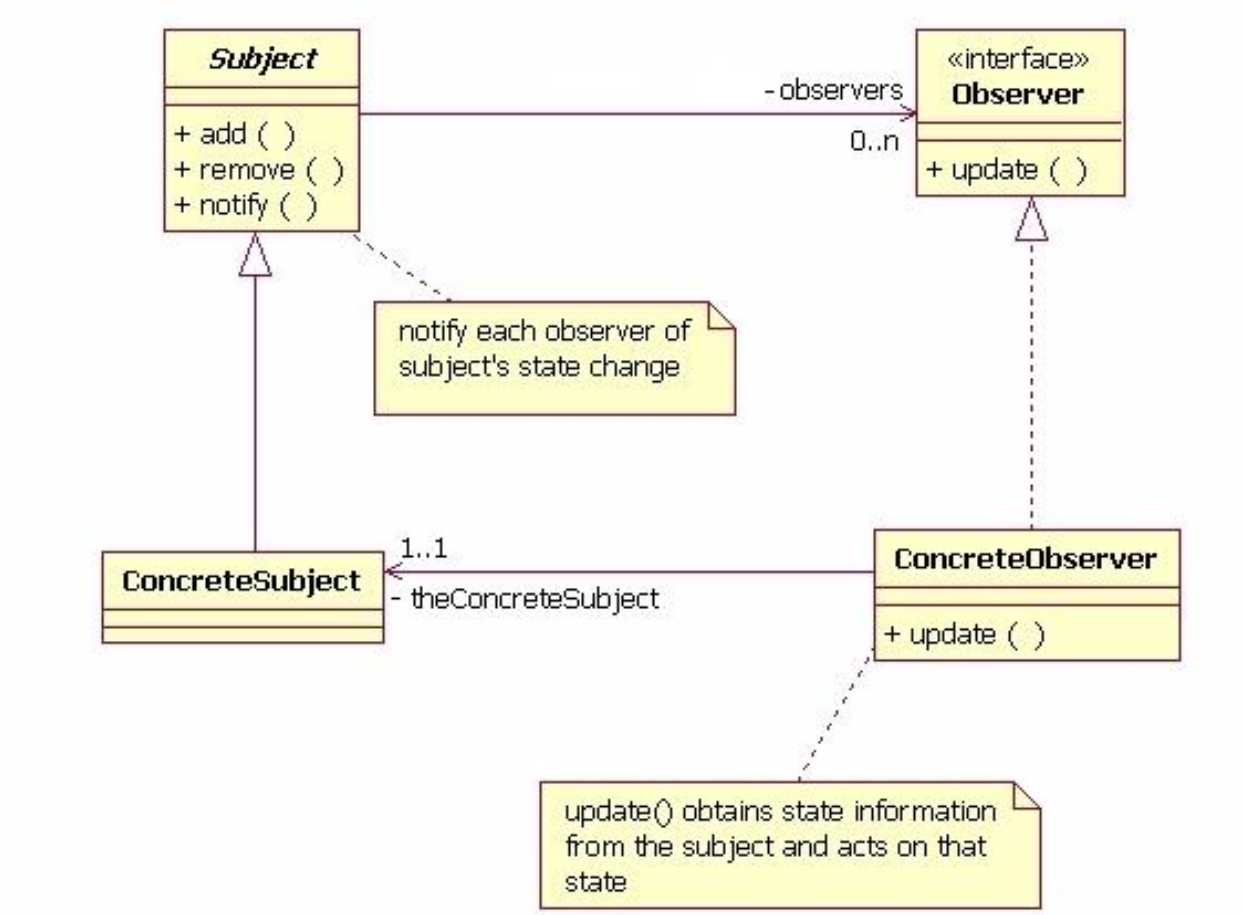
```
public class ToysFactory {
    private simpleFactory;
    public ToysFactory(SimpleFactory simpleFactory) {
        this.simpleFactory = simpleFactory;
    }

    public Toy produceToy(String toyName) {
        Toy toy = simpleFactory.createToy(toyName);
        toy.build();
        toy.package();
        return toy;
    }
}
```

# OBSERVER PATTERN

“Define a one to many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.”

- There are many observers as needed
- Some objects depend on the state of other objects and would like to be notified about any changes to that state
- Classic problem in UI. When user clicks, several objects should be notified

# OBSERVER PATTERN (1)



# OBSERVER PATTERN (2)

- The subject keeps a list of observers that registered to be notified
- Subjects implement a notify() method
- Observers have their update() meyhod called by the subject's notify()

# OBJECT POOL PATTERN

- In some situations objects can be reused
- Object creation can be time-consuming
- An object pool caches objects so that they can be reused when needed
  - Example: database connection pools

# OBJECT POOL PATTERN

- ObjectPool has an internal array of objects
- ObjectPool has acquire() and release() methods
- ObjectPool is a singleton

# PROTOTYPE PATTERN

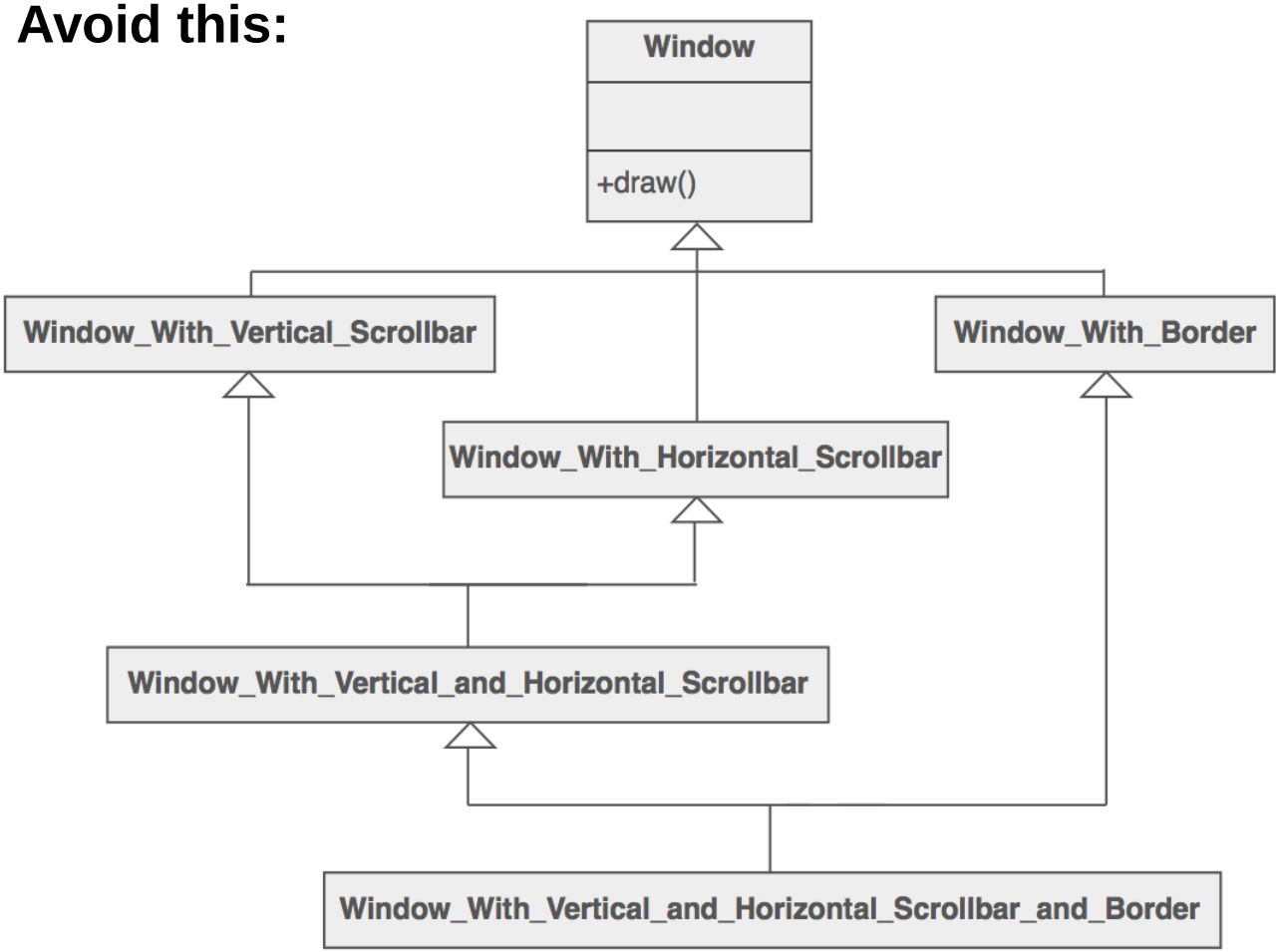
- Maintain a list of prototypes
- Has a clone() method
- The client calls the clone() method to get a new object

# DECORATOR PATTERN

- Add aditional behavior or structure to an object at run time
- The client embelishes an object by wrappint it
- Inheritance does not work because it applies to the class (and to all instances) and it is static

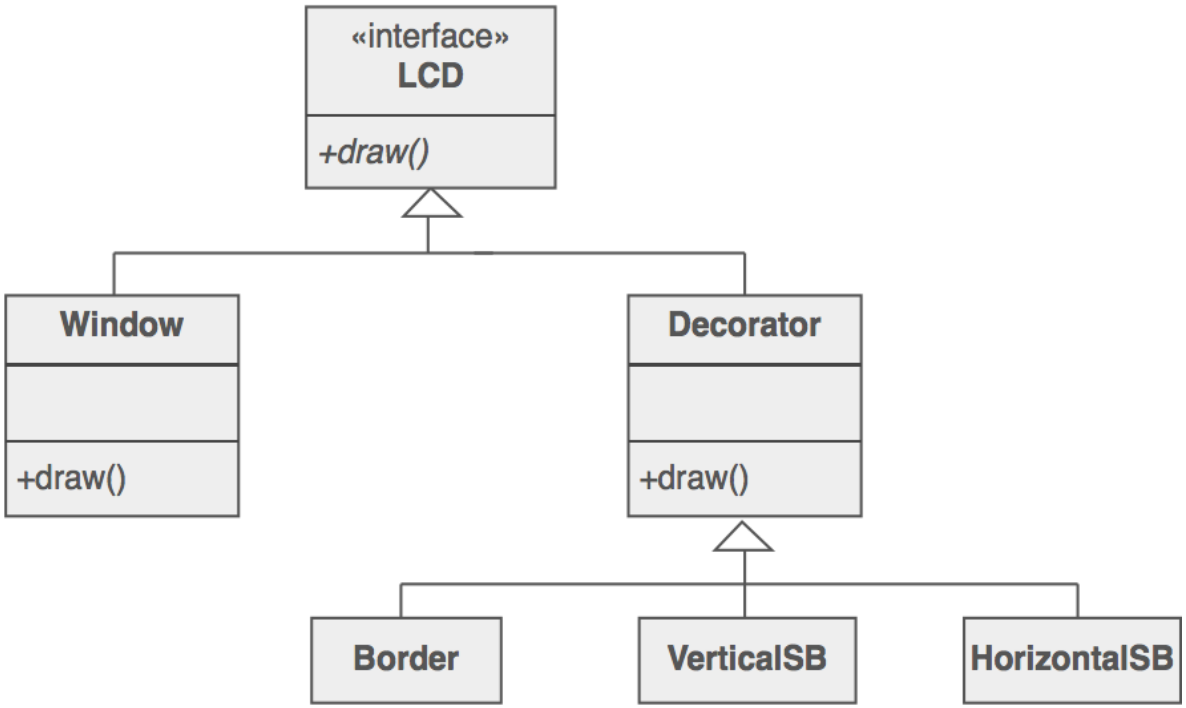
# DECORATOR PATTERN

**Avoid this:**



# DECORATOR PATTERN

```
Widget* aWidget = new BorderDecorator(new
        HorizontalScrollBarDecorator(new
            VerticalScrollBarDecorator( new Window( 80, 24 ))));

aWidget->draw();
```



# NORMA ESA

Feliz Gouveia
Engenharia de Software
Universidade Fernando Pessoa

# ESA PSS

- A norma PSS era adotada pela ESA para o desenvolvimento de projetos de software
- Define um ciclo de vida, e os procedimentos a adotar para gerir o projeto
- Define um conjunto de documentos a produzir
- A seguir mostra-se o modelo de ciclo de vida do software

*Material retirado de ESA PSS-05-0 Issue 2*

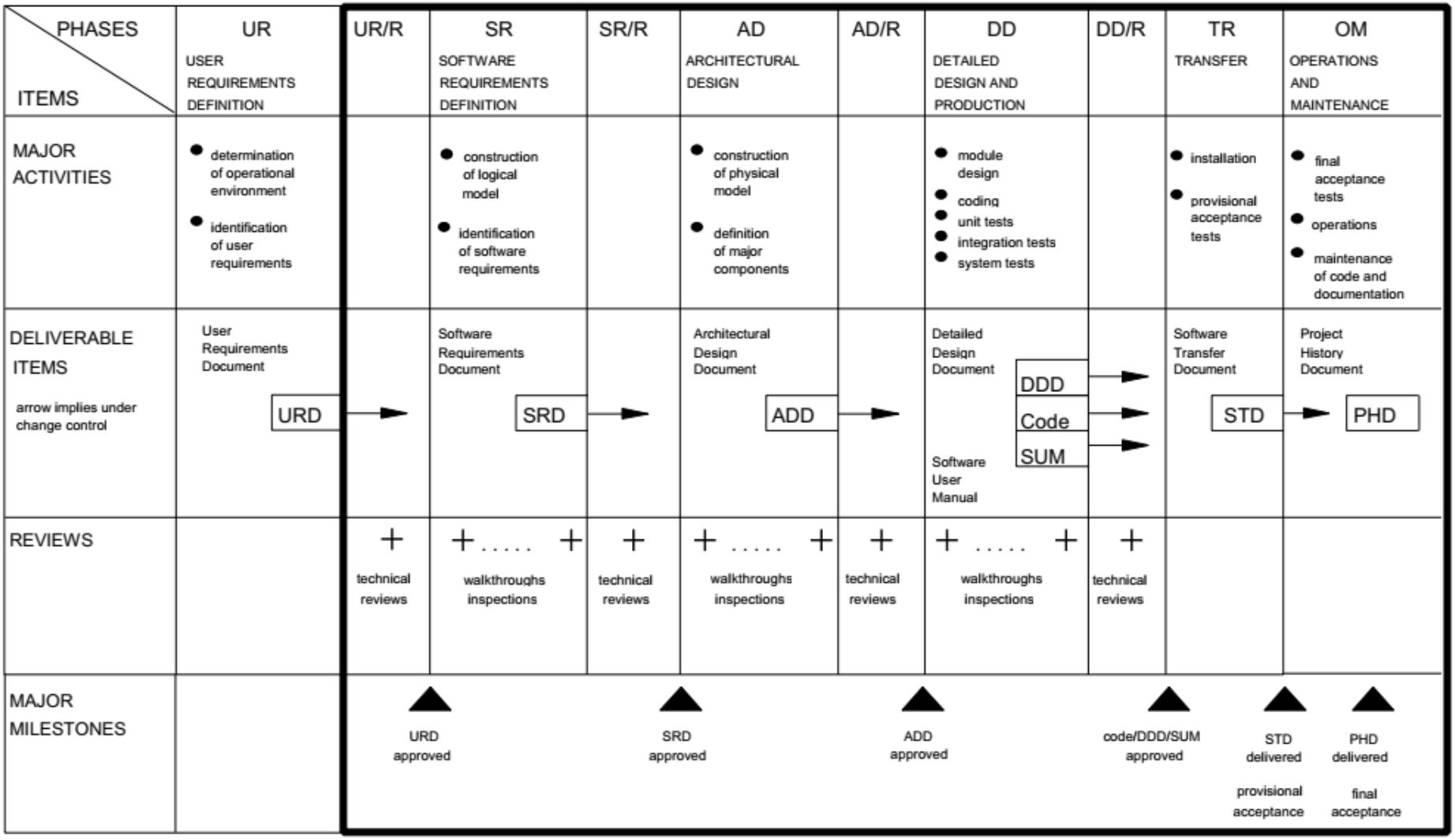



Figure 1.2: The Software Life Cycle Model

# Fases da norma

Todos os projetos devem ter pelo menos as seguintes fases:

- Definição de Requisitos Utilizador (User Requirements - UR)
- Especificação de Requisitos de Software (Software Requirements - SR)
- Especificação de Projeto de Arquitetura (Architectural Design - AD)
- Projeto detalhado e produção de código (Detailed Design - DD)
- Transferência do software para produção (Transfer - TR)
- Operação e Mautenção (OM).

# As fases

- A fase *UR* define o problema, devendo o âmbito do projeto ser claramente definido pelos utilizadores em cooperação com a equipa técnica.
- O ambiente operacional do projeto deve ser definido.
- Os requisitos utilizador são registados num documento *(URD).*
- O documento *URD* deve ser revisto e aprovado *(UR/R)* pela mesma equipa que trabalhou na fase UR.

# As fases

- A fase *SR* consiste na análise e na especificação dos requisitos de software
- É produzido um modelo lógico do software, usado para testar se todos o requisitos foram considerados, se são exequíveis, e se podem ser testados
- O documento *Software Requirements Document (SRD)* deve ser revisto (SR/R) pelos utilizadores, programadores e responsáveis de projeto.

# As fases

- A fase *Architecture Design (AD)* define a arquitetura do sofware, definindo os seus componentes e interfaces.
- Os fluxos de dados entre componentes são identificados.
- O resultado desta fase é o documento *Architectural Design Document (ADD).* O ADD deve ser revisto e aprovado pela mesma equipa.

# As fases

- As atividades da fase *DD* incluem o projeto, programação, teste e integração dos modulos do sistema.
- Paralelamente à programação e teste produz-se o *Detailed Design Document (DDD)* e o *Software User Manual (SUM)*.
- Os testes unitários, de integração e de sistema são realizados de acordo com o planeado nas fases SR e AD.

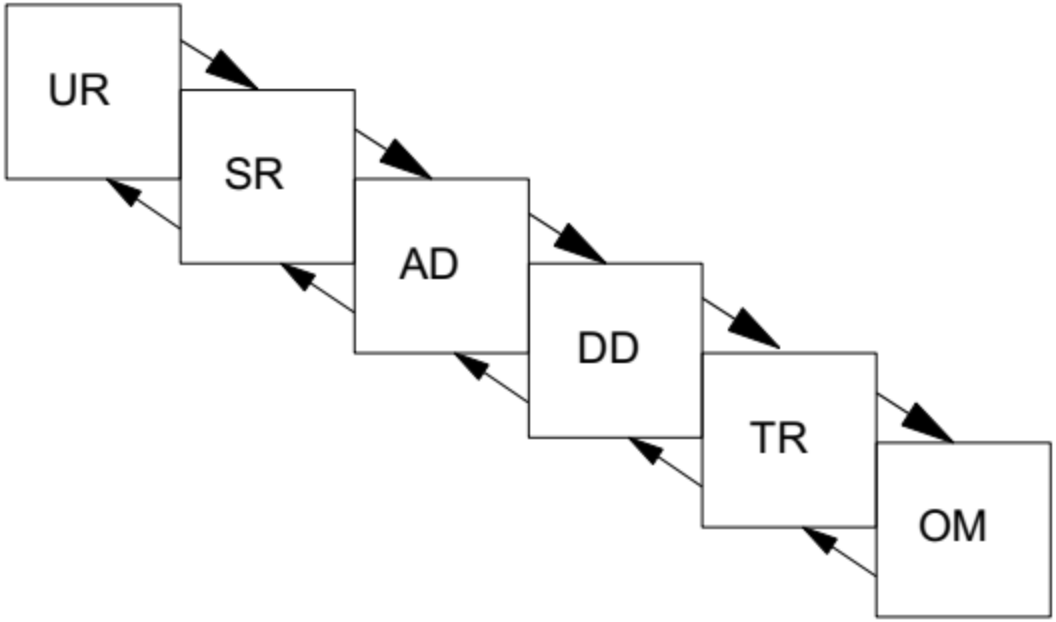
# As fases

- O código produzido e os documentos DDD e SUM são revistos, pelos engenheiros de software e responsáveis de projeto, na fase *Detailed Design Review (DD/R).*
- A fase *TR* inclui a instalação e os testes de aceitação provisória e destinam-se a demonstrar ao cliente a conformidade com com os reuquisitos.
- O documento *Software Transfer document (STD)* descreve as atividades necessárias para transferir a operação do sistema para a equipa do cliente.
- Na fase *OM* a operação do sistema é monitorada para se poder realizar a aceitação final por parte do cliente.

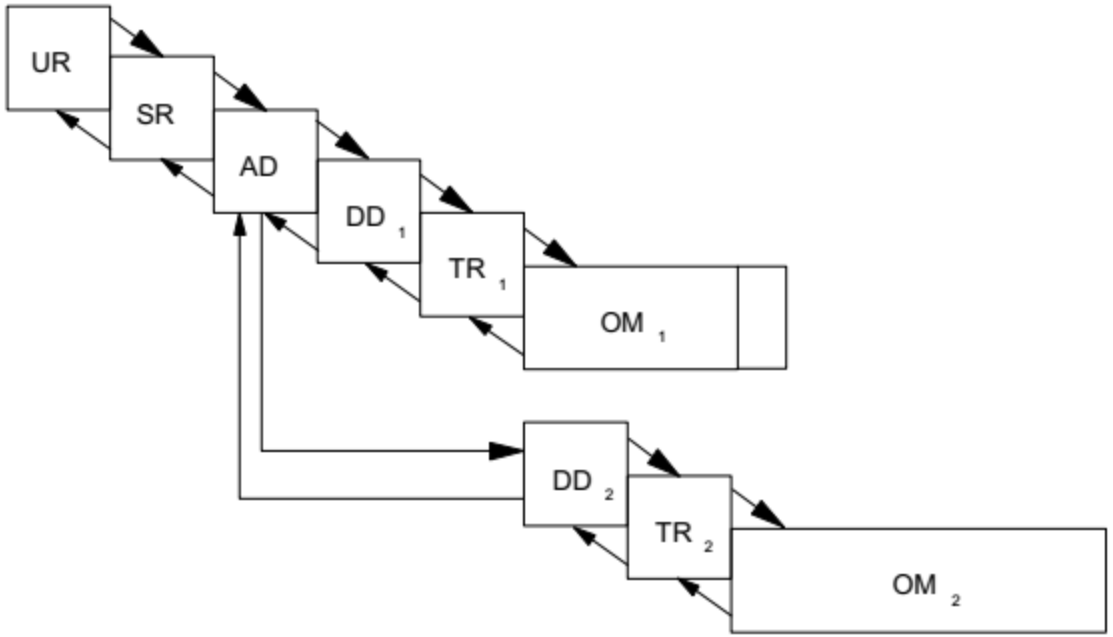
# Exemplos de ciclos de vida

- O modelo de ciclo de vida do software pode ser implementado com diferentes ciclos de vida
- A seguir mostram-se alguns

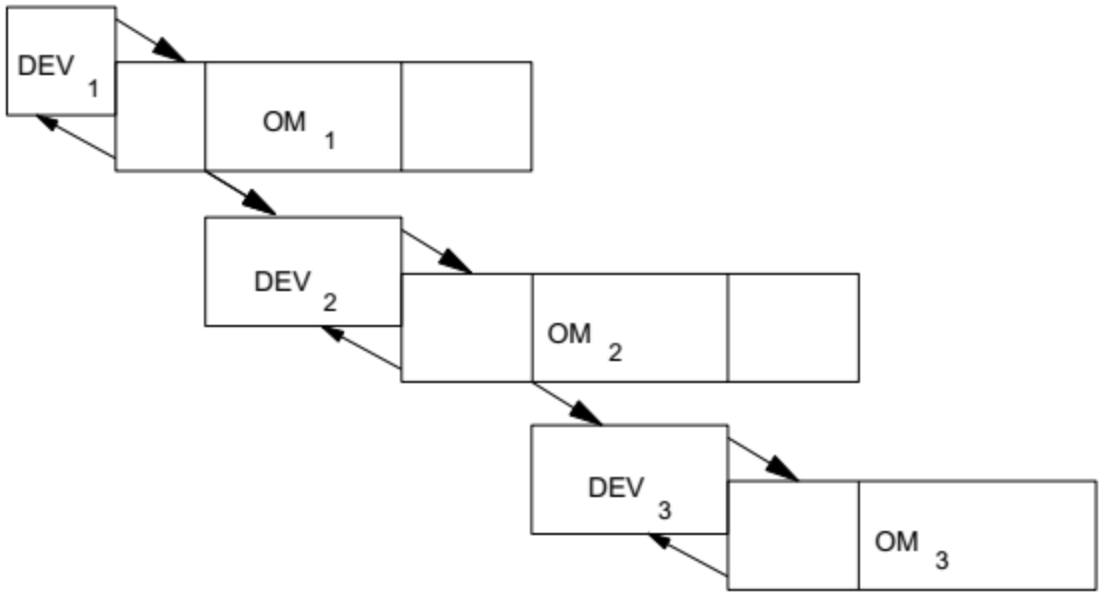
## The waterfall approach



## The incremental delivery approach



## The evolutionary development approach



# STANDARDS: ESA PSS-05

Introduction to the UR Phase

Feliz Gouveia

Software Engineering

# The UR Phase

- The UR phase can be called the ‘problem definition phase' of the life cycle.
- The phase refines an idea about a task to be performed using computing equipment, into a definition of what is expected from the computer system.

# Capture of UR

- user requirements should be clarified through criticism and experience of existing software and prototypes;
- wide agreement should be established through interviews and surveys;
- knowledge and experience of the potential development organisations should be used to help decide on implementation feasibility, and,
  perhaps to build prototypes.

# UR should be realistic

- Realistic user requirements are:
  - clear;
  - verifiable;
  - complete;
  - accurate;
  - feasible.

# Requirements

- Clarity and verifiability help ensure that delivered systems will meet user requirements.
- Completeness and accuracy imply that the URD states the user's real needs.
- A URD is inaccurate if it requests something that users do not need, for example a superfluous capability or an unnecessary design constraint

# Realistic UR

- Realistic user requirements must be feasible. If the resources and timescales available for its implementation are insufficient, it may be unrealistic to put them in a URD.

# Operational environment

- A clear description of the real world that the software will operate in should be built up, as the user requirements are captured.
- This description of the operational environment must clearly establish the problem context

# Users

- The roles and responsibilities of the users and operators of software should be established by defining the:
  - characteristics of each group (e.g. experience, qualifications);
  - operations they perform (e.g. the user of the data may not operate the software).

# Capability requirements

- Capability requirements describe the process to be supported by software.
- Simply stated, they describe ‘what' the users want to do.
- A capability requirement should define an operation, or sequence of related operations, that the software will be able to perform.

# Capability requirements (1)

- Quantitative statements that specify performance and accuracy attributes should form part of the specification of capability.
- This means that a capability requirement should be qualified with values of:
  - capacity;
  - speed;
  - accuracy.
- The performance attribute is the combination of the capacity and speed attributes.

# Capacity

- The capacity attribute states ‘how much' of a capability is needed at any moment in time. Each capability requirement should be attached with a quantitative measure of the capacity required.
- For example the:
  - number of users to be supported;
  - number of terminals to be supported;
  - number of satellites that can be controlled simultaneously;
  - amount of data to be stored.

# Speed

- The speed attribute states how fast the complete operation, or sequence of operations, is to be performed.
- Each capability requirement should be attached with a quantitative measure of the speed required. There are various ways to do this, for example the:
  - number of operations done per unit time interval;
  - time taken to perform an operation.
- For example: ‘95% of the transactions shall be processed in less than 1 second', is acceptable whilst, ‘95% of the transactions will be done as soon as possible' is not.

# Constraint requirements

- Constraint requirements place restrictions on how the user requirements are to be met.
- The user may place constraints on the software related to interfaces, quality, resources and timescales.
- Users may constrain how communication is done with other systems, what hardware is to be used, what software it has to be compatible with, and how it must interact with human operators.
- These are all interface constraints. An interface is a shared boundary between two systems; it may be defined in terms of what is exchanged across the boundary

# Communications interfaces

- A communications interface requirement may specify the networks and network protocols to be used. Performance attributes of the interface may be specified (e.g. data rate).

# Software interfaces

- A software interface requirement specifies whether the software is to be compatible with other software (e.g other applications, compilers, operating systems, programming languages and database management systems).

# Human-Computer Interaction

- A Human-Computer Interaction (HCI) requirement may specify any aspect of the user interface. This may include a statement about style (e.g. command language, menu system, icons), format (e.g. report content and layout), messages (e.g. brief, exhaustive) and responsiveness (e.g. time taken to respond to command). The hardware at the user interface (e.g. colour display and mouse) may be included either as an HCI requirement or as a hardware interface requirement.

# Availability

- Availability measures the ability of a system to be used during its intended periods of its operation. Availability requirements may specify:
  - mean and minimum capacity available (e.g. all terminals);
  - start and end times of availability (e.g. from 0900 to 1730 daily);
  - time period for averaging availability (e.g. 1 year).
- Examples of availability requirements are: ‘the user shall be provided with 98% average availability over 1 year during working hours and never less than 50% of working hours in any one week';

# Portability

- Software portability is measured by the ease that it can be moved from one environment to another. Portable software tends to be long lived, but more code may have to be written and performance requirements may be more difficult to meet.
- An example of a portability requirement is: ‘the software shall be portable between environments X and Y'.

# Security

- A system may need to be secured against threats to its confidentiality, integrity and availability.
- For example, a user may request that unauthorised users be unable to use the system, or that no single event such as a fire should cause the loss of more than 1 week's information.
- The user should describe threats that the system needs to be protected against, e.g. virus intrusions, hackers, fires, computer breakdowns.
- The security of a system can be described in terms of the ownership of, and rights of access to, the capabilities of the system.

# Safety

- The consequences of software failure should be made clear to developers.
- Safety requirements define the needs of users to be protected against potential problems such as hardware or software faults.
- They may define scenarios that the system should handle safely (e.g. ‘the system should ensure that no data is lost when a power failure occurs')

# Standards

- Standards requirements normally reference the applicable documents that define the standard.

# Resources

- The resources available for producing and operating the software are a constraint on the design.
- Resource requirements may include specifications of the computer resources available (e.g. main memory). They may define the minimum hardware that the system must run on.

# Acceptance Test Planning

- Validation confirms whether the user requirements are satisfied when the software is delivered.
- This is done by performing acceptance tests in the Transfer Phase.

# UR Review

- The outputs of the UR phase must be formally reviewed during the User Requirements Review. This should be a technical review.
- Normally, only the URD and the Acceptance Test Plan undergo the full technical review procedure involving users, developers, management and quality assurance staff.

# Methods for UR definition

- Interviews and surveys
- Observation of existing systems
- Prototyping
- Feasibility studies
- …

# Output of this phase: URD

- A URD is ‘clear' if each requirement is unambiguous and understandable to project participants.
- A requirement is unambiguous if it has only one interpretation. To be understandable, the language used in a URD should be shared by all project participants and should be as simple as possible

# The URD

- Each requirement should be stated in a single sentence. Justifications and explanations of a requirement should be clearly separated from the requirement itself.
- Clarity is enhanced by grouping related requirements together. The capability requirements in a group should be structured to reflect any temporal or causal relationships between them.

# The URD (1)

- A URD is consistent if no requirements conflict. Using different terms for what is really the same thing, or specifying two incompatible qualities, are examples of lack of consistency.

# The URD (2)

- A URD is modifiable if any necessary requirements changes can be documented easily, completely, and consistently.
- A URD contains redundancy if there are duplicating or overlapping requirements.
- Redundancy itself is not an error, and redundancy can help to make a URD more readable, but a problem arises when the URD is updated.

# The URD (3)

- Changes to the URD are the user's responsibility. The URD should be put under change control by the initiator at soon as it is first issued.
- This requires that a change history be kept.
- New user requirements may be added and existing user requirements may be modified or deleted. If anyone wants to change the user requirements after the UR phase, the users should update the URD and resubmit it to the UR/R board for approval.

# The URD (4)

- The definition of the user requirements must be the responsibility of the user.
- This means that the URD must be written by the users, or someone appointed by them.
- The expertise of software engineers, hardware engineers and operations personnel should be used to help define and review the user requirements.

# Acceptance Test Plans

- The initiator(s) of the user requirements should lay down the acceptance test principles.
- The developer must construct an acceptance test plan in the UR phase and document it.
- This plan should define the scope, approach, resources and schedule of acceptance testing activities.
- Specific tests for each user requirement are not formulated until the DD phase.

# Acceptance Test Plans (1)

- The Acceptance Test Plan should deal with the general issues, for example:
  - where will the acceptance tests be done?
  - who will attend?
  - who will carry them out?
  - are tests needed for all user requirements?
  - what kinds of tests are sufficient for provisional acceptance?
  - what kinds of tests are sufficient for final acceptance?
  - must any special test software be used?

# STANDARDS: ESA PSS-05

Introduction to the ADD Phase

Feliz Gouveia

Software Engineering

# The ADD Phase

- The UR phase can be called the ‘solution phase' of the life cycle.
- The purpose of this phase is to define a collection of software components and their interfaces to establish a framework for developing the software.
- This is the ‘Architectural Design’, and it must cover all the requirements in the SRD.

# ADD activities

- The principal activity of the AD phase is to develop the architectural design of the software and document it in the ADD. This involves:
  - constructing the physical model;
  - specifying the architectural design;
  - selecting a programming language;
  - reviewing the design.

# Construction of the physical model

- The physical model **should** be derived from the logical model, described in the SRD.
- In transforming a logical model to a physical model, “design decisions” are made in which functions are allocated to components and their inputs and outputs defined.
- Design decisions **should** also satisfy non-functional requirements, design quality criteria and implementation technology considerations.
- Design decisions **should** be recorded.
- CASE tools **should** be used.

# Decomposition of the software into components

- The software **should** be decomposed into a hierarchy of components according to a partitioning method. Examples of partitioning methods are “functional decomposition” and “correspondence with real world objects”.
- Top-down decomposition is vital for controlling complexity because it enforces “information hiding” by demanding that lower-level components behave as “black boxes”.
- Only the function and interfaces of a lower-level component are required for the higher-level design.

# Implementation of non-functional requirements

- The design of each component **should** be reviewed against each of the SRD requirements. While some non-functional requirements may apply to all components in the system, other non-functional requirements may affect the design of only a few components.
- Some SRD requirements:
  - Performance requirements
  - Interface requirements
  - Operational requirements
  - Resource requirements
  - Verification requirements
  - Acceptance testing requirements
  - Documentation requirements
  - Security requirements

# Design quality criteria

- Designs **should** be adaptable, efficient and understandable.
- Adaptable designs are easy to modify and maintain.
- Efficient designs make minimal use of available resources.
- Designs must be understandable if they are to be built, operated and maintained effectively.
- Designs **should** be ‘modular’, with minimal coupling between components and maximum cohesion within each component. There is minimal duplication between components in a modular design.
- Components of a modular design are often described as “black boxes” because they hide internal information from other components.

# Trade-off between alternative designs

- There is no unique design for any software system.
- Studies of the different options may be necessary. A number of criteria will be needed to choose the best option. The criteria depend on the type of system.
- Only the selected design approach **shall** be reflected in the ADD (and DDD).

# Specification of the architectural design

- The architectural design is the fully documented physical model.
- This **should** contain diagrams showing, at each level of the architectural design, the data flow and control flow between the components.
- Block diagrams, showing entities such as tasks and files, may also be used to describe the design.
- The diagramming techniques used **should** be documented or referenced.

# Functional definition of the components

- The process of architectural design results in a set of components having defined functions and interfaces.
- The functions of each component will be derived from the SRD.
- The level of detail in the ADD will show which functional requirements are to be met by each component, but not necessarily how to meet them: this will only be known when the detailed design is complete.
- For each component the following information **shall** be defined in the ADD:
  - data input;
  - functions to be performed;
  - data output.

# Definition of the data structures

- Data structures that interface components **shall** be defined in the ADD.
- Data structure definitions **shall** include the:
  · description of each element (e.g. name, type, dimension);
  · relationships between the elements (i.e. the structure);
  · range of possible values of each element;
  · initial values of each element.

# Definition of the control flow

- The definition of the control flow between components is essential for the understanding of the software's operation.
- The control flow between the components **shall** be defined in the ADD.

# Definition of the computer resource utilisation

- The computer resources (e.g. CPU speed, memory, storage, system software) needed in the development environment and the operational environment **shall** be estimated in the AD phase and defined in the ADD

# Selection of programming languages

- Programming languages **should** be selected that support top down decomposition, structured programming and concurrent production and documentation.
- The programming language and the AD method **should** be compatible.
- Non-functional requirements may influence the choice of programming language.

# Reviews

- The architectural design **should** be reviewed and agreed layer by layer as it is developed during the AD phase.
- The design of any level invariably affects upper layers: a number of review cycles may be necessary before the design of a level can be finalised.
- Walkthroughs **should** be used to ensure that the architectural design is understood by all those concerned.

# OUTPUTS FROM THE PHASE

- The main outputs of the phase are the ADD and the plans for the DD phase.

# The ADD

- The Architectural Design Document (ADD) is the key document that summarises the solution.
- It is the kernel from which the detailed design grows. The ADD **shall** define the major components of the software and the interfaces between them.
- The ADD **shall** define or reference all external interfaces.
- The ADD **shall** be an output from the AD phase.
- The ADD **shall** be complete, covering all the software requirements described in the SRD. To demonstrate this, a table cross-referencing software requirements to parts of the architectural design **shall** be placed in the ADD.

# The ADD (1)

- The ADD **shall** be consistent. Software engineering methods and tools can help achieve consistency, and their output may be included in the ADD.
- The ADD **shall** be sufficiently detailed to allow the project leader to draw up a detailed implementation plan and to control the overall project during the remaining development phases.
- The ADD **should** be detailed enough to enable the cost of the remaining development to be estimated to within10%.

# The ADD (2)

- The estimate of the total project cost (accurate to 10%), and the management plan for the DD phase, must be documented in the DD phase section of the Software Project Management Plan.
- The configuration management procedures for the documents, deliverable code, CASE tool products and prototype software, to be produced in the DD phase, must be documented in the Software Configuration Management Plan.