



# Software Engineering I

Dr. Elham Mahmoudzadeh  
Isfahan University of Technology  
[mahmoudzadeh@iut.ac.ir](mailto:mahmoudzadeh@iut.ac.ir)

202\

The background features a light gray gradient with several realistic water droplets of varying sizes scattered across the surface. In the center, there is a large, faint, circular logo. The logo consists of a gear-like outer ring with Persian text 'سازمان صنعتی اصفهان' (Industrial Organization of Ahvaz) written along its top arc. Inside the gear is a stylized sunburst or star-like symbol. Overlaid on this logo is the chapter title in a bold, black, serif font.

# **Chapter 7**

## **Moving To Design**

# Steps(I)

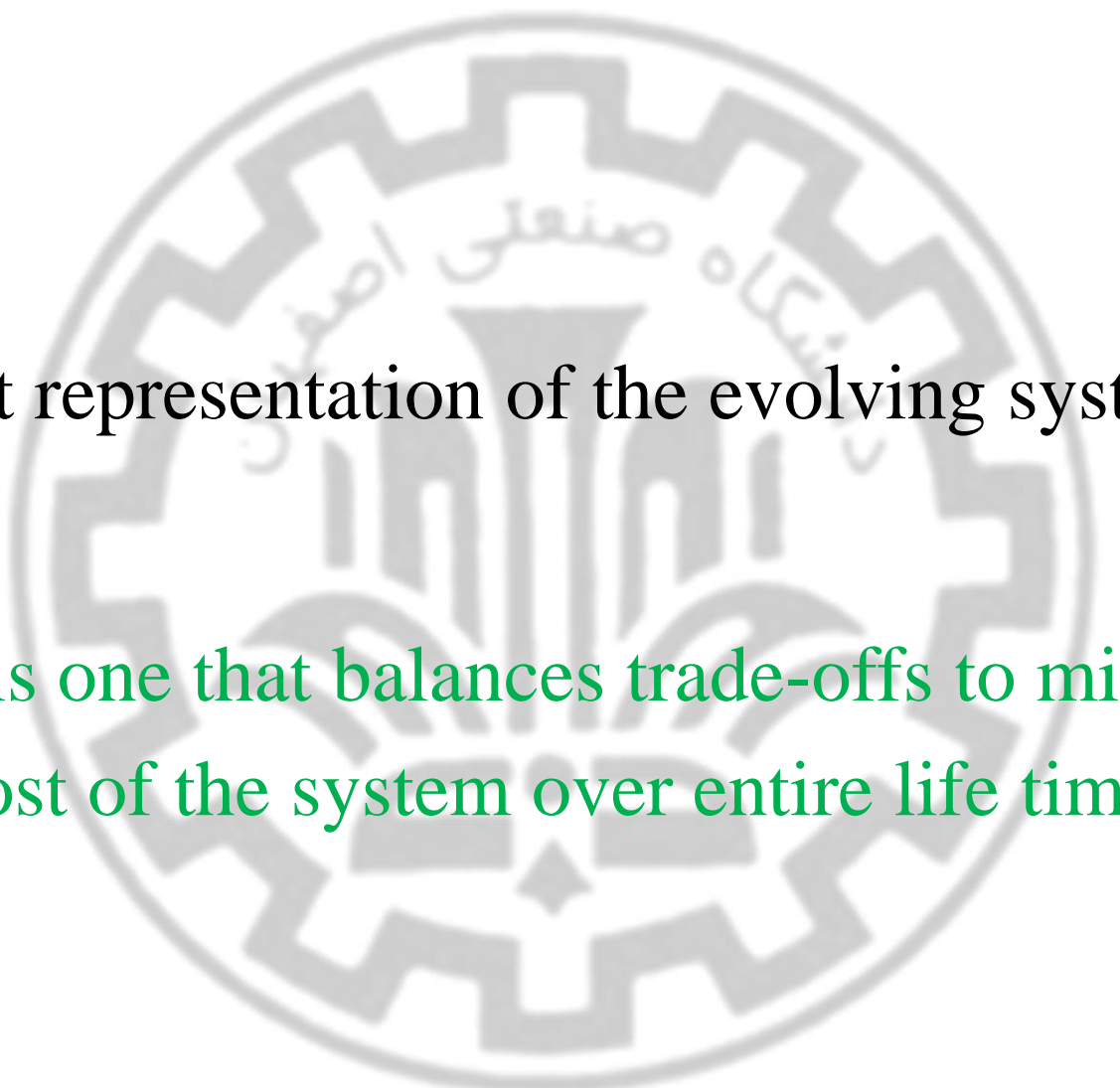
1. Preparing proposal
2. Requirements determination
  - User story
3. Abstract Business Process Modelling
4. Analysis
  - Functional Modelling
  - Structural Modelling
  - Behavioral Modelling

## Steps(II)

### 5. Design

- Optimization
- Database Management
- User Interface
- Physical Architecture





Is the current representation of the evolving system optimal?

A good design is one that balances trade-offs to minimize the total cost of the system over entire life time.



# What is design?

- Design is what almost every engineer wants to do.
- It is the place where creativity rules—where stakeholder requirements, business needs, and technical considerations all come together in the formulation of a product or system.
- Creates a representation or model of the software, but unlike the requirements model (that focuses on describing required data, function, and behavior), the design model provides detail about software architecture, data structures, interfaces, and components that are necessary to implement the system.



## طراحی:

چیدن اوان المان های تشکیل دهنده

همون چیزی که مهندس ها انجام میدن همون ذهن خلاق یک مهندس است

توی طراحی خیلی نمیشه از رویکردهای اتوماتیک استفاده کرد

ینی همه اوان نیاز ها و مدل های تحلیل و.. میان یه جا جمع میشن تا چیده بشه

مدل نرم افزار رو به ما میگه و این مدله علارغم اوان مدل های تحلیل که تمرکزشون روی رفتار یا

فرایند بود... مدل های طراحی با جزئیات بیشتری سیستم رو مدل می کنن ینی جزئیاتش بیشتره -->

این جزئیات به دنیای پیاد سازی نزدیک تره پس مدل های هست که جزئیات به اندازه کافی داشته

باشه برای پیاده سازی



# Why is Design important?

- Design allows you to model the system or product that is to be built.
- This model can be assessed for quality and improved before code is generated, tests are conducted, and end users become involved in large numbers.
- Design is the place where software quality is established.

پایه سیستم رو داریم توی طراحی می چینیم

پس اگر مدل ایراد داشت باشه یا به نحو درستی در نظر گرفته نشده باشه فرایند تولید و کارکرد نرم افزار با شکست مواجه میشه پس خیلی مهمه ما ریزبینانه توی فاز طراحی بریم جلو و خلاقیت اینجا خیلی نقش مهمی داری

اهمیت دیگه از بعد کیفیت سیستم است ما توی فاز دیزاین به کیفیت برمی خوریم ینی در نظر بگیریم کیفیت رو و قبل از اینکه کدی تولید بشه و سیستم بیاد بالا باید ارزیابی بشه اون کیفیت نرم افزار که توی فاز طراحی باید مورد ارزیابی کیفی قرار بگیره

# How do I ensure that I've done it right?

- The design model is assessed by the software team in an effort to determine whether it contains errors, inconsistencies, or omissions; whether better alternatives exist; and whether the model can be implemented within the **constraints**, **schedule**, and **cost** that have been established.

در حالت کلی باید بررسی کنیم خروجی قابل قبول سیستم با در نظر گرفتن هزینه ها و.. و با محدودیت ها داره میاد بالا

ینی بررسی هایی که داریم انجام میدیم اینا با هم ناسازگاری نداشته باشند  
ایا اون پلنی که داریم پلن بهینه است یا نه

و مدل باید جوری باید بالا و دیزاین بشه که توی زمان مورد نظر خودش، با هزینه و با در نظر گرفتن محدودیت ها --> ینی خروجی با اینا باید بالا --> پس اینا روی نحوه عملکرد درست ما می تونه کاملاً اثر بذاره

# Three important features of a good design

- The Roman architecture critic Vitruvius advanced the notion that well-designed buildings were those which exhibited **firmness**, **commodity**, and **delight**. The same might be said of good software.
- *Firmness*: A program should not have any bugs that inhibit its function.
- *Commodity*: A program should be suitable for the purposes for which it was intended.
- *Delight*: The experience of using the program should be a pleasurable one.

ویژگی هایی که بیانگر اینه که طراحی ما درست بوده:

ینی سیستم ما باگ نداشته باشه و عملکرد درست خودش رو انجام بده : 1

برای اون هدف مورد نظر برنامه ما و نرم افزار ما مناسب باشه ینی مناسب جامعه هدف باشه : 2

تجربه کار کردن با این نرم افزار تجربه خوبی باشه : 3

## Two main steps

- **Diversification:** is the acquisition of a repertoire of alternatives, the raw material of design: components, component solutions, and knowledge, all contained in catalogs, textbooks, and the mind.” Once this diverse set of information is assembled, you must pick and choose elements from the repertoire that meet the requirements defined by requirements engineering and the analysis model.
- **Convergence.** As this occurs, alternatives are considered and rejected, and you converge on “one particular configuration of components, and thus the creation of the final product”.



دوتا مرحله داریم:

- 1- توی هر جنبه طراحی که بخوایم بریم تنوع زیاد داریم و باید به اون راه حلی برسیم که کمترین هزینه رو داشته باشه توی طول عمر اون نرم افزار --> از روی متن بالا بخون --> ابعاد مختلف رو در نظر بگیریم تا به یک راه حل برسیم



Software design sits at the technical kernel of SE  
and  
is applied regardless of the software process model that is used.

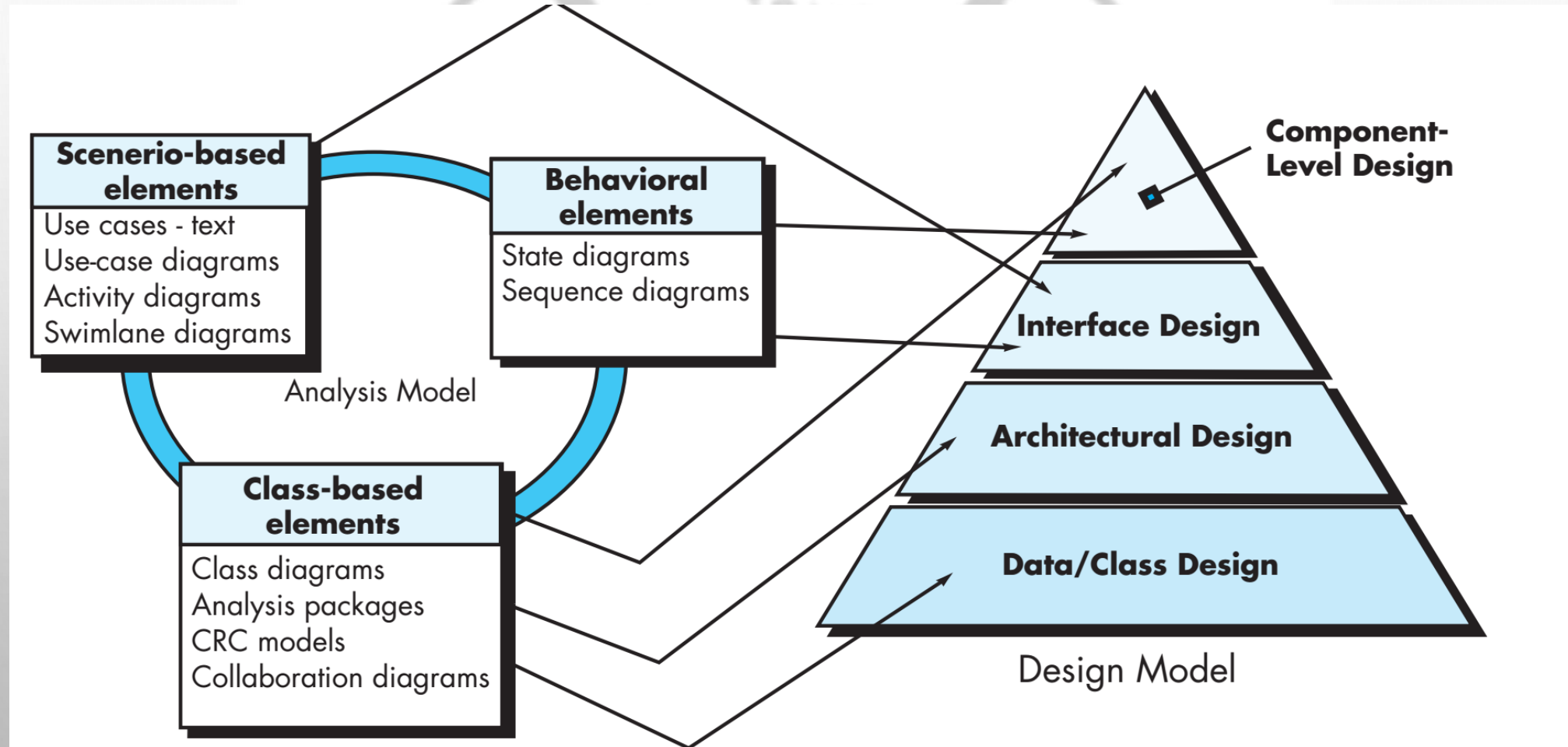
طراحی نرم افزار اون تکنیکال کرنل است ینی هسته فنی کار ما است و مستقل از اینه که مدل فرایند چی باشه و ترتیب کارها چیه خود طراح نرم افزار یک تیم خلاق می خواد --> اتاق فکر نرم افزار طراح های نرم افزار هستن

# Moving from Analysis to Design

- Once software requirements have been analyzed and modeled, software design is the last software engineering action within the modeling activity and sets the stage for **construction** (code generation and testing).
- Each of the elements of the requirements model provides information that is necessary to create design models required for a complete specification of design.
- The requirements model feed the design task.
- Using design notation and design methods, design produces a data/class design, an architectural design, an interface design, and a component design.

- اهمیت دیزاین خیلی زیاده  
نرم افزارهایی قوی و ماندگارن که اتاق فکر قویه پشتشون باشه  
خروجی طراحی وارد فاز تولید کد نرم افزار و تست کد میشه  
هر کدوم از اون مولفه هایی که توی مدل هامون داریم چه یوزکیس دیاگرام و... یک بخشی از  
اطلاعات رو به ما میدن که مدل هایی طراحی رو به چه نحوی تولید بکنیم  
در نهایت از متدهای دیزاین استفاده میکنیم

# Translating the requirements model into the design model







# Data/Class design

- The data/class design transforms class models into design class realizations and the requisite data structures required to implement the software.
- The objects and relationships defined in the CRC and the detailed data content depicted by class attributes and other notation provide the basis for the data design activity.
- More detailed class design occurs as each software component is designed.

توی مدل کلاس دیاگراممون تبدیل میشه به مدل کلاس طراحی ینی داده های بیشتر بهش اضافه میشه تا اون دبیی امس برای اینکه وصل بشه به سیستم و دیتا رو بازیابی بکنه  
ابجکت ها که تعریف کردیم توی کلاس دیاگرام و CRC کمک میکنه که ویژگی ها و اپریشن هارو تعریف بکنیم توی قسمت دیزاین  
کلاس دیزاین جزئی تر است

# Architectural design

- The architectural design defines the relationship between major structural elements of the software, the architectural styles and patterns that can be used to achieve the requirements defined for the system, and the constraints that affect the way in which architecture can be implemented.

توی این لول ما میایم طراحی و چیدن مولفه های نرم افزار رو انجام میدیم  
که این مولفه ها می تونن ماژول های ساختاری سیستم باشن که بهش میگیم معماری داخلی  
یا می تونن دیوایس ها باشن مثل کلاینت ها  
پس چیدن این مولفه ها میشن معماری نرم افزار و چه ارتباطی این مولفه ها با هم دیگه دارن

# Interface design

- The interface design describes how the software communicates with systems that interoperate with it, and with humans who use it.
- An interface implies a flow of information (e.g., data and/or control) and a specific type of behavior.
- Therefore, usage scenarios and behavioral models provide much of the information required for interface design.

-  
توی این قسمت ما داریم نحوه اینترکشن اند یوزرها رو داریم با سیستم داریم ترسیم می کنیم

# Component-level design

- The component-level design transforms structural elements of the software architecture into a procedural description of software components.
- Information obtained from the class-based models and behavioral models serve as the basis for component design.



-  
ینی المان هایی مازول های نرم افزاری درون خودشون چه اطلاعاتی دارن

# The Design Process

- Software design is an iterative process through which requirements are translated into a “blueprint” for constructing the software.
- Initially, the blueprint depicts a holistic view of software. That is, the design is represented at a high level of abstraction—a level that can be directly traced to the specific system objective and more detailed data, functional, and behavioral requirements.
- As design iterations occur, subsequent refinement leads to design representations at much lower levels of abstraction. These can still be traced to requirements, but the connection is more subtle.

فرایند یک فرایند iterative است ینی ما نباید انتظار داشته باشیم خروجی اول دیزاین بهترین خروجی باشه بلکه این ها در قالب تکراری های بیشتر تبدیل میشه به اون معماری اولیه و طراحی بقیه المان های سیستم تا یک بیس اولیه از سیستم در بیاد و این وارد فاز پیاده سازی بشه

# Importance of Design in terms of *quality*

- During design you make decisions that will ultimately affect the success of software construction.
- The importance of software design can be stated with *quality*.
- Design is the place where quality is fostered in software engineering.
- Design provides you with representations of software that can be assessed for quality.
- Design is the only way that you can accurately translate stakeholder's requirements into a finished software product or system.

چرا طراحی نرم افزار مهمه؟ چون بیس سیستمون است و پایه های سیستم اینجا چیده میشه  
هر تصمیمی که می گیریم خیلی مهمه  
اینجا ما متوجه میشیم که ایا نیازهای کاربر به نحوی درستی پاسخگو بوده است یا نه  
اینجا داریم پایه ها رو می چینیم و خیلی مهمه پایه ها رو درست بچینیم

# Importance of Design in terms of *quality*( Cnt'd)

- Software design serves as the foundation for all the software engineering and software support activities that follow.
- Without design, you risk building an unstable system—one that will fail when small changes are made; one that may be difficult to test; one whose quality cannot be assessed until late in the software process, when time is short and many dollars have already been spent.

پس دیزاین پایه ی تولید همه فعالیت های دیگه است  
سیستمی که درست طراحی نشده باشه به راحتی می تونه فاز تست سنگینی داشته باشه  
نرم افزارهای ماندگار نرم افزارهایی که چیدن المان ها درست است --> چون تغییر بعدا راحت و  
تست راحت تره و هزینه اش کمتره



# Software Quality Guidelines and Attributes

- Throughout the design process, the quality of the evolving design is assessed with a series of technical reviews.
- Three characteristics that serve as a guide for the evaluation of a good design as follows.
  - The design should implement all of the explicit requirements contained in the requirements model, and it must accommodate all of the implicit requirements desired by stakeholders.
  - The design should be a readable, understandable guide for those who generate code and for those who test and subsequently support the software.
  - The design should provide a complete picture of the software, addressing the data, functional, and behavioral domains from an implementation perspective.

اگر بخوایم گام هایی داشته باشیم که بفهمیم دیزاینمون دیزاین خوبی است:  
باید همه ریکورمنت هایی توی ریکورمنت مدلمون است بهشون پرداخته شده باشه پس ریکورمنت هایی که صرحتا گفته شده یا ضمنی گفته شده  
یکی هر انچه که کاربر می خواهد باید توی مدل دیزاین بهش پرداخته شده باشه  
باید مدل های طراحی کاملا خوانا و قابل فهم باشن برای افراد و تسترها  
بهتره تسترها از تیم فنی جدا باشن  
همه ابعاد رو گفته باشه --> طراحی باید تصویر کاملی از نرم افزار ارائه دهد که از منظر پیاده سازی به حوزه های داده، عملکردی و رفتاری می پردازد

# *Assessing Design Quality—The Technical Review(TR)*

- A technical review is a meeting conducted by members of the software team.
- Usually two, three, or four people participate depending on the scope of the design information to be reviewed.
- When the meeting commences, the intent is to note all problems with the work product so that they can be corrected before implementation begins.
- At the conclusion of the TR, the review team determines whether further actions are required on the part of the producer before the design work product can be approved as part of the final design model.

بررسی فنی کار:

یک جلسه است که توسط افراد تیم مدل دیزاین رو بررسی میکنن

دو دید داریم:

1- ادم ها قرار نیست زیر سوال برن ینی دنبال مقصر نیستیم

2- همه مشکلات هم نمی تونیم در بیاریم چرا؟ چون هنوز توی فاز دیزاین هستیم و چیزی پیاده سازی نشده

در اول کار هرچی خطای بیشتر در بیاریم بهتره ولی این گارانتی نمیکنه که همه خطاها در اومده

پس در نهایت می گن چه قسمت هایی نیاز به تحول داره

# Quality Guidelines

- A design should exhibit an architecture that
  1. has been created using recognizable architectural styles or patterns,
  2. is composed of components that exhibit good design characteristics,
  3. can be implemented in an evolutionary fashion, thereby facilitating implementation and testing.
- A design should be modular; that is, the software should be logically partitioned into elements or subsystems.
- A design should contain distinct representations of data, architecture, interfaces, and components.
- A design should lead to data structures that are appropriate for the classes to be implemented and are drawn from recognizable data patterns.

-

در حالت کلی:

وقتی که می‌خواهیم ارزیابی کیفیت رو انجام بدیم:

دیزاین ما باید به نحوی باشه که ساختاری قابل قبولی داشته باشه

دیزاین باید ماژولار باشه

دیزاین باید نمایش‌های مختلفی داشته باشه

# Quality Guidelines(Cnt'd)

- A design should lead to components that exhibit independent functional characteristics.
- A design should lead to interfaces that reduce the complexity of connections between components and with the external environment.
- A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis.
- A design should be represented using a notation that effectively communicates its meaning.





# Quality Attributes: FURPS

- *Functionality* is assessed by evaluating the feature set and capabilities of the program, the generality of the functions that are delivered, and the security of the overall system.
- *Usability* is assessed by considering human factors, overall aesthetics, consistency, and documentation.
- *Reliability* is evaluated by measuring the frequency and severity of failure, the accuracy of output results, the mean-time-to-failure (MTTF), the ability to recover from failure, and the predictability of the program.
- *Performance* is measured using processing speed, response time, resource consumption, throughput, and efficiency.
- *Supportability* combines extensibility, adaptability, and serviceability. These three attributes represent a more common term, *maintainability*—and in addition, testability, compatibility, configurability (the ability to organize and control elements of the software configuration), the ease with which a system can be installed, and the ease with which problems can be localized.

یکی از معیارهای کیفیت نرم افزار آینه که عملکرد خودش رو به درستی انجام بده ینی قابلیت هایی مورد انتظار نرم افزار باید برآورده شده باشه: 1

نرم افزار چقدر قابل کار کردن است: 2

قابلیت اطمینان: احتمال خطای سیستم چقدر است ینی چقدر می توان به جواب های یک سیستم اعتماد کرد

پرفرمنس: سرعت پاسخگویی و کار کردن سیستم و مصرف ریسورس ها و...

چقدر می شه نرم افزار رو زنده نگه داشت و چقدر سیستم رو میتونیم راحت استفاده بکنیم...: اخری

# Quality Attributes

- Not every software quality attribute is weighted equally as the software design is developed.
- One application may stress functionality with a special emphasis on security.
- Another may demand performance with particular emphasis on processing speed.
- A third might focus on reliability.
- Regardless of the weighting, it is important to note that these quality attributes must be considered as design commences, *not* after the design is complete and construction has begun.

- همه این ویژگی هایی که گفتیم به یک وزن توی پروژه های مختلف نیستن ولی باید مینیمم هارو داشته باشن همشون

# Design

- Software design encompasses the set of concepts, principles and practices that lead to the development of a high-quality system or product.
- **Design concepts** must be understood before the mechanics of design practice are applied.
- **Design principles** establish an overriding philosophy that guides the design work you must perform.
- **Design practice** itself leads to the creation of various representations of the software that serve as a guide for the construction activity that follows.

یکسری پایه هایی است که اگه روشن متحد باشیم می تونیم بفهمیم کارمون درست بوده  
به طور عمده روی concept اش صحبت میکنیم

# Design Concepts

- Has evolved over the history of software engineering.
- Each provides the software designer with a foundation from which more sophisticated design methods can be applied.
- Each helps you define criteria that can be used to partition software into individual components, separate or data structure detail from a conceptual representation of the software, and establish uniform criteria that define the technical quality of a software design.

- مفهوم دیزاین به طور تکمیل یافته در طول تاریخ مهندسی نرم افزار داره رشد میکنه



# Design Concepts (Cnt'd)

- M. A. Jackson once said: “The beginning of wisdom for a [software engineer] is to recognize the difference between getting a program to work, and getting it right.”
- Fundamental software design concepts provide the necessary framework for “getting it right”.

برای یک مهندس نرم افزار اون میزان برتری کارش نسبت به فنی بودن اینه که تفاوتی که بین کاری که انجام میشه و نحوه انجام کار رو بفهمه به خوبی اون کار انجام بشه  
نرم افزار به درستی کار بکنه خیلی مهم است اینجا

# 1- Abstraction

- When you consider a modular solution to any problem, many levels of abstraction can be posed.
- At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment. At lower levels of abstraction, a more detailed description of the solution is provided. Finally, at the lowest level of abstraction, the solution is stated in a manner that can be directly implemented.
- As different levels of abstraction are developed, you work to create both procedural and data abstractions.

- این اصل:

وقتی داریم رویکرد حل مسئله رو می بینیم برای حل مسائل پیچیده نیاز داریم تا به حدی جزئیات رو کم بکنیم

بعد توی لول های پایین تر این اصل ما میایم جزئیات فنی رو بهش اضافه می کنیم و همینطوری این رویکرد ادامه پیدا میکنه تا اینکه کد رو پیاده بکنیم  
این اصل داره کمک میکنه استپ به استپ بریم جلو

# Types of Abstraction (Cnt'd)

- A *procedural abstraction* refers to a sequence of instructions that have a specific and limited function. The name of a procedural abstraction implies these functions, but specific details are suppressed. An example of a procedural abstraction would be the word *open* for a door. *Open* implies a long sequence of procedural steps (e.g., walk to the door, reach out and grasp knob, turn knob and pull door, step away from moving door, etc.).
- A *data abstraction* is a named collection of data that describes a data object.
- In the context of the procedural abstraction *open*, we can define a data abstraction called **door**. Like any data object, the data abstraction for **door** would encompass a set of attributes that describe the door (e.g., door type, swing direction, opening mechanism, weight, dimensions).



## 2- Architecture

- The overall structure of the software and the ways in which that structure provides conceptual integrity for a system.
- In its simplest form, architecture is the structure or organization of program components (modules), the manner in which these components interact, and the structure of data that are used by the components.
- In a broader sense, components can be generalized to represent major system elements and their interactions.

معماری: چیدن مولفه ها به نحوی که بتونه کل سیستم رو جواب بده  
توی این بحث هدف اینه که بتونیم ساختار اون سیستم رو دربیاریم



## 2- Architecture (Cnt'd)

- One goal of software design is to derive an architectural rendering of a system. This rendering serves as a framework from which more detailed design activities are conducted.
- A set of properties that should be specified as part of an architectural design.
  - *Structural properties* define “the components of a system (e.g., modules, objects, filters) and the manner in which those components are packaged and interact with one another.”
  - *Extra-functional properties* address “how the design architecture achieves requirements for performance, capacity, reliability, security, adaptability, and other system characteristics.
  - *Families of related systems* “draw upon repeatable patterns that are commonly encountered in the design of families of similar systems.”
- Given the specification of these properties, the architectural design can be represented using one or more of a number of different models.

-  
استفاده از سیستم های مشابه

### 3- Patterns

- A pattern is a named nugget of insight which conveys the essence of a proven solution to a recurring problem within a certain context amidst competing concerns”.
- A design pattern describes a design structure that solves a particular design problem within a specific context and amid “forces” that may have an impact on the manner in which the pattern is applied and used.
- The intent of each design pattern is to provide a description that enables a designer to determine
  - (1) whether the pattern is applicable to the current work,
  - (2) whether the pattern can be reused and
  - (3) Whether the pattern can serve as a guide for developing a similar, but functionally or structurally different pattern.

پترن: برای یک مسئله ای یک راه حل اثبات شده ای داریم  
این راه حل اثبات شده در طول زمان که درسته پس می تونه اون مسئله رو حل بکنه  
ایا واقعا پترن خاص برای کار ما مناسب است یا نه؟

# Separation of Concerns

- *Separation of concerns* is a design concept that suggests that any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently.
- A *concern* is a feature or behavior that is specified as part of the requirements model for the software.
- By separating concerns into smaller, and therefore more manageable pieces, a problem takes less effort and time to solve.
- It follows that the perceived complexity of two problems when they are combined is often greater than the sum of the perceived complexity when each is taken separately.
- This leads to a divide-and-conquer strategy—it's easier to solve a complex problem when you break it into manageable pieces. This has important implications with regard to software modularity.
- Separation of concerns is manifested in other related design concepts: modularity, aspects, functional independence, and refinement.

رفتاری که پاسخ یک دغدغه باشد  
ینی که ما بیایم مسئله بزرگ رو تبدیل بکنیم به یکسری فیچرها و.. که قابل مدیریت هستن  
مثل همون مسئله تقسیم و غلبه است  
بعضی وقت ها یک مسئله ای داریم که پارامتریک بهم وابسته اند و اگر بخوایم یک جاشو درست  
بکنیم باید یه جا دیگه هم درست بکنیم اینجا این تفکیک نگرانی خیلی کمک میکنه

## 5- Modularity

- Is the most common manifestation of separation of concerns.
- Software is divided into separately named and addressable components, sometimes called *modules*, that are integrated to satisfy problem requirements.
- It has been stated that “modularity is the single attribute of software that allows a program to be intellectually manageable” .
- Monolithic software (i.e., a large program composed of a single module) cannot be easily grasped by a software engineer. The number of control paths, span of reference, number of variables, and overall complexity would make understanding close to impossible.
- In almost all instances, you should break the design into many modules, hoping to make understanding easier and, as a consequence, reduce the cost required to build the software.

در حالت کلی تقسیم کردن اون نرم افزار به ماژول ها که در حد ممکن از هم مستقل باشن و وقتی که کل این ماژول ها با هم در نظر گرفته میشن کل اون نرم افزار ما پاسخگوی اون نیازها است پس ویژگی است در حالت کی میگه تبدیل بکنیم اون نرم افزار رو به ماژول هایی که با هم در ارتباط هستند و قابل مدیریت هستن

مثلا اگر سینگل ماژول بود اصلا نمی تونیم اونو مدیریت بکنیم ینی هر چی سایز بزرگ باشه مدیریت سخت تر است

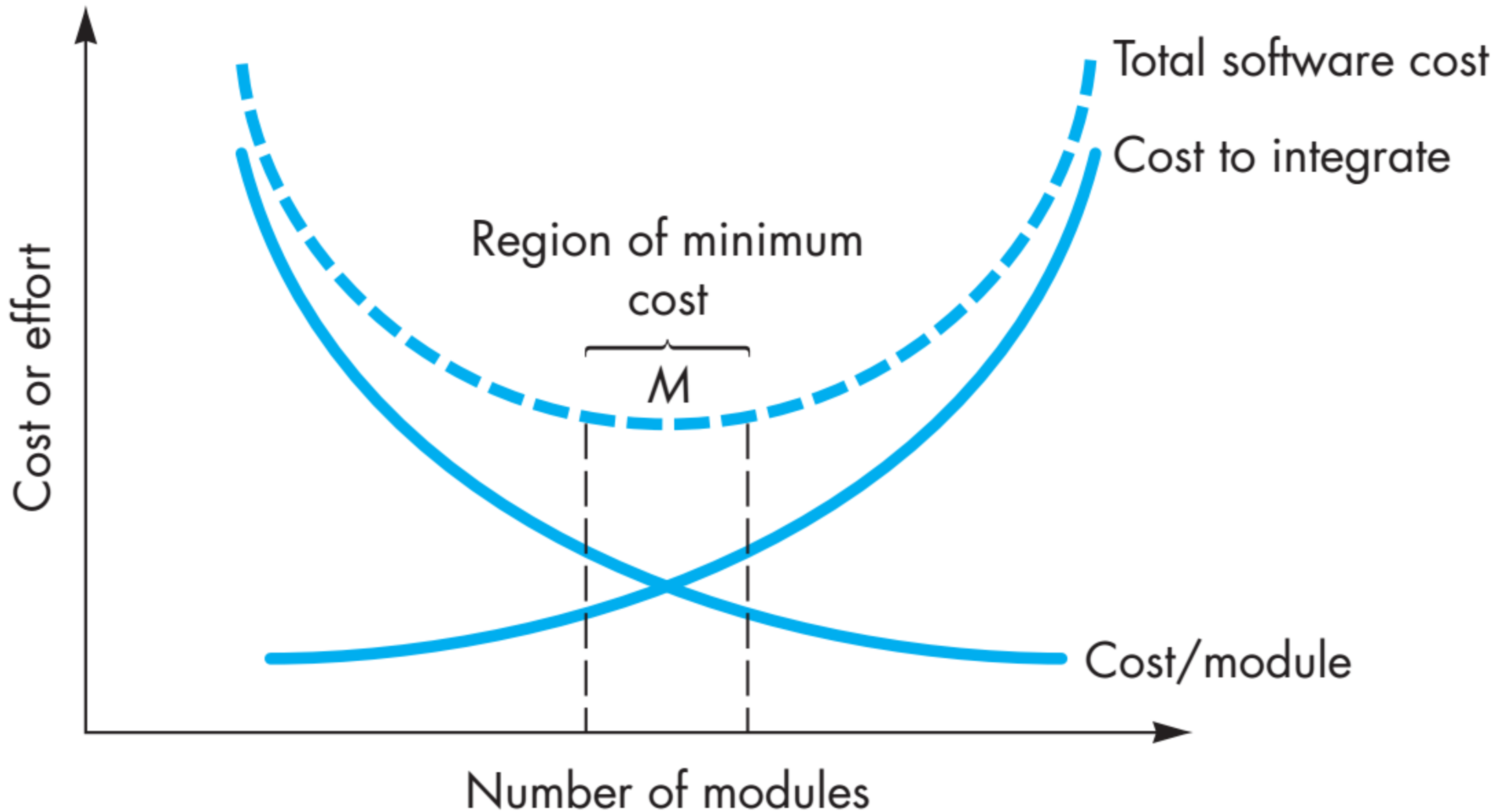
ولی وقتی که ماژولار بود ساده تر اصلاح، توسعه و مدیریت میکنیم و هزینه پشتیبانی از نرم افزار کمتر میشه



## 5- Modularity (Cnt'd)

- Recalling our discussion of separation of concerns, it is possible to conclude that if you subdivide software indefinitely the effort required to develop it will become negligibly small!
- Unfortunately, other forces come into play, causing this conclusion to be (sadly) invalid.
- The effort (cost) to develop an individual software module does decrease as the total number of modules increases.

پس اگر کار رو تقسیم بکنیم به ماژول ها کار ما راحت تر و هزینه کم میشه  
اگر تعداد ماژول ها بی نهایت بشه هزینه میاد پایین --> این جمله اشتباه است ینی از یه جایی به بعد  
اگ تعداد ماژول ها بیشتر بشه هزینه بیشتری برای ما خواهد داشت



هرچی ماژول ها دارن اضافه میشن هزینه ای که داریم میدیم بیشتر است چون هماهنگی بین ماژول ها، سینک شدن این ها بیشتر است و همینطور اصلاح خطاها و تغییرات بیشتر است ولی هرچی ماژول هم بیشتر بشه هزینه تولید هر ماژول کمتر میشه چون ماژول ها ریزتر میشن و حوزه مسئولیتشون کمتر میشه

اینجا یک مینیمم هزینه ای رو باید دربیاریم:

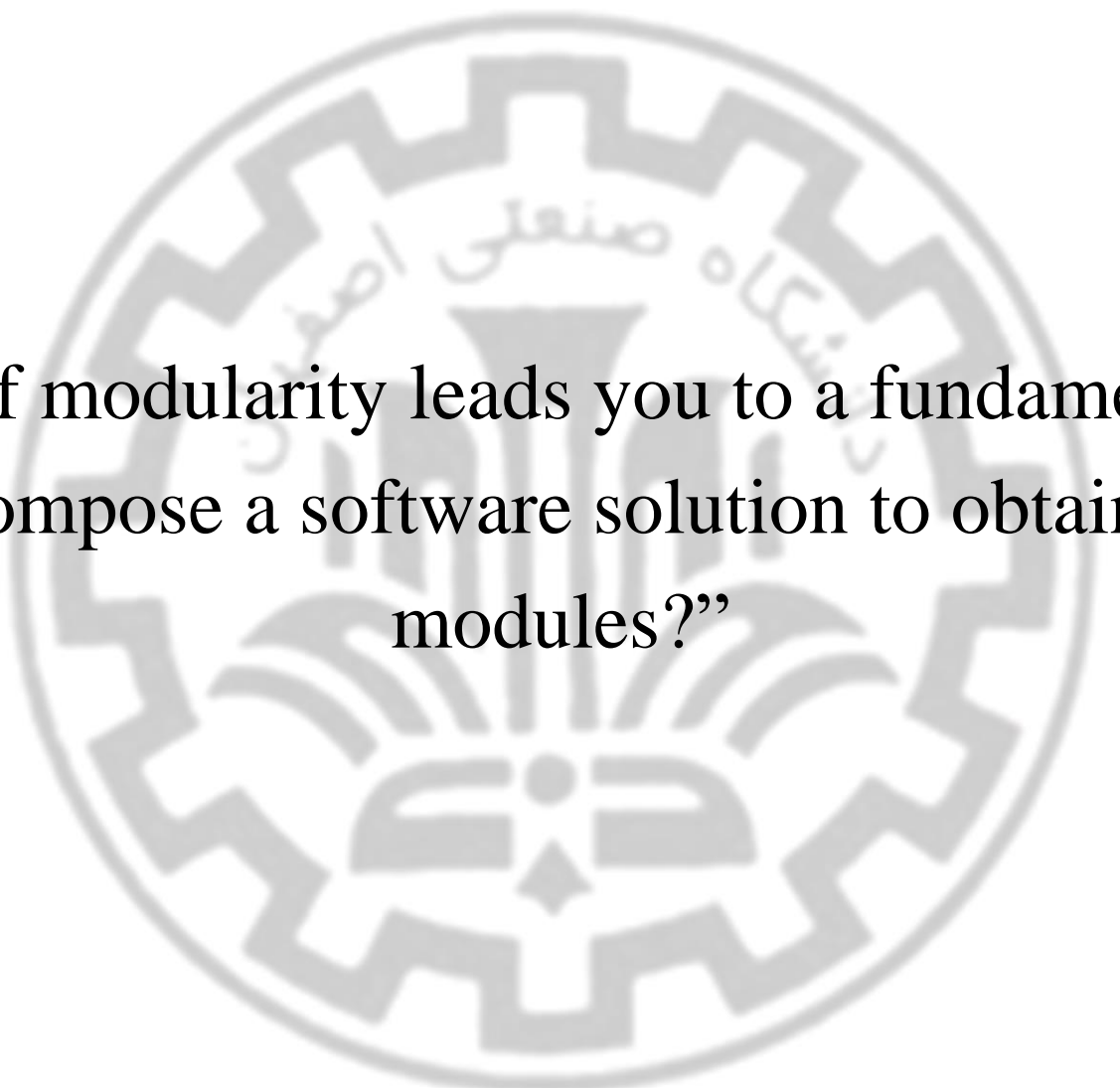
اولایل کار وقتی که داریم کار رو انجام میدیم تعداد ماژول ها اگر تا یه حدی برسه هزینه تولید ماژول خیلی بالا نیست و هزینه **integrate** هم کمه بعد اگر تعداد ماژول ها زیاد بشه درسته که هزینه تولید ماژول داره کم میشه ولی نه اونقدر کم که غلبه بکنه به هزینه **integrate** پس یک منطقه ای است که نه اونقدر کم تعداد خیلی زیاد بشه و **integrate** نمایی هزینه برای ما داشته باشه و نه اونقدر هم زیاد که مسئله اصلا قابل حل نباشه پس یه چیزی این وسط می خوایم

## 5- Modularity (Cnt'd)

- Given the same set of requirements, more modules means smaller individual size. However, as the number of modules grows, the effort (cost) associated with integrating the modules also grows.
- There is a number,  $M$ , of modules that would result in minimum development cost, but we do not have the necessary sophistication to predict  $M$  with assurance.
- You should modularize, but care should be taken to stay in the vicinity of  $M$ .
- Undermodularity or overmodularity should be avoided.
- You modularize a design (and the resulting program) so that development can be more easily planned; software increments can be defined and delivered; changes can be more easily accommodated; testing and debugging can be conducted more efficiently, and long-term maintenance can be conducted without serious side effects.

-

اگر ریکورمنت های مشخصی داشته باشیم هرچی ماژول ها بیشتر باشه سائز هر کدوم کمتر میشه  
پس تعداد ماژول ها که بیشتر بشه integrat هم زیاد میشه  
M میشه همون مینیمم



The concept of modularity leads you to a fundamental question:  
“How do I decompose a software solution to obtain the best set of  
modules?”





## 6- Information Hiding

- The principle of *information hiding* suggests that modules be “characterized by design decisions that (each) hides from all others.”
- In other words, modules should be specified and designed so that information (algorithms and data) contained within a module is inaccessible to other modules that have no need for such information.
- Hiding implies that effective modularity can be achieved by defining a set of independent modules that communicate with one another only that information necessary to achieve software function.

-  
ینی ماژول ها تا حد ممکن از هم دیگه بی خبر باشن ینی تصمیماتی که داره گرفته میشه از بقیه پنهان باشه ینی اینکه تصمیمات غیرقابل دسترس باشه توسط بقیه ماژول ها

## 6- Information Hiding (Cnt'd)

- Abstraction helps to define the procedural (or informational) entities that make up the software.
- Hiding defines and enforces access constraints to both procedural detail within a module and any local data structure used by the module.
- The use of information hiding as a design criterion for modular systems provides the greatest benefits when modifications are required during testing and later during software maintenance.
- Because most data and procedural detail are hidden from other parts of the software, inadvertent errors introduced during modification are less likely to propagate to other locations within the software.

- اینجا abstraction هم به ما کمک میکنه

مهمترین مزیتش اینه که: هرچی ماژول ها از هم دیگه بی خبر باشن تغییرات یکی بردیگری تاثیری نمی ذاره و این ینی اینکه تست ما راحتتره و ریکآوری سیستم بهتر انجام میگیره  
توی این تغییراتی که ما میدیم خطاها به وجود میان --> هرچی که این اصل 6 بیشتر باشه احتمال اینکه خطاهایی که بعد از اعمال تغییرات توی سیستم میاد جدی باشه کمه

## 7- Functional Independence

- Is a direct outgrowth of separation of concerns, modularity, and the concepts of abstraction and information hiding.
- Is achieved by developing modules with “single minded” function and an “aversion” to excessive interaction with other modules.
- You should design software so that each module addresses a specific subset of requirements and has a simple interface when viewed from other parts of the program structure.
- It is fair to ask why independence is important. Software with effective modularity, that is, independent modules, is easier to develop because function can be compartmentalized and interfaces are simplified.
- Independent modules are easier to maintain (and test) because secondary effects caused by design or code modification are limited, error propagation is reduced, and reusable modules are possible.
- To summarize, functional independence is a key to good design, and design is the key to software<sup>43</sup> quality.

- اینجا به نوعی دیگه داره به مسئله نگا میکنه:

تا حد ممکن مازول ها سینگل مایندد باشن ینی از نظر عملکردی مستقل باشن و تعامل کمی با

بقیه مازول ها داشته باشن

یک مجموعه مرتبط از ریکورمنت ها رو بیاد سرویس دهی بکنه و این اینترفیس ساده ای داشته

باشه وقتی که بقیه المان ها دارن بهش نگاه میکنن

## 7- Functional Independence (Cnt'd)

- Independence is assessed using two qualitative criteria: cohesion and coupling. *Cohesion* is an indication of the relative functional strength of a module. *Coupling* is an indication of the relative interdependence among modules.
- Cohesion is a natural extension of the information-hiding concept. A cohesive module performs a single task, requiring little interaction with other components in other parts of a program. Stated simply, a cohesive module should (ideally) do just one thing. Although you should always strive for high cohesion (i.e., single-mindedness), it is often necessary and advisable to have a software component perform multiple functions. However, “schizophrenic” components (modules that perform many unrelated functions) are to be avoided if a good design is to be achieved.
- Coupling is an indication of interconnection among modules in a software structure. Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface. In software design, you should strive for the lowest possible coupling. Simple connectivity among modules results in software that is easier to understand and less prone to a “ripple effect”<sup>44</sup> caused when errors occur at one location and propagate throughout a system.

این اصل دوتا مفهوم داره:

**cohesion:** میزان انسجام و میزان یکی بودن عملیات یک ماژول است و کارهای مشابه انجام

دادن --> این در حالت ایده ال: ینی این که یک کار انجام بده

**coupling:** اتصال و ارتباطی که مولفه ها با هم دیگه دارن --> هرچی این مینیمم باشه بهتر است

البته که نمی تونه صفر باشه چون اگر صفر باشه سیستم ما جزیره ای میشه ینی باید این ماژول ها با

هم در ارتباط باشن که سرویس اصلی رو به جواب برسونن ولی باید مینیمم باشه



## 8-Refinement

- Is a top-down design strategy.
- An application is developed by successively refining levels of procedural detail.
- A hierarchy is developed by decomposing a macroscopic statement of function (a procedural abstraction) in a stepwise fashion until programming language statements are reached.
- Refinement is actually a process of *elaboration*. You begin with a statement of function (or description of information) that is defined at a high level of abstraction. That is, the statement describes function or information conceptually but provides no indication of the internal workings of the function or the internal structure of the information. You then elaborate on the original statement, providing more and more detail as each successive refinement (elaboration) occurs.

-  
ینی کم کم جزئیات رو اضافه کردن --> استپ به استپ جزئیات رو بهش اضافه بکنیم



**Abstraction** and **refinement** are complementary concepts.

*Abstraction* enables you to specify procedure and data internally but suppress the need for “outsiders” to have knowledge of low-level details.

*Refinement* helps you to reveal low-level details as design progresses.

Both concepts allow you to create a complete design model as the design evolves.

abstraction , refinement دوتا مفهوم مکمل هم هستن

توی Ab.. میایم سطح انتظار رو بالاتر می بریم ینی نیازه که برای اول کار مسئله رو حل بکنیم و جزئیات رو مخفی نگه میداریم و از نگاه بیرون و re... به ما کمک میکنه که جزئیات رو اضافه بکنیم برای اینکه بتونیم کم کم برسیم به پیاده سازی و هر دو تا مولفه نیاز هستن ولی در جای خاص خودش

## 9- Aspects

- As requirements analysis occurs, a set of “concerns” is uncovered. These concerns “include requirements, use cases, features, data structures, quality-of-service issues, variants, intellectual property boundaries, collaborations, patterns and contracts”.
- Ideally, a requirements model can be organized in a way that allows you to isolate each concern (requirement) so that it can be considered independently.
- In practice, however, some of these concerns span the entire system and cannot be easily compartmentalized.
- As design begins, requirements are refined into a modular design representation.



## 9- Aspects (Cnt'd)

- Consider two requirements,  $A$  and  $B$ . Requirement  $B$  *crosscuts* requirement  $A$  “if a software decomposition [refinement] has been chosen in which  $A$  cannot be satisfied without taking  $B$  into account”.
- As design refinement occurs,  $A^*$  is a design representation for requirement  $A$  and  $B^*$  is a design representation for requirement  $B$ . Therefore,  $A^*$  and  $B^*$  are representations of concerns, and  $B^*$  *crosscuts*  $A^*$ .
- **An *aspect* is a representation of a crosscutting concern.** Therefore, the design representation,  $B^*$ , is an aspect of the system.
- In an ideal context, an aspect is implemented as a separate module (component) rather than as software fragments that are “scattered” or “tangled” throughout many components. To accomplish this, the design architecture should support a mechanism for defining an aspect—a module that enables the<sup>48</sup> concern to be implemented across all other concerns that it crosscuts.

-  
مثال: دوتا ریکورمنت داریم  $a, b$  و اگر اینو بشکنیم به یکسری مازول ها  $-->$  ریکورمنت  $a$  رو نمی تونیم بدون برآورده شدن  $b$  داشته باشیم  $-->$  این میشه مال مدل تحلیل وقتی میاد توی طراحی جزئیات بیشتری بهش اضافه میشه که  $a$  میشه  $a^*$  و  $b$  هم میشه  $b^*$   
حالا این  $b^*$  میشه aspect سیستم  
ینی بررسی بکنیم کدوم یک از این ریکورمنت ها می تونن مستقل باشن و کدوماش امکان استقلال براشون وجود نداره



# 10- Refactoring

- Is a reorganization technique that simplifies the design (or code) of a component without changing its function or behavior.
- Is the process of changing a software system in such a way that it does not alter the external behavior of the code [design] yet improves its internal structure.”
- When software is refactored, the existing design is examined for redundancy, unused design elements, inefficient or unnecessary algorithms, poorly constructed or inappropriate data structures, or any other design failure that can be corrected to yield a better design.
- For example, a first design iteration might yield a component that exhibits low cohesion (i.e., it performs three functions that have only limited relationship to one another). After careful consideration, you may decide that the component should be refactored into three separate components, each exhibiting high cohesion. The result will be software that is easier to integrate, easier to test, and easier to maintain.
- Although the intent of refactoring is to modify the code in a manner that does not alter its external behavior, inadvertent side effects can and do occur.

-  
ینی تغییر مکانیزیم های داخلی یک ماژول بدون اینکه توی رفتار خارجی اثر بذاره

# 11- Design Classes

- The analysis model defines a set of analysis classes. Each of these classes describes some element of the problem domain, focusing on aspects of the problem that are user visible. The level of abstraction of an analysis class is relatively high.
- As the design model evolves, you will define a set of *design classes* that refine the analysis classes by providing design detail that will enable the classes to be implemented, and implement a software infrastructure that supports the business solution.



# Characteristics of a good class

- **Complete and sufficient.** A design class should be the **complete** encapsulation of all attributes and methods that can reasonably be expected (based on a knowledgeable interpretation of the class name) to exist for the class. **Sufficiency** ensures that the design class contains only those methods that are sufficient to achieve the intent of the class, no more and no less.
- **Primitiveness.** Methods associated with a design class should be focused on accomplishing one service for the class. Once the service has been implemented with a method, the class should not provide another way to accomplish the same thing.

خصوصیت یک کلاس خوب چیه؟

اول کامل باشه ینی چیزی از قلم نیوفتاده باشه ینی چیزی کم نداشته باشن  
دوم کافی باشه ینی کلاس های دیزاین نباید چیزی اضافه تر داشته باشن  
ینی نه چیزی کم باشه و نه چیزی زیاد

کم بودن خلی قابل لمس و خلی راحت می تونیم بفهمیم یه چیزی کم است  
ولی چیزی اضافه تر نداشته باشه خلی قابل لمس نیست

وقتی که میخوایم یک مسئله رو حل بکنیم توسط یک متد باید به اون جواب بدیم ینی فقط یک راه حل  
برای اون متد باید وجود داشته باشه ینی کلاس ما نباید راه حل های مختلفی براش داشته باشه ینی  
نباید متدهایی باشن که به طور موازی جواب یک مسئله باشن ینی انحصاری باید کار کنن -->  
1- اسییی که می زنه عدم قطعیت توی مسئله بالا می ره و 2- اگر خطایی روی اون مسئله رخ بده  
نمی دونیم کدوم مسیر اجرا بوده که باعث خطا شده

پس متدها فقط تنها جوابگوی اون مسئله باشن

متدهای موازی نباید داشته باشیم برای اینکه بتونن یک مسئله رو جواب بدن پس کلاسی خوبه که نه  
دیتایی اضافه بگیره و نه دیتایی کم و نه راه حل های موازی برای یک مسئله داشته باشه

## Characteristics of a good class(Cnt'd)

- **High cohesion.** A cohesive design class has a small, focused set of responsibilities and single-mindedly applies attributes and methods to implement those responsibilities.
- **Low coupling.** Within the design model, it is necessary for design classes to collaborate with one another. However, collaboration should be kept to an acceptable minimum.
  - If a design model is highly coupled (all design classes collaborate with all other design classes), the system is difficult to implement, to test, and to maintain over time.
  - In general, design classes within a subsystem should have only limited knowledge of other classes.

کلاسی خوبه که cohesion بالایی داشته باشه و coupling اش هم کم باشه

کلاسی ک منسجم باشه ینی یک مسوولیتی رو انجام بده و روی مسؤلیت هایی خودش متمرکز میشه و جوابگوی کارش هم باید باشه ینی اگر خطایی رخ بده می دونیم منشاش کجاست کاپلینگ کلاس ها باید کم باشه ینی اگر قراره کلاس ها با هم ارتباطی داشته باشن ارتباط باید در حد مینیمم باشه و اگر مدل دیزاین جوری باشه که کلاس ها خیلی با هم کولپریشن و ارتباط داشته باشن همه چی در هم تنیده است و همه چی سخته و نقص اصل ماژولاریته است



# Cohesion

- Refers to how single-minded a module (class, object, or method) is within a system. A class or object should represent only one thing, and a method should solve only a single task.
  - *Method cohesion* addresses the cohesion within an individual method. Methods should do one and only one thing.
  - *Class cohesion* is the level of cohesion among the attributes and methods of a class. A class should represent only one thing.

تعریف کوهیژن یا انسجام معنیش همیشه

تعریف نرم افزایش: ینی چقدر یک ماژول داره یک کار رو انجام میده مثلاً چندتا سرویس رو بتونه  
سایورت بکنه داره نقض میکنه این اصل کوهیژن رو  
کوهیژن دو دسته میشه:

در سطح یک فانکشن بررسی کنیم عملکرد اون فانکشن رو که چقدر منسجم است ینی فقط یک کار  
انجام بده : متد کوهیژن

یکی در سطح کلاس: ینی یک مسئولیت به اون متد یا کلاس بسپاریم --> اگر متده در حد ممکن یک  
کار انجام بده و اگر کلاس است در حد ممکن یک چیز رو بیان کنه

# Types of method cohesion

Level	Type	Description
Good ↓	Functional	A method performs a single problem-related task (e.g., calculate current GPA).
	Sequential	The method combines two functions in which the output from the first one is used as the input to the second one (e.g., format and validate current GPA).
	Communicational	The method combines two functions that use the same attributes to execute (e.g., calculate current and cumulative GPA).
	Procedural	The method supports multiple weakly related functions. For example, the method could calculate student GPA, print student record, calculate cumulative GPA, and print cumulative GPA.
	Temporal or Classical	The method supports multiple related functions in time (e.g., initialize all attributes).
	Logical	The method supports multiple related functions, but the choice of the specific function is chosen based on a control variable that is passed into the method. For example, the called method could open a checking account, open a savings account, or calculate a loan, depending on the message that is sent by its calling method.
Bad	Coincidental	The purpose of the method cannot be defined or it performs multiple functions that are unrelated to one another. For example, the method could update customer records, calculate loan payments, print exception reports, and analyze competitor pricing structure.

Source: These types are based on material from Page-Jones, *The Practical Guide to Structured Systems*; Myers, *Composite/Structured Design*; Edward Yourdon and Larry L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design* (Englewood Cliffs, NJ: Prentice-Hall, 1979).

متد کوهیژن:

بهترین حالتش اینه که فانکشنال باشه ینی هر متدی یک کاری انجام بده  
از بالا به پایین ینی داره بدتر میشه

ینی متد ما دوتا کار انجام میده ینی یک کار رو اول انجام میده و بعد می ره سراغ کاری بعدی یینی  
یک فانکشن ک کارش تموم شد خروجیش میره توی فانکشن بعدی **se**..

**CO**..ینی متد ما دوتا کار انجام میده --> دوتا کاره ولی صفت هاش یکیه مثلا معدل ترم و معدل  
دانشکده --> اینا دوتا کار متفاوت هستن ولی ورودی دوتاشون یکی است

**p**.. ینی کارهای کم مرتبط ضعیف داریم انجام میدیم مثلا همه کارهای مرتبط با دانشجو مثلا مدل  
ترمی یا.. ینی چندتا کاره ولی مرتبط با دانشجو


ینی کارها خیلی نامرتبطه ولی ارتباطی که باهم دارن ارتباط زمانی است : **te**..

**lo**.. خیلی توصیه نمیشه و ینی از نظر منطقی با هم ارتباط دارن ینی بر مبنای یک ورودی میایم  
مسیر اجرای متد رو عوض میکنیم مثلا

**CO**.. این بدترین است ینی یه عالمه متد داریم که اصلا مرتبط با هم نیستن یا دوبه دو بهم مرتبط اند

از بالا تا یه جایی می ریم پایین ولی دیگه بعد از اون نمی تونیم ادامه بدیم واسه همین گفتیم تا حد  
ممکن

# Types of class cohesion

Level	Type	Description
Good 	Ideal	The class has none of the mixed cohesions.
	Mixed-Role	The class has one or more attributes that relate objects of the class to other objects on the same layer (e.g., the problem domain layer), but the attribute(s) has nothing to do with the underlying semantics of the class.
	Mixed-Domain	The class has one or more attributes that relate objects of the class to other objects on a different layer. As such, they have nothing to do with the underlying semantics of the thing that the class represents. In these cases, the offending attribute(s) belongs in another class located on one of the other layers. For example, a port attribute located in a problem domain class should be in a system architecture class that is related to the problem domain class.
	Mixed-Instance	The class represents two different types of objects. The class should be decomposed into two separate classes. Typically, different instances only use a portion of the full definition of the class.
Worse		

Based upon material from Page-Jones, *Fundamentals of Object-Oriented Design in UML*.

در سطح کلاس داریم اینو بررسی میکنیم  
حالت ایده ال: هر کلاسی بیانگر یک ابجکت باشه و در یک لایه تعریف شده باشه

ابجکت های کلاس و صفت های کلاس روی یک لایه هستن ولی صفت ها بیانگر ابجکت های دیگه هم هستن <-- m..

ینی توی کلاس صفت های کلاسی رو تعریف میکنیم که مرتبط با لایه های مختلف هستن <-- 3

ینی صفت های ابجکت هم بیانگر کلاس های متفاوت باشن و هم بیانگر لایه های متفاوت باشن <--  
بدترین حالت

# Coupling

- Refers to how interdependent or interrelated the modules (classes, objects, and methods) are in a system.
- The higher the interdependency, the more likely changes in part of a design can cause changes to be required in other parts of the design.
  - *Interaction coupling* deals with the coupling among methods and objects through message passing.
  - *Inheritance coupling*, deals with how tightly coupled the classes are in an inheritance hierarchy.

توی کاپلینگ ما نگاهمون درون ماژولی نیست بلکه نگاهمون، نگاه بین ماژول میشه ینی ارتباط ماژول ها چیه ینی چقدر باید بینشون پیام و رد و بدل بشه هرچه وابستگی ماژول ها بیشتر باشه تاثیر تغییر هر ماژول نسبت به یک ماژول دیگه بیشتر میشه دو نوع داریم:

- 1: ارتباط هایی که بین اجکت ها است
- 2: در قالب ارتباطات ارث بری است

تا حد ممکن باید کاپلینگمون کم باشه



# Types of interaction coupling

Level	Type	Description
Good ↓ Bad	No Direct Coupling	The methods do not relate to one another; that is, they do not call one another.
	Data	The calling method passes a variable to the called method. If the variable is composite (i.e., an object), the entire object is used by the called method to perform its function.
	Stamp	The calling method passes a composite variable (i.e., an object) to the called method, but the called method only uses a portion of the object to perform its function.
	Control	The calling method passes a control variable whose value will control the execution of the called method.
	Common or Global	The methods refer to a “global data area” that is outside the individual objects.
	Content or Pathological	A method of one object refers to the inside (hidden parts) of another object. This violates the principles of encapsulation and information hiding. However, C++ allows this to take place through the use of “friends.”

Source: These types are based on material from Meilir Page-Jones, *The Practical Guide to Structured Systems Design*, 2nd Ed. (Englewood Cliffs, NJ: Yardon Press, 1988); Glenford Myers, *Composite/Structured Design* (New York: Van Nostrand Reinhold, 1978).

-  
حالت ایده ال اینه که ارتباط نداشته باشن

حالت بعدی --> یک مقداری رو از یک ابجکتی پاس میدیم به یک ابجکت دیگه

3: اینم همون دیتا رو می فرسته ولی یک بخشی از اون دیتا رو استفاده میکنه

4: اینجا هم دوتا ابجکتی هستن که یک متغیری بینشون پاس داده میشه که اون متغیر یک متغیر کنترلی است

5: ینی یک دیتای گلوبالی رو داشته باشیم که همه ازش استفاده میکنن

6: ینی به درون ابجکت ها بتونیم دسترسی داشته باشیم

## 12- Design for Test

- There is an ongoing chicken-and-egg debate about whether software design or test case design should come first.
- Advocates of test-driven development (TDD) write tests before implementing any other code.
- “Test fast, fail fast, adjust fast.”

جز یکی از رویکردهای خیلی نو است

ینی اول برنامه رو بنویسیم و بعد تست بکنیم

یا اول تست رو بنویسیم و بعد برنامه رو

**TDD:** ینی قبل از اینکه اصلا وارد پیاده سازی بشیم ساختار پیاده سازی برمحوریت تسته ینی اول میگیریم چه تست کیس هایی میخوایم و بر مبنای اون تست کیس ها پیاده سازی رو انجام میدیم

ینی اول تست کیس ها رو طراحی بکنیم و بعد کد رو بزنیم

# Evolve problem domain-oriented analysis models into optimal solution domain-oriented design models

- Object-oriented systems development is both incremental and iterative.
- In this time we begin looking at the models of the problem domain through a design lens.
- In this step, we make modifications to the problem domain models that will enhance the efficiency and effectiveness of the evolving system.
  - Factoring,
  - Partitions and collaborations,
  - Layers

به چه نحوی باید این مدل هارو بیاریم توی مدل های طراحی به نحوی که بهینه باشه؟

اول باید استپ به استپ بریم جلو یی نمیتونیم بگیم پلن اولیه بهترین پلن است و این باید توی چندتا راند اتفاق بیوفته

نوع نگاه ما اینجا نوع نگاه طراحی است پس اینجا یه سری تغییرات به اون مدل های تحلیل میدیم به طوری که هزینه کل نرم افزار در طول عمرش مینیمم باشه

# Factoring

- Is the process of separating out a *module* into a stand-alone module. The new module can be a new *class* or a new *method*. For example, when reviewing a set of classes, it may be discovered that they have a similar set of attributes and methods. Thus, it might make sense to factor out the similarities into a separate class.

- اگر عملیاتی وجود داشته باشد که توی چندتا مازول تکراریه میایم یک مازول جدیدی می سازیم



# Partitions and Collaborations

- The actual size of the system representation can overload the user and the developer. At this point in the evolution of the system, it might make sense to split the representation into a set of *partitions*.
- A partition is the object-oriented equivalent of a subsystem, where a subsystem is a decomposition of a larger system into its component systems.
- From an object-oriented perspective, partitions are based on the pattern of activity (messages sent) among the objects in an object-oriented system.

پارتیشن یک قسمتی از سیستم  
سیستم خیلی بزرگه --> اینجا برای اینکه بتونیم کار رو انجام بدیم پارتیشن بندی میکنیم سیستم رو  
ینی مدل های تحلیل رو در آوردیم حالا میایم دسته بندی میکنیم

# Partitions and Collaborations(Cnt'd)

- A good place to look for potential partitions is the *collaborations* modeled in UML's communication diagrams.
- One useful way to identify collaborations is to create a communication diagram for each use case. However, because an individual class can support multiple use cases, an individual class can participate in multiple use-case-based collaborations. In cases where classes are supporting multiple use cases, the collaborations should be merged.
- The class diagram should be reviewed to see how the different classes are related to one another.

The greater the coupling between classes, the more likely the classes should be grouped together in a collaboration or partition.

-  
این communication دیاگرام است که کمک میکند به ما که بتوانیم این پارتیشن هایی که بالقوه  
توی نرم افزار هستن رو بشناسیم  
پارتیشن بر مبنای فعالیت بین خودشون تعریف میشن  
قانون این است: هر چه کاپلینگ بین کلاسها بیشتر باشه یعنی هرچی کلاس های ارتباط بیشتری با  
همدیگه داشته باشن یا باید اون دوتا کلاس یکی بشه یا اینکه توی یک پارتیشن قرار بگیره

# The general rule

The more messages sent between objects, the more likely the objects belong in the same partition.

The fewer messages sent, the less likely the two objects belong together.

پس قانون کلی این شد که

هرچی مسیج های بیشتری بین کلاس ها رد و بدل بشه به احتمال زیاد این ابجکت ها باید یک پارتیشن قرار بگیرن و هرچه تعداد مسیج کمتر باشه ابجکت ها می تونن توی پارتیشن های مختلفی قرار بگیرن

# Another useful approach to identifying potential partitions

- Model each collaboration between objects in terms of clients, servers, and contracts.
- A *client* is an instance of a *class* that sends a *message* to an instance of another class for a *method* to be executed;
- A *server* is the instance of a class that receives the message;
- *Contract* is the specification that formalizes the interactions between the client and server objects.
- Allows the developer to build up potential partitions by looking at the contracts that have been specified between objects.

The more contracts there are between objects, the more likely that the objects belong in the same partition. The fewer contracts, the less chance there is that the two classes belong in the same partition.

رویکرد دیگه ای که برای اینکه بتونیم پارتیشن هارو در بیاریم وجود داره: اینه که بتونیم کلاس ها و ارتباط بین اون هارو در قالب کلاینت و سرور ببینیم

کلاسی که سند میکنه یک فانکشن رو از یک کلاس دیگه میشه کلاینت و کلاسی که پاسخ میده به اون پیام میشه سرور



# Reference

- **Dennis, Wixon, Tegarden**, “System Analysis and Design, An Object Oriented Approach with UML”, 5th Edition, 2015.
- **R. G. Pressman, B. R. Maxim**, Software Engineering\_ A Practitioner’s Approach, 8th Edition, 2014.

