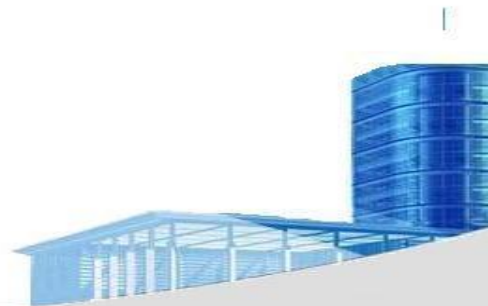




Ch.6 Human Aspects of Software Engineering

March 18, 2024



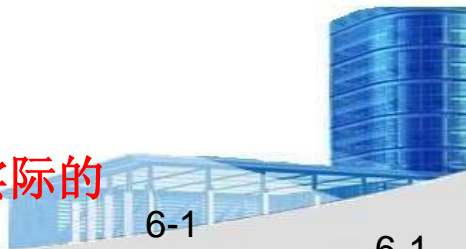


6.1 Characteristics of A Software Engineer

- Traits of Successful Software Engineers



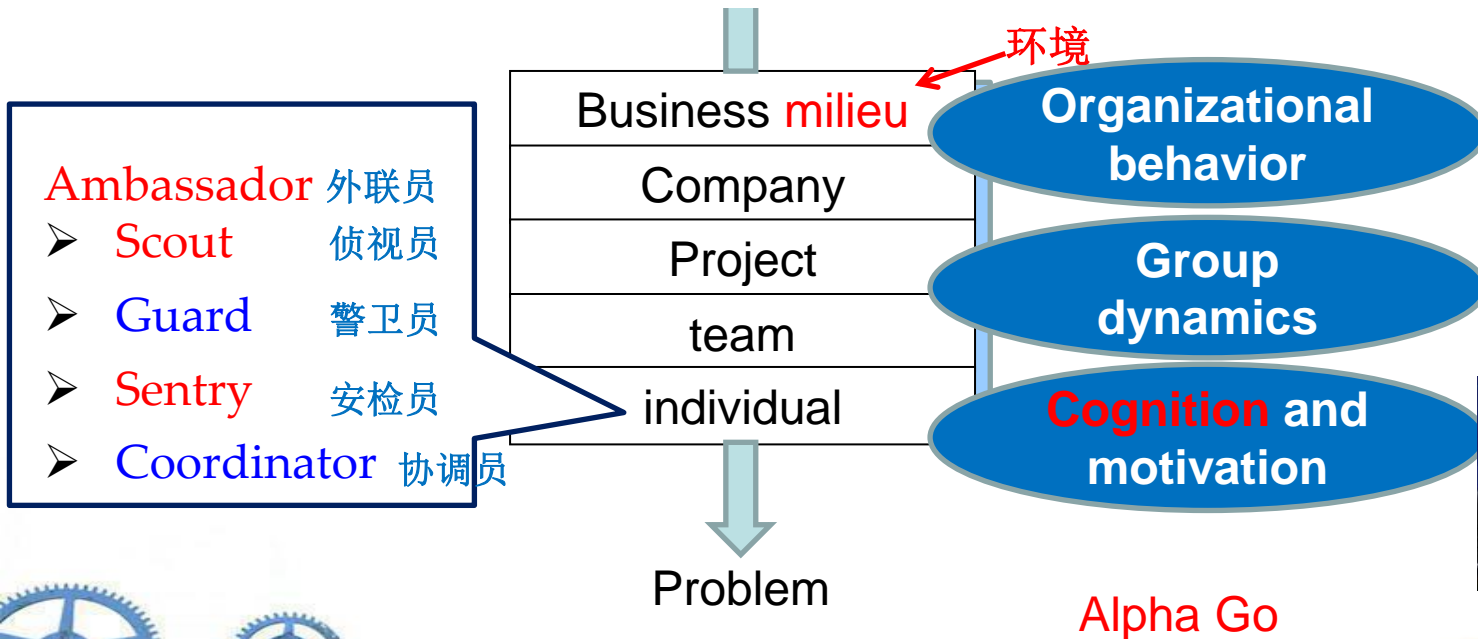
- Sense of individual responsibility
- **Acute**ly aware of the needs of team members and stakeholders
- **Brutally** honest about design flaws and offers constructive **criticism**
- **Resilient** (有弹性的、抗压的) under pressure
- Heightened sense of fairness
- Attention to detail
- **Pragmatic** ← 务实的, 实际的





6.2 The Psychology Of Software Engineering

- Behavioral Model for Software Engineering
- Boundary **Spanning** (跨越) Team Roles



拉里·佩奇和谢尔盖·布林



Google (googol)

DeepMind

Alpha Go

德米斯·哈萨比斯
(Demis Hassabis)



Alpha Go

黄世杰(人肉臂)



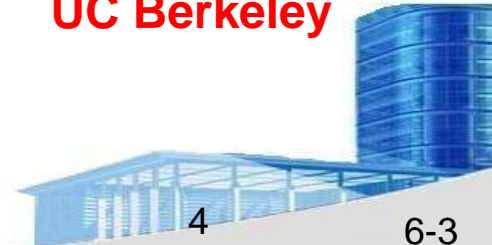
6.3 The Software Team

- Effective Software Team Attributes

- Sense of purpose
- Sense of **involvement**
- Sense of trust
- Sense of **improvement**
- **Diversity** of team member skill sets



UC Berkeley

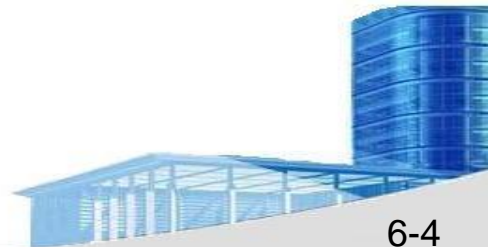




6.5 Agile Teams

- Generic Agile Teams

- Stress individual competency coupled with group collaboration as critical success factors
- People trump process and politics can trump people
- Agile teams as self-organizing and have many structures
 - An adaptive team structure
 - Uses elements of Constantine's random, open, and synchronous structures
 - Significant autonomy
- Planning is kept to a minimum and constrained only by business requirements and organizational standards
(Ex. Daily meeting for Scrum)





6.5 Agile Teams

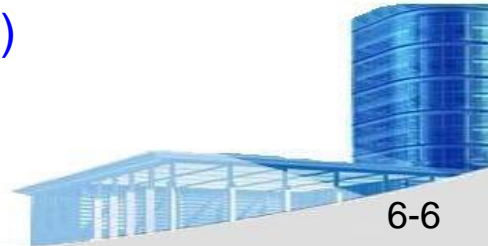
- XP Team Values
 - **Communication** – close informal (**verbal**, 口头的) communication among team members and stakeholders and establishing meaning for **metaphors** (象征, 隐喻) as part of continuous feedback;
 - **Simplicity** – design for immediate needs nor future needs
 - **Feedback** – derives from the implemented software, the customer, and other team members
 - **Courage** – the **discipline** to resist pressure to design for unspecified future requirements
 - **Respect** – among team members and stakeholders





6.6 Impact of Social Media

- **Blogs** – can be used share information with team members and customers
- **Microblogs** – allow posting of real-time messages to individuals following the poster (e.g. Twitter)
- **Targeted on-line forums** – allow participants to post questions or opinions and collect answers
- **Social networking sites**– allows connections among software developers for the purpose of sharing information (e.g. Facebook, 人人, LinkedIn, QQ, WeChat, Dingding, etc)
- **Social book marking**– allow developers to keep track of and share web-based resources (e.g. Delicious, Stumble, CiteULike)





6.7 Software Engineering using the Cloud

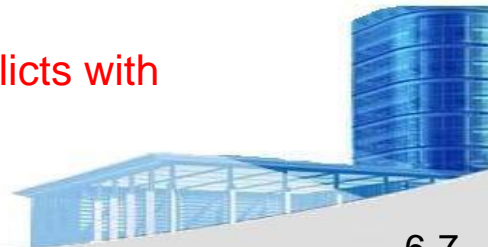
- **Benefits**

- Provides **access** to all software engineering work products
- Removes **device dependencies** and available every where
- Provides **avenues** (途径) for distributing and testing software
- Allows software engineering information developed by one member to be available to all team members



- **Concerns**

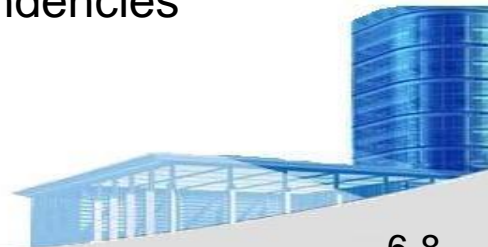
- **Dispersing** (分布式) cloud services (e.g. 阿里云) outside the control of the software team may present **reliability** and **security risks**
- Potential for **interoperability problems** becomes high with large number of services distributed on the cloud
- Cloud services stress usability and performance which often **conflicts with** security, privacy, and reliability





6.8 Collaboration Tools

- Services of Collaborative Development Environments(CDEs)
 - **Namespace** that allows secure, private storage or work products
 - **Calendar** for coordinating project events
 - **Templates** that allow team members to create **artifacts** (加工品) that have common look and feel
 - **Metrics support** to allow quantitative assessment of each team member's contributions
 - **Communication analysis** to track messages and isolates patterns that may imply issues to resolve
 - **Artifact** (加工品) **clustering** showing work product dependencies
Ex. Github, Gitlab?



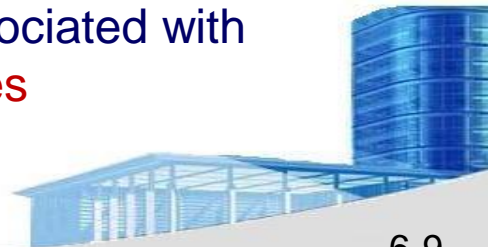


6.9 Global Teams

- Team Decisions Making Complications

Q2: What's the factors effecting **Global Software Development**?

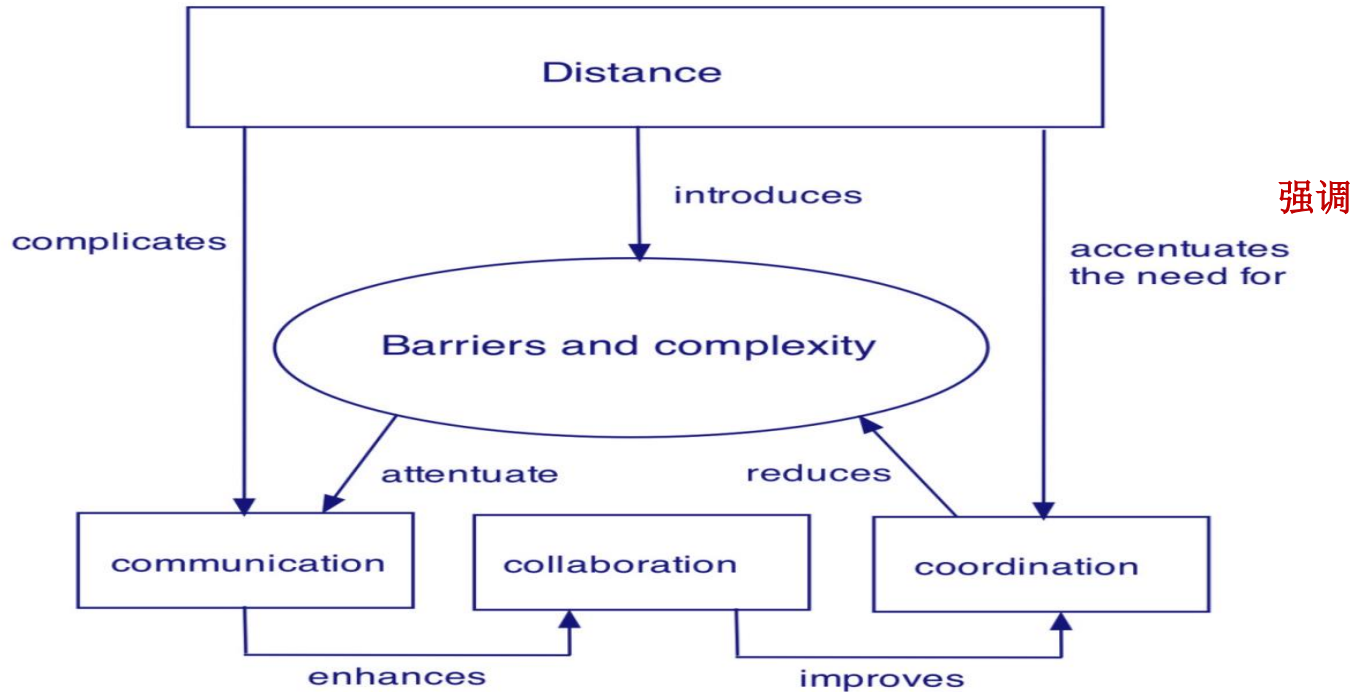
- Problem **complexity**
- **Uncertainty and risk** associated with the decision
- Work associated with decision has **unintended effect** on another project object (law of unintended consequences)
- **Different views** of the problem lead to different conclusions about the way forward
- **Global software teams** face additional challenges associated with collaboration, coordination, and **coordination difficulties**





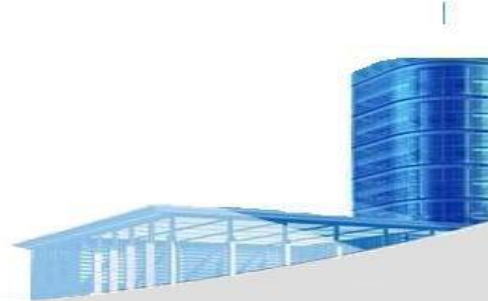
6.9 Global Teams

- Factors Affecting Global Software Development Team





Ch.7 Principles that Guide Practice

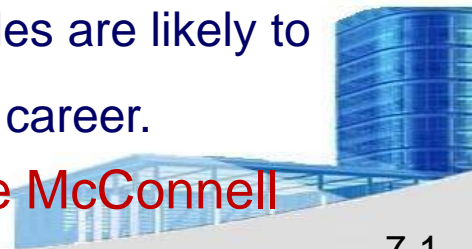




• Software Engineering Knowledge

- You often hear people say that software development knowledge has a **3-year half-life**: half of what you need to know today will be **obsoleted** (废弃的,过时的) within 3 years. In the domain of technology-related knowledge, that's probably about right. But there is **another kind** of software development knowledge—a kind that I think of as "**software engineering principles**"—that does not have a three-year half-life. These software engineering principles are likely to serve a professional programmer **throughout** his or her career.

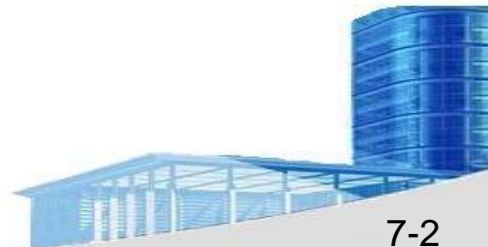
---Steve McConnell





Kinds of Principles

- Principles that Guide **Process** (8);
- Principles that Guide **Practice**(8);
- **Communication** Principles(10); *****
- **Planning** Principles(10);
- **Modeling** Principles (10);
 - Agile** Modeling Principles(10);*****
 - Requirements** Modeling Principles(5);
 - Design** Modeling Principles(10);
 - Living** Modeling Principles(8); *****
- **Construction** Principles:
 - Coding** Principles(9);
 - Preparation** Principles(5);
 - Validation** Principles(3);
 - Testing** Principles(9);
- **Deployment** Principles(5);
- **Work Practices** (10)





• Communication Principles - I

- *Principle #1. Listen.* Try to focus on the speaker's words, rather than formulating your response to those words.
- *Principle # 2. Prepare before you communicate.* Spend the time to understand the problem before you meet with others.
- *Principle # 3. Someone should facilitate (促进) the activity.*
 - (1) Every communication meeting should have a leader (a facilitator) to keep the conversation moving in a productive direction;
 - (2) to **mediate** any conflict that does occur, and
 - (3) to ensure than other principles are followed.
- *Principle #4. Face-to-face communication is best.* But it usually works better when some other representation of the relevant information is present.





• Communication Principles - II

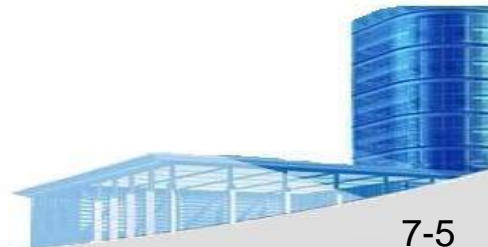
- *Principle # 5. Take notes and document decisions.* Someone participating in the communication should serve as a “recorder” and write down all important points and decisions.
- *Principle # 6. Strive for collaboration.* Collaboration and consensus(一致、合意) occur when the collective knowledge of members of the team is combined ...
- *Principle # 7. Stay focused, modularize your discussion.* The more people involved in any communication, the more likely that discussion will bounce from one topic to the next.
- *Principle # 8. If something is unclear, draw a picture.*
- *Principle # 9. (a) Once you agree to something, move on; (b) If you can't agree to something, move on; (c) If a feature or function is unclear and cannot be clarified at the moment, move on.*
- *Principle # 10. Negotiation is not a contest or a game. It works best when both parties win.*





• Agile Modeling Principles - I

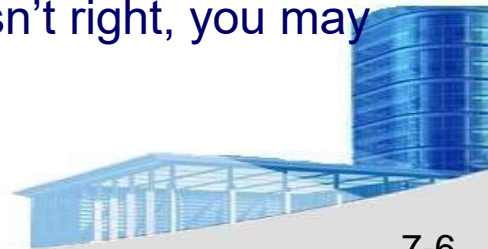
- Principle #1. The primary goal of the software team is to **build** software not create models.
- Principle #2. **Travel light** – don't create more models than you need.
- Principle #3. Strive to produce **the simplest model** that will describe the problem or the software.
- Principle #4. Build models in a way that makes them **amenable** to change.
- Principle #5. Be able to state an **explicit purpose** for each model that is created.





• Agile Modeling Principles - II

- Principle #6. Adapt the models you create to the system **at hand** (现有的, 手头的).
- Principle #7. Try to build **useful** models, **forget** about building **perfect** models.
- Principle #8. Don't become **dogmatic** about model syntax. Successful communication is key.
- Principle #9. If your **instincts** tell you a paper model isn't right, you may have a reason to be concerned.
- Principle #10. Get **feedback** as soon as you can.





• Principles that Guide Process - I

- *Principle #1. Be agile.* Whether the process model you choose is prescriptive or agile, the basic tenets of agile development should govern your approach.
- *Principle #2. Focus on quality at every step.* The exit condition for every process activity, action, and task should focus on the quality of the work product that has been produced.
- *Principle #3. Be ready to adapt.* Process is not a religious experience and dogma has no place in it. When necessary, adapt your approach to constraints imposed by the problem, the people, and the project itself.
- *Principle #4. Build an effective team.* Software engineering process and practice are important, but the bottom line is people. Build a self-organizing team that has mutual trust and respect.





• Principles that Guide Process - II

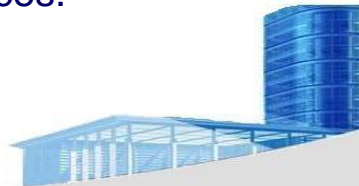
- *Principle #5. Establish mechanisms for communication and coordination.* Projects fail because important information falls into the cracks and/or stakeholders fail to coordinate their efforts to create a successful end product.
- *Principle #6. Manage change. Focus on quality at every step.* The approach may be either formal or informal, but mechanisms must be established to manage the way changes are requested, assessed, approved and implemented.
- *Principle #7. Assess risk.* Lots of things can go wrong as software is being developed. It's essential that you establish contingency plans.
- *Principle #8. Create work products that provide value for others.* Create only those work products that provide value for other process activities, actions or tasks.





- **Principles that Guide Practice - I**

- *Principle #1. Divide and conquer.* Stated in a more technical manner, analysis and design should always emphasize separation of concerns (SoC).
- *Principle #2. Understand the use of abstraction.* At its core, an abstraction is a simplification of some complex element of a system used to communicate meaning in a single phrase.
- *Principle #3. Strive for consistency.* A familiar context makes software easier to use.
- *Principle #4. Focus on the transfer of information.* Pay special attention to the analysis, design, construction, and testing of interfaces.





• Principles that Guide Practice - II

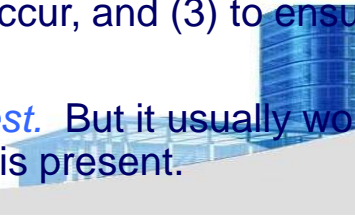
- *Principle #5. Build software that exhibits effective modularity.* Separation of concerns (Principle #1) establishes a philosophy for software. Modularity provides a mechanism for realizing the philosophy.
- *Principle #6. Look for patterns.* Brad Appleton [App00] suggests that: “The goal of patterns within the software community is to create a body of literature to help software developers resolve recurring problems encountered throughout all of software development.
- *Principle #7. When possible, represent the problem and its solution from a number of different perspectives.*
- *Principle #8. Remember that someone will maintain the software.*





- **Communication Principles - I**

- *Principle #1. Listen.* Try to focus on the speaker's words, rather than formulating your response to those words.
- *Principle #2. Prepare before you communicate.* Spend the time to understand the problem before you meet with others.
- *Principle #3. Someone should facilitate the activity.* Every communication meeting should have a leader (a facilitator) to keep the conversation moving in a productive direction; (2) to mediate any conflict that does occur, and (3) to ensure that other principles are followed.
- *Principle #4. Face-to-face communication is best.* But it usually works better when some other representation of the relevant information is present.





• Communication Principles - II

- *Principle # 5. Take notes and document decisions.* Someone participating in the communication should serve as a “recorder” and write down all important points and decisions.
- *Principle # 6. Strive for collaboration.* Collaboration and consensus occur when the collective knowledge of members of the team is combined ...
- *Principle # 7. Stay focused, modularize your discussion.* The more people involved in any communication, the more likely that discussion will bounce from one topic to the next.
- *Principle # 8. If something is unclear, draw a picture.*
- *Principle # 9. (a) Once you agree to something, move on; (b) If you can't agree to something, move on; (c) If a feature or function is unclear and cannot be clarified at the moment, move on.*
- *Principle # 10. Negotiation is not a contest or a game. It works best when both parties win.*





• Planning Principles - I

- *Principle #1. Understand the scope of the project.* It's impossible to use a roadmap if you don't know where you're going. Scope provides the software team with a destination.
- *Principle #2. Involve the customer in the planning activity.* The customer defines priorities and establishes project constraints.
- *Principle #3. Recognize that planning is iterative.* A project plan is never engraved in stone. As work begins, it very likely that things will change.
- *Principle #4. Estimate based on what you know.* The intent of estimation is to provide an indication of effort, cost, and task duration, based on the team's current understanding of the work to be done.





• Planning Principles - II

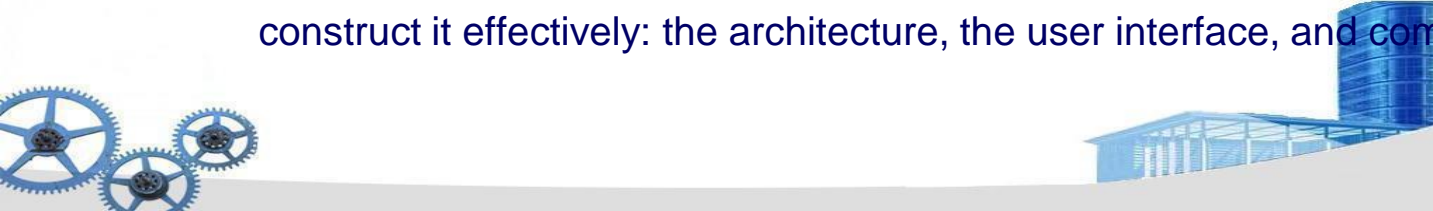
- *Principle #5. Consider risk as you define the plan.* If you have identified risks that have high impact and high probability, contingency planning is necessary.
- *Principle #6. Be realistic.* People don't work 100 percent of every day.
- *Principle #7. Adjust granularity as you define the plan.* Granularity refers to the level of detail that is introduced as a project plan is developed.
- *Principle #8. Define how you intend to ensure quality.* The plan should identify how the software team intends to ensure quality.
- *Principle #9. Describe how you intend to accommodate change.* Even the best planning can be obviated by uncontrolled change.
- *Principle #10. Track the plan frequently and make adjustments as required.* Software projects fall behind schedule one day at a time.





• Modeling Principles

- *In software engineering work, two classes of models can be created:*
 - *Requirements models (also called analysis models)* represent the customer requirements by depicting the software in three different domains: the information domain, the functional domain, and the behavioral domain.
 - *Design models* represent characteristics of the software that help practitioners to construct it effectively: the architecture, the user interface, and component-level detail.





• Agile Modeling Principles - I

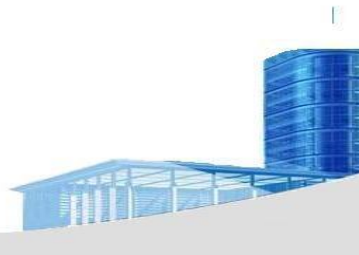
- Principle #1. The primary goal of the software team is to build software not create models.
- Principle #2. Travel light – don't create more models than you need.
- Principle #3. Strive to produce the simplest model that will describe the problem or the software.
- Principle #4. Build models in a way that makes them amenable to change.
- Principle #5. Be able to state an explicit purpose for each model that is created.





• Agile Modeling Principles - II

- Principle #6. Adapt the models you create to the system at hand.
- Principle #7. Try to build useful models, forget about building perfect models.
- Principle #8. Don't become dogmatic about model syntax. Successful communication is key.
- Principle #9. If your instincts tell you a paper model isn't right you may have a reason to be concerned.
- Principle #10. Get feedback as soon as you can.





• Requirements Modeling Principles

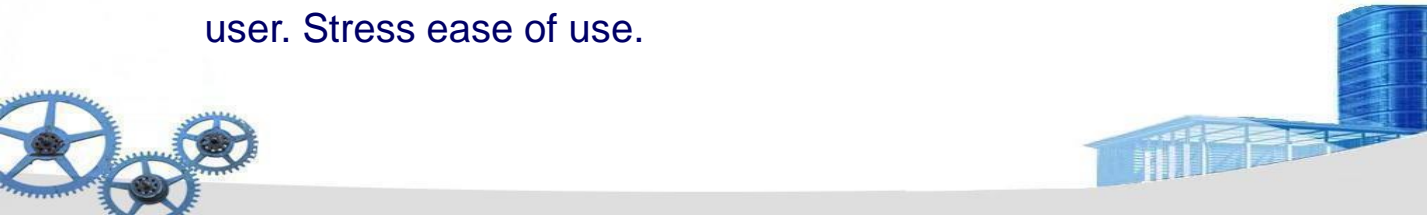
- Principle #1. The information domain of a problem must be represented and understood.
- Principle #2. The functions that the software performs must be defined.
- Principle #3. The behavior of the software (as a consequence of external events) must be represented.
- Principle #4. The models that depict information, function, and behavior must be partitioned in a manner that uncovers detail in a layered (or hierarchical) fashion.
- Principle #5. The analysis task should move from essential information toward implementation detail.





• Design Modeling Principles - I

- Principle #1. Design should be traceable to the requirements model.
- Principle #2. Always consider the architecture of the system to be built.
- Principle #3. Design of data is as important as design of processing functions.
- Principle #4. Interfaces (both internal and external) must be designed with care.
- Principle #5. User interface design should be tuned to the needs of the end-user. Stress ease of use.





• Design Modeling Principles - II

- Principle #6. Component-level design should be functionally independent.
- Principle #7. Components should be loosely coupled to each other than the environment.
- Principle #8. Design representations (models) should be easily understandable.
- Principle #9. The design should be developed iteratively.
- Principle #10. Creation of a design model does not preclude using an agile approach.





• Living Modeling Principles - I

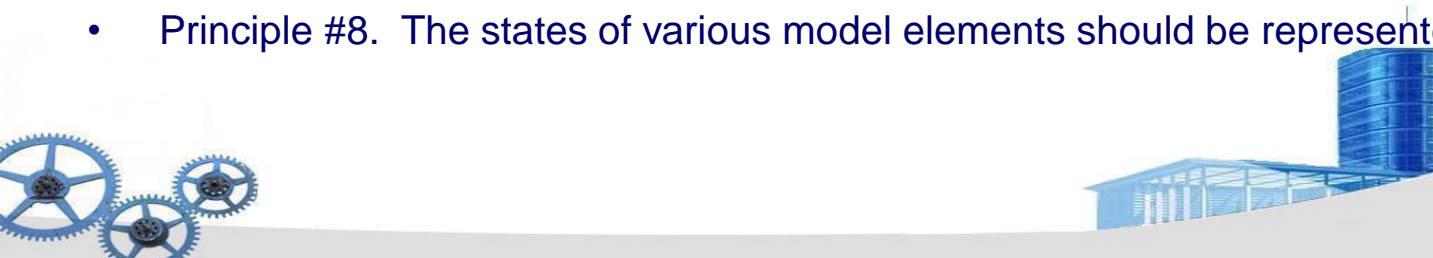
- Principle #1. Stakeholder-centric models should target specific stakeholders and their tasks.
- Principle #2. Models and code should be closely coupled.
- Principle #3. Bidirectional information flow should be established between models and code.
- Principle #4. A common system view should be created.





• Living Modeling Principles - II

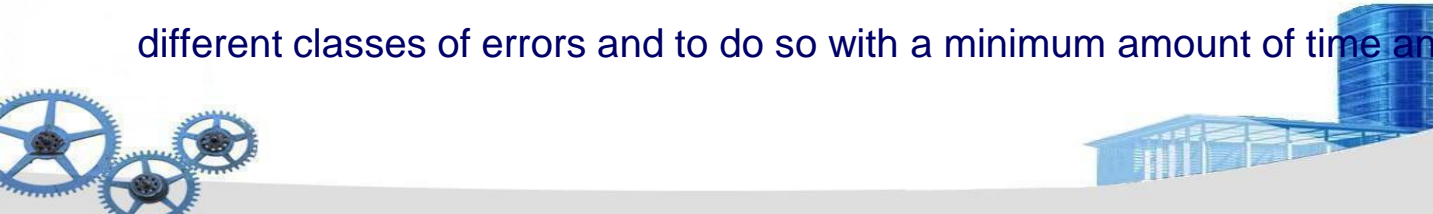
- Principle #5. Model information should be persistent to allow tracking system changes.
- Principle #6. Information consistency across all model levels must be verified.
- Principle #7. Each model element has assigned stakeholder rights and responsibilities.
- Principle #8. The states of various model elements should be represented.





• Construction Principles

- The construction activity encompasses a set of coding and testing tasks that lead to operational software that is ready for delivery to the customer or end-user.
- *Coding principles and concepts* are closely aligned programming style, programming languages, and programming methods.
- *Testing principles and concepts* lead to the design of tests that systematically uncover different classes of errors and to do so with a minimum amount of time and effort.





• Preparation Principles

• *Before you write one line of code, be sure you:*

- Understand of the problem you're trying to solve.
- Understand basic design principles and concepts.
- Pick a programming language that meets the needs of the software to be built and the environment in which it will operate.
- Select a programming environment that provides tools that will make your work easier.
- Create a set of unit tests that will be applied once the component you code is completed.





• Coding Principles

- *As you begin writing code, be sure you:*
 - Constrain your algorithms by following structured programming [Boh00] practice.
 - Consider the use of pair programming
 - Select data structures that will meet the needs of the design.
 - Understand the software architecture and create interfaces that are consistent with it.
 - Keep conditional logic as simple as possible.
 - Create nested loops in a way that makes them easily testable.
 - Select meaningful variable names and follow other local coding standards.
 - Write code that is self-documenting.
 - Create a visual layout (e.g., indentation and blank lines) that aids understanding.





- **Validation Principles**

- *After you've completed your first coding pass, be sure you:*

- Conduct a code walkthrough when appropriate.
- Perform unit tests and correct errors you've uncovered.
- Refactor the code.





- **Testing Principles - I**

- *Al Davis [Dav95] suggests the following:*

- Principle #1. All tests should be traceable to customer requirements.
- Principle #2. Tests should be planned long before testing begins.
- Principle #3. The Pareto principle applies to software testing.
- Principle #4. Testing should begin “in the small” and progress toward testing “in the large.”





• Testing Principles - II

- *Al Davis [Dav95] suggests the following:*
 - Principle #5. Exhaustive testing is not possible.
 - Principle #6. Testing effort for each system module commensurate to expected fault density.
 - Principle #7. Static testing can yield high results.
 - Principle #8. Track defects and look for patterns in defects uncovered by testing.
 - Principle #9. Include test cases that demonstrate software is behaving correctly.





• **Deployment Principles**

- Principle #1. Customer expectations for the software must be managed.
- Principle #2. A complete delivery package should be assembled and tested.
- Principle #3. A support regime must be established before the software is delivered.
- Principle #4. Appropriate instructional materials must be provided to end-users.
- Principle #5. Buggy software should be fixed first, delivered later.





Ch.8 Understanding Requirements





Requirements Engineering (Ex. Safehome)

- **Inception**(起始)—ask a set of questions that establish ...
 - **basic understanding** of the problem
 - the people **who want** a solution
 - the nature of the solution that is desired, and
 - the **effectiveness** of preliminary **communication** and **collaboration** between the customer and the developer
- **Elicitation** (引出) —elicit requirements from all stakeholders
- **Elaboration** (详尽阐述) —create an **analysis model** that identifies data, function and behavioral requirements
- **Negotiation**—agree on a **deliverable system** that is **realistic** for developers and customers

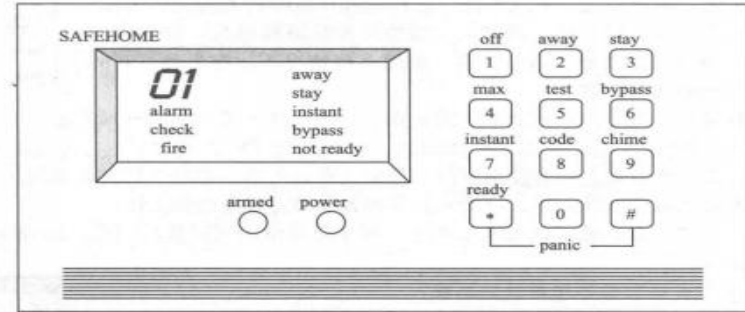
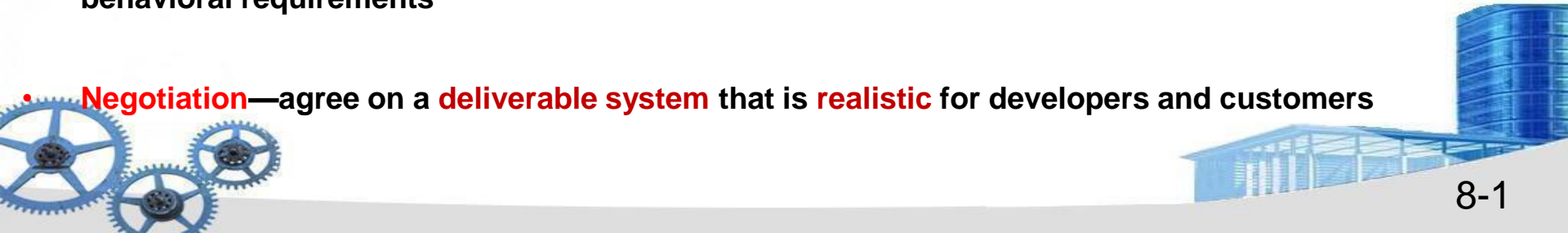


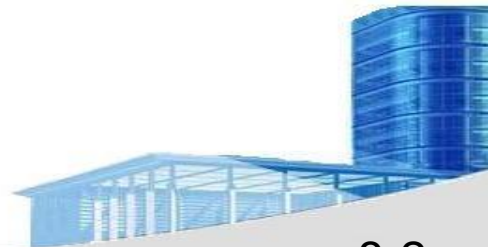
图 7-1 SafeHome 控制面板





Requirements Engineering (cont.)

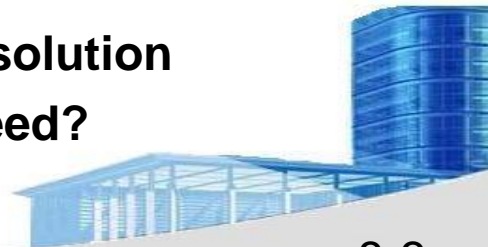
- **Specification (规格说明)**—can be any one (or more) of the following
 - A written **document**
 - A set of **models**
 - A formal **mathematical**
 - A collection of user scenarios (**use-cases**)
 - A **prototype**
- **Validation**—a review mechanism that looks for
 - **errors** in content or interpretation
 - areas where **clarification** (说明) may be required
 - **missing** information
 - **inconsistencies** (a major problem when large products or systems are engineered)
 - **conflicting or unrealistic** (unachievable) requirements.
- **Requirements management**





Inception

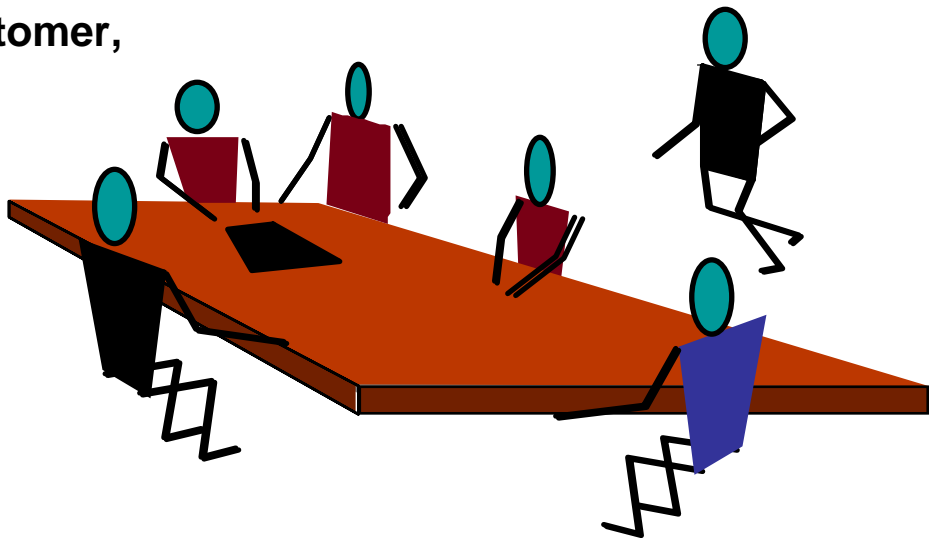
- Identify **stakeholders**(利益相关者)
 - “who else do you think I should talk to?”
- Recognize **multiple** points of **view**
- Work toward **collaboration**
- The first set of context-free questions
 - Who is **behind** the request for this work?
 - Who will **use** the solution?
 - What will be the **economic benefit** of a successful solution
 - Is there **another source** for the solution that you need?





Eliciting Requirements

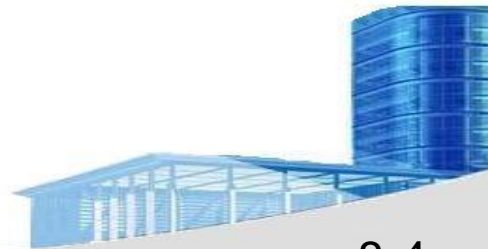
- meetings are conducted and attended by both software engineers and customers
- rules for preparation and participation are established
- an **agenda** (议事日程) is suggested
- a “**facilitator**” (主持人, can be a customer, a developer, or an outsider) controls the meeting





Eliciting Requirements (cont.)

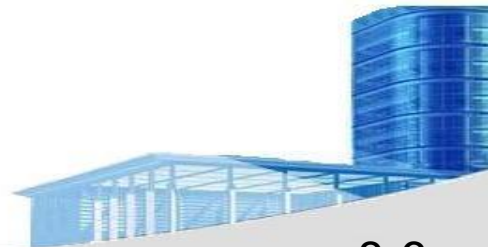
- a “**definition mechanism**” (can be **work sheets** (工作表), flip charts (活动挂图), or **wall stickers** or an electronic bulletin board, chat room or virtual forum) is used
- the **goal** is
 - to identify the problem
 - propose elements of the solution
 - negotiate different approaches, and
 - specify a preliminary set of solution requirements





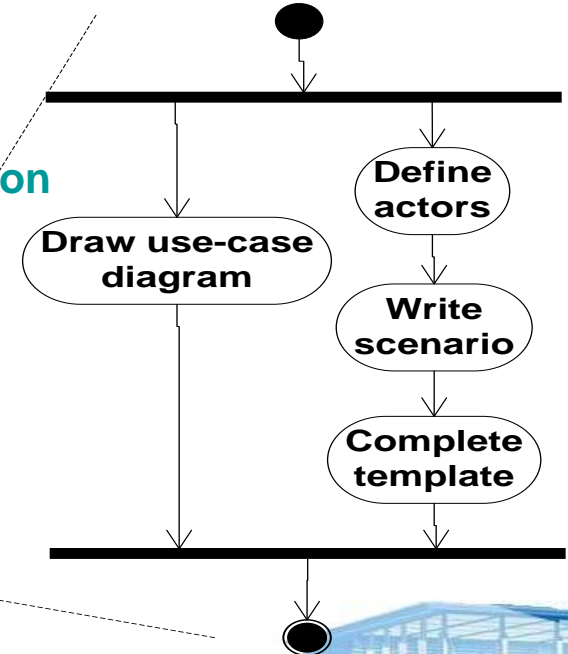
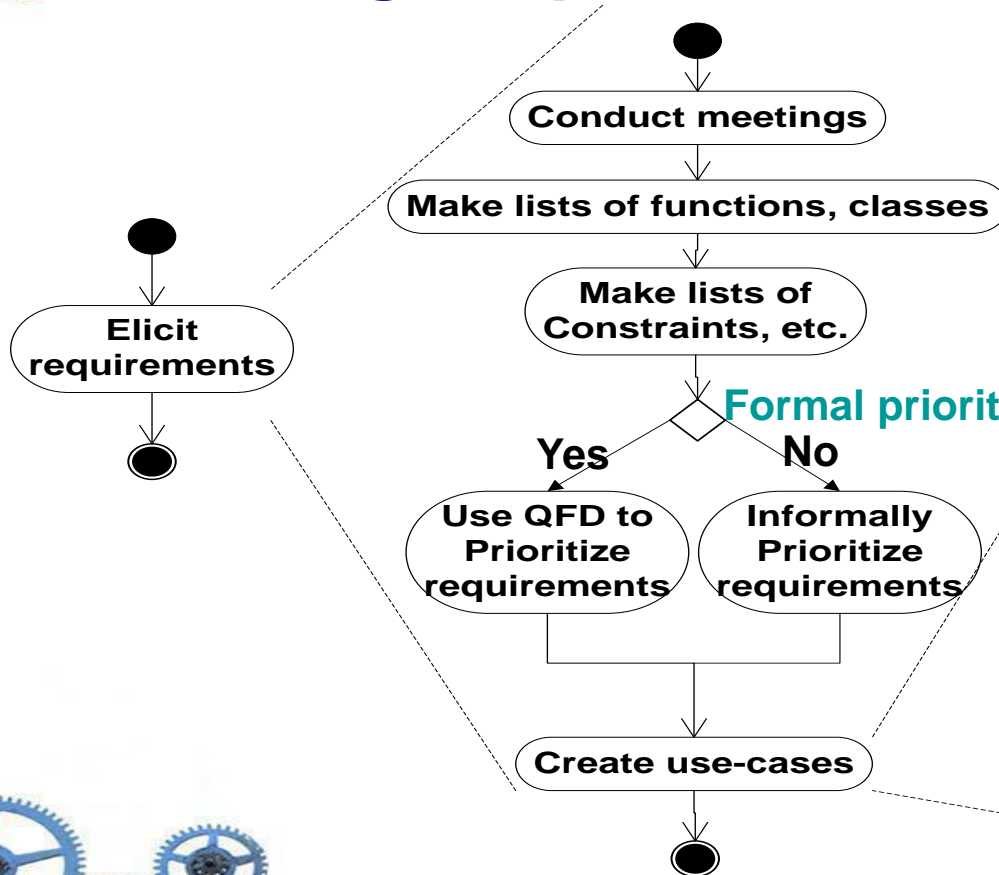
Quality Function Deployment (QFD, 质量功能部署)

- **Function deployment** determines the “value” (as perceived by the customer) of each function required of the system **Information deployment identifies** data objects and events
- **Task deployment examines** the behavior of the system
- **Value analysis** determines the **relative priority** of requirements
(1、沉浸感的多人社交场景 / 2、虚拟旅游业 / 3、虚拟资产交易)
- It identifies **three** types of requirements
 - **Normal** requirements
 - **Expected** requirements
 - **Exciting** requirements





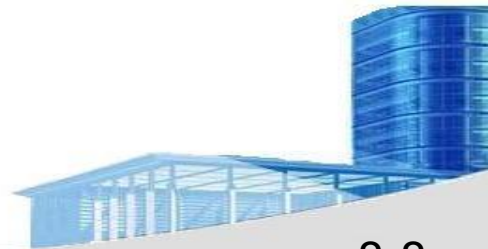
Eliciting Requirements





Non-Functional Requirements

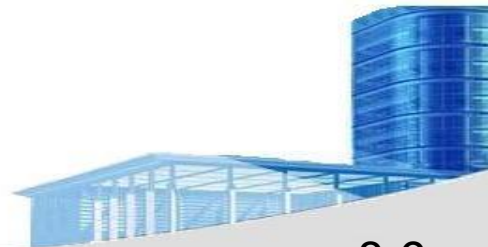
- **Non-Functional Requirements (NFR)** – **quality** attribute, **performance** attribute, **security** attribute, or general system **constraint**. A two phase process is used to determine which NFR's are compatible:
 - **The first phase** is to create a **matrix** using each **NFR** as a **column** (列) **heading** and the system **SE guidelines** (准则) a **row** (行) **labels**
 - **The second phase** is for the team to **prioritize** each NFR using a set of decision rules to decide which to implement by classifying each NFR and guideline pair as **complementary** (补充的), **overlapping** (重叠的), **conflicting**, or **independent**





Elicitation Work Products

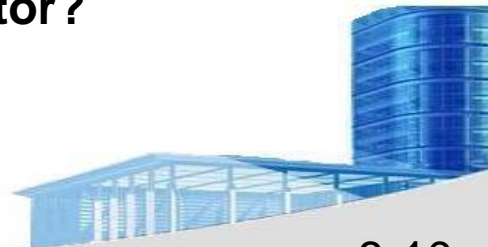
- a statement of need and **feasibility**.
- a bounded statement of **scope** for the system or product
- a **list** of customers, users, and other **stakeholders** who participated in requirements elicitation
- a **description** of the system's **technical environment**.
- a list of requirements (preferably organized by function) and the domain **constraints** that apply to each
- a set of **usage scenarios** that provide insight into the use of the system or product under different operating conditions.
- any **prototypes** developed to better define requirements





Use-Cases

- A collection of user scenarios that describe the thread of usage of a system
- Each scenario is described from the point-of-view of an “**actor** (行动者)” — a person or device that interacts with the software in some way
- Each scenario answers the following **questions**:
 - **Who** is the primary actor, the secondary actor (s)?
 - **What** are the actor’s goals?
 - What **preconditions** should exist before the story begins?
 - What **main tasks or functions** are performed by the actor?





Use-Cases

- Each scenario answers the following questions **(cont.)**
 - What extensions might be considered as the story is described?
 - **What variations** in the actor's interaction are possible?
 - What system **information** will the actor acquire, produce, or change?
 - Will the **actor** have to **inform the system** about changes in the external environment?
 - What information does the actor **desire from** the system?
 - Does the actor **wish to be informed** about unexpected changes?

Ex. SafeHome

《苍穹元宇宙》





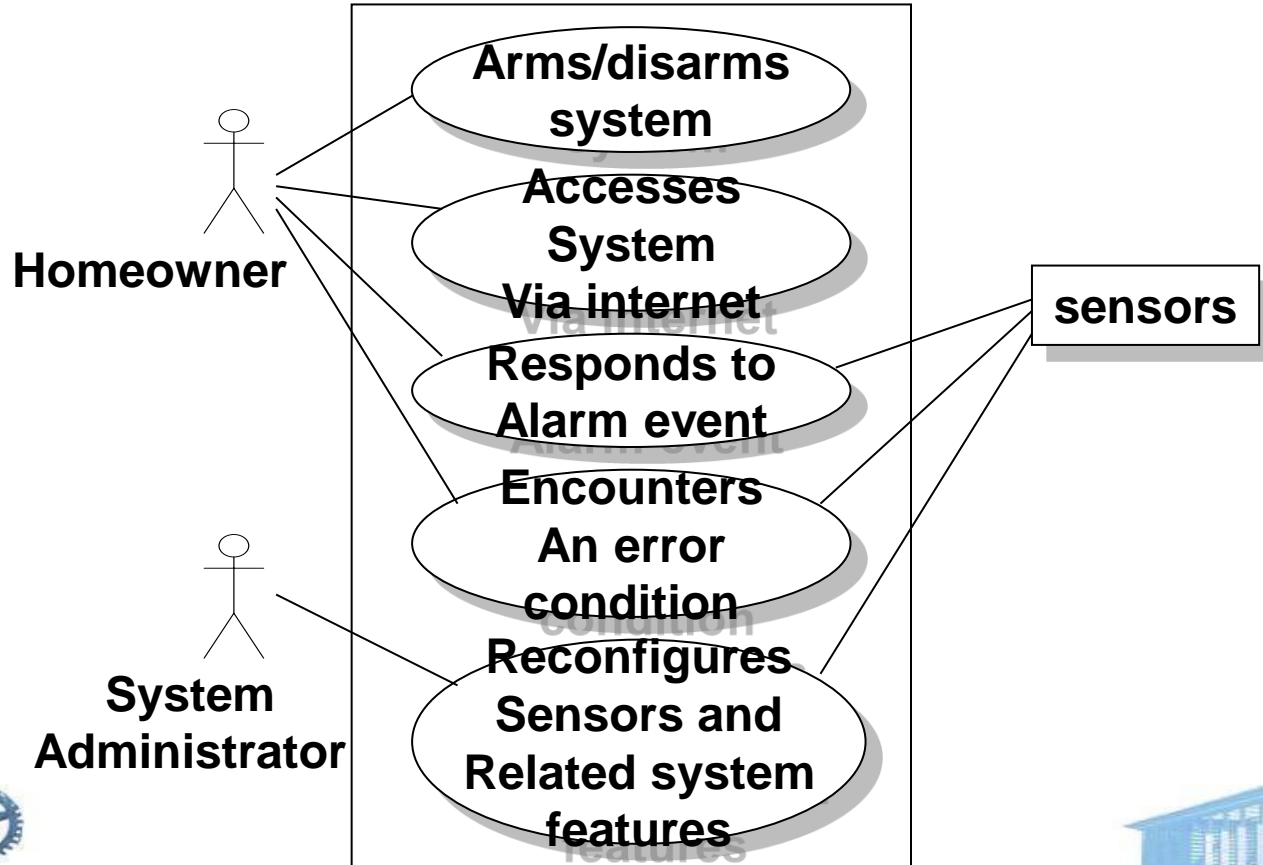
A Example -- SafeHome

*Our research indicates that the market for home security systems is growing at a rate of 40% per year. We would like to enter this market by building a **microprocessor-based** home security system that would protect against and/or recognize a variety of **undesirable situations** such as illegal entry, fire, flooding, and others. The product will use appropriate **sensors** to detect each situation, can be programmed by the homeowner, and will automatically telephone a **monitoring agency** when a situation is detected.*





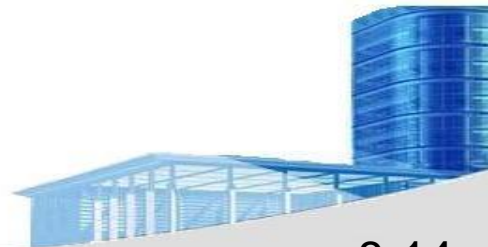
Use-Case Diagram





Building the Analysis Model

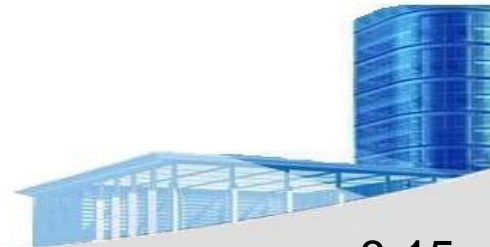
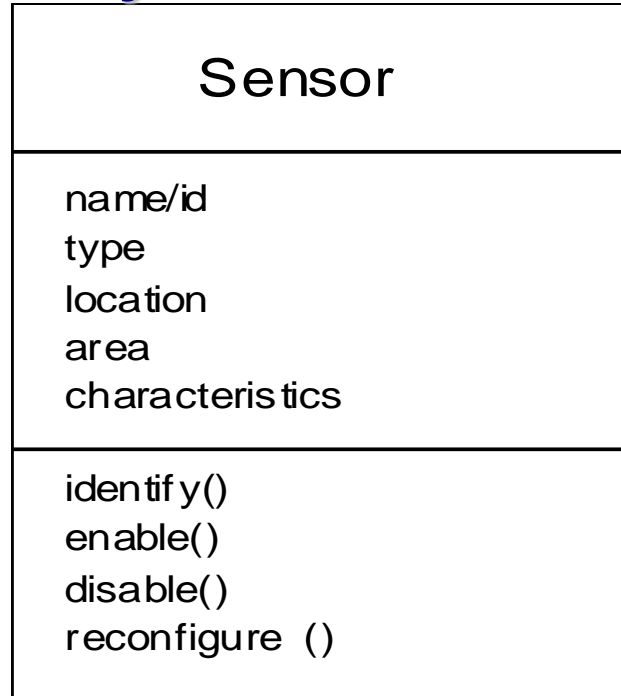
- Elements of the analysis model
 - **Scenario-based elements**
 - Use-case and user-case diagram
 - Sequence of activities within certain context
 - **Class-based elements**
 - Class diagram
 - **Behavioral elements**
 - State diagram
 - **Flow-oriented elements**
 - Data flow diagram





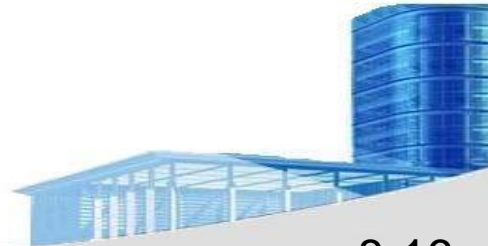
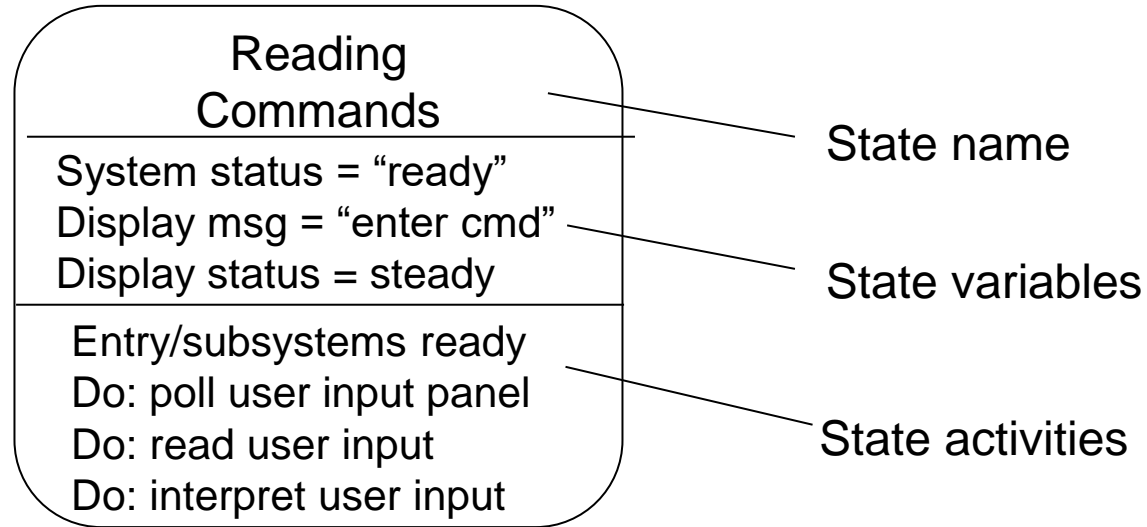
Class Diagram

From the *SafeHome* system ...





State Diagram





State Diagram

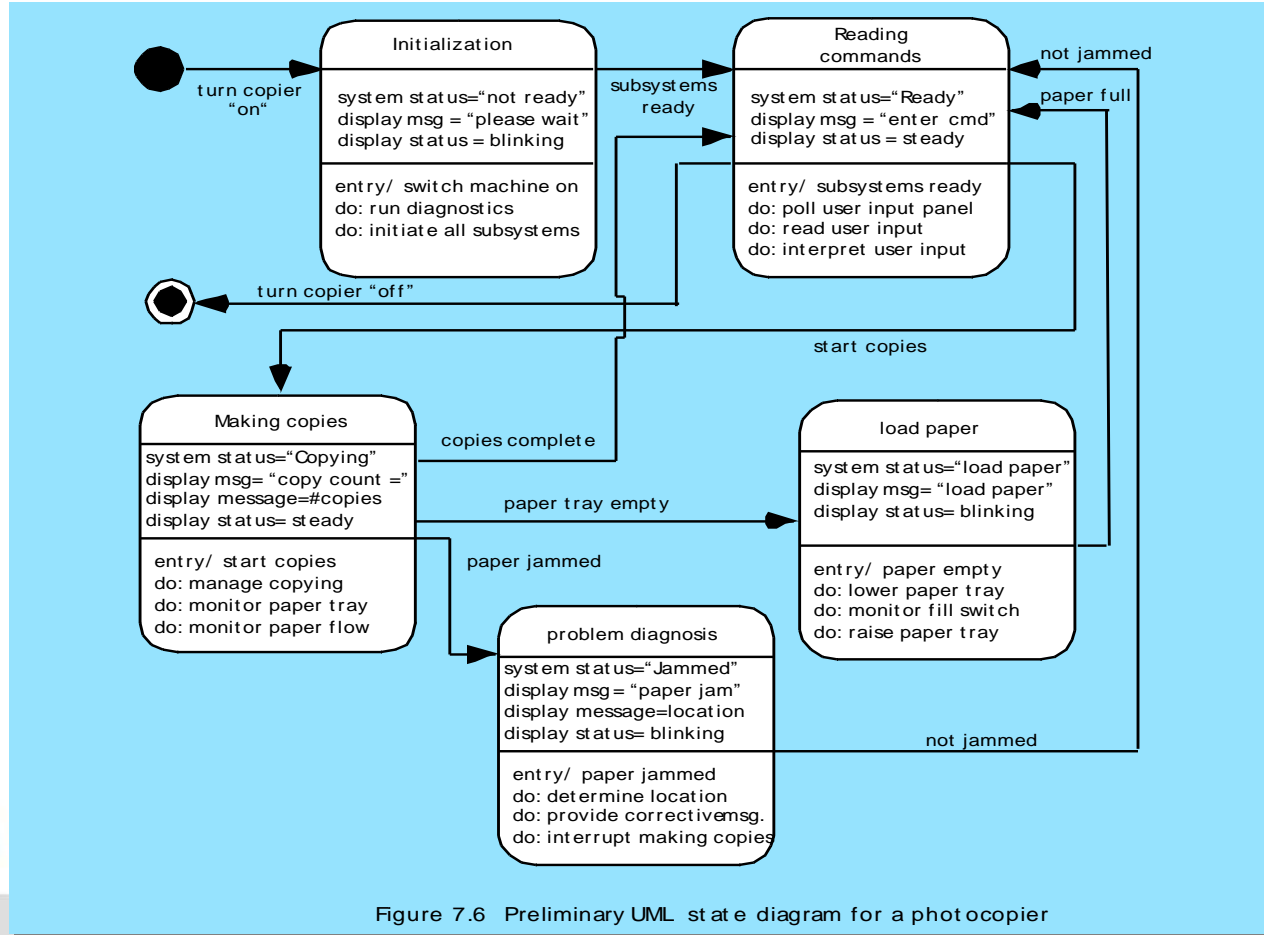


Figure 7.6 Preliminary UML state diagram for a photocopier



****Analysis Patterns (See Ch. 16)**

Pattern name: A descriptor that captures the essence of the pattern.

Intent: Describes what the pattern accomplishes or represents

Motivation: A scenario that illustrates how the pattern can be used to address the problem.

Forces and context: A description of **external issues** (forces) that can affect how the pattern is used and also the external issues that will be resolved when the pattern is applied.

Solution: A description of how the pattern is applied to solve the problem with an emphasis on structural and behavioral issues.





Analysis Patterns (cont.)

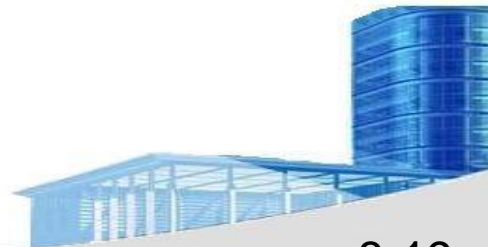
Consequences: Addresses what happens when the pattern is applied and what **trade-offs**(权衡) exist during its application.

Design: Discusses how the analysis pattern can be achieved through the use of known **design patterns**.

Known uses: Examples of uses within actual systems.

Related patterns: One or more analysis patterns that are related to the named pattern because

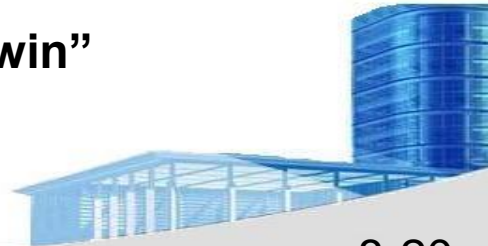
- (1) it is commonly used with the named pattern;
- (2) it is structurally similar to the named pattern;
- (3) it is a variation of the named pattern.





Negotiating Requirements


- **Identify the key stakeholders**
 - These are the people who will be involved in the negotiation
- **Determine each of the stakeholders “win conditions”**
 - Win conditions are not always obvious
- **Negotiate**
 - Work toward a set of requirements that lead to “win-win”





Negotiating Requirements

- **Identify the key stakeholders**
 - These are the people who will be involved in the negotiation
- **Determine each of the stakeholders “win conditions”**
 - Win conditions are not always obvious
- **Negotiate**
 - Work toward a set of requirements that lead to “win-win”

Decorative elements at the bottom of the slide include a set of interlocking blue gears on the left and a blue architectural rendering of a modern building on the right.

If different customers/users cannot agree on requirements, the **risk** of failure is very high.



Requirements Monitoring

Especially needs in **incremental** development

- **Distributed debugging** – uncovers errors and determines their cause
- **Run-time (运行时) verification (证实)** – determines whether software **matches** its **specification** (Ex. 门诊预约数)
- **Run-time validation(确认)** – assesses whether evolving software **meets** user **goals**
- **Business activity monitoring** – evaluates whether a system **satisfies** **business goals**
- **Evolution and co-design** – provides information to stakeholders as the system evolves (Ex. 二维码或刷脸支付!)





Validating Requirements

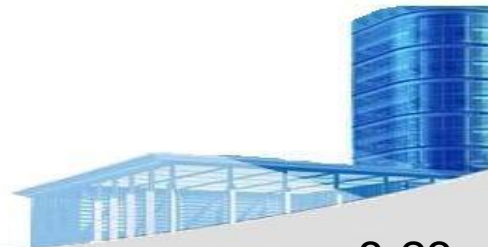
- Is each requirement **consistent** with the overall objective for the system/product?
- Have all requirements been specified at the proper **level of abstraction**? That is, do some requirements provide a level of **technical detail** that is inappropriate at this stage?
(**刷脸**支付?)
- Is the requirement really necessary or does it represent an **add-on** (**附加的**) feature that may not be essential to the objective of the system?





Validating Requirements (cont.)

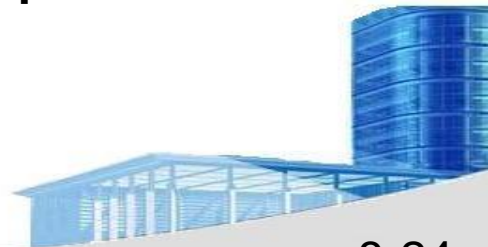
- Is each requirement bounded and **unambiguous**? (e.g. **Web/Mobile side?**)
- Does each requirement have **attribution**? **That is**, is a **source** (generally, **a specific individual**) noted for each requirement? (e.g. **开门** vs **安全属性**)
- Do any requirements **conflict** with other requirements?
- Is each requirement **achievable** in the technical environment that will house the system or product?
- Is each requirement **testable**, once implemented?





Validating Requirements (cont.)

- Does the requirements model properly **reflect** the **information, function and behavior** of the system to be built.
- Has the requirements model been “**partitioned**” in a way that exposes progressively more detailed information about the system.
- Have **requirements patterns** been used to simplify the requirements model. Have all patterns been properly validated? Are all patterns **consistent with** customer requirements?





《Software Requirements Specification》

Due: **22:00 on April 1, 2024**

Minimum requirement of contents:

Introduction (2 points);

User Scenarios(8 points); Data Flow Diagram (7 points); State Diagrams(5 points); Class Diagrams(5 points) and CRC Cards (5 points); Validation Criteria (15 points).

Concerned points:

The accuracy of the validation criteria: full marks can be obtained if more than 90% of the functions are covered. The acceptance testing of the subsystem version 1.0 will strictly go by the criteria.

The language and style of the document must be uniformed (3 points).

Grading: The full mark = **total acquired points × 2**



Tasks

- **Review** Ch. 6, 7, 8
- **Finish** “Problems and points to ponder” in **Ch. 6, 7, 8**
- **Preview** Ch.9,10,11,12
- **Submit** Software Requirements Specification **due April 1!**

