



Chapter 1: Views and Quality Objectives of Software Construction

1.2 Quality Objectives of Software Construction

软件构造的质量目标

Wang Zhongjie
rainy@hit.edu.cn

February 21, 2020

Objective of this lecture

- To know quality factors to be cared in software construction
软件构造过程中应考虑的重要质量指标
- To understand the consequences if quality objectives cannot be achieved
如果达不到期望的质量目标，会有什么后果
- To know what construction techniques are to be studied for each quality factor in this course
有哪些面向质量指标的软件构造技术

上一节搞清楚了“要构造的结果是什么”
这一节要理解清楚“构造的结果如何才算好”

Outline

- **Quality properties of software systems**
 - External vs. internal quality factors
 - Important external quality factors
 - Tradeoff between quality factors
- **Five key quality objectives of software construction**
 - **Easy to understand:** elegant and beautiful code / understandability
 - **Ready for change:** maintainability and adaptability
 - **Cheap for develop:** design for/with reuse: reusability
 - **Safe from bugs:** robustness
 - **Efficient to run:** performance
- **Summary**

Reading

- Object-Oriented Software Construction: 第1章
- 代码大全: 第20章
- 软件工程--实践者的研究方法: 第14章





1 Quality properties of software systems



External and internal quality factors

- **External quality factors:** qualities such as speed or ease of use, whose presence or absence in a software product may be detected by its users.

External quality factors affect users
外部质量因素 影响 用户

- Other qualities applicable to a software product, such as being modular, or readable, are **internal factors**, perceptible only to developers who have access to the actual software text.

Internal quality factors affect the software itself and its developers
内部质量因素 影响 软件本身和它的开发者

- In the end, only external factors matter.
- But the key to achieving these external factors is in the internal ones: for the users to enjoy the visible qualities, the designers and implementers must have applied internal techniques that will ensure the hidden qualities.

External quality results from internal quality
外部质量取决于内部质量

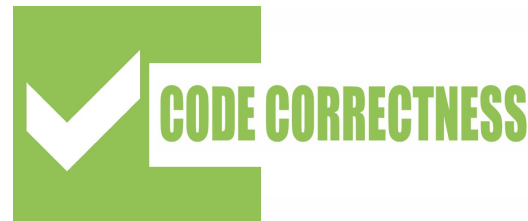


(1) External quality factors



External 1: Correctness

- **Correctness** is the ability of software products to perform their exact tasks, as defined by their specification. 按照预先定义的“规约”执行
- **Correctness is the prime quality** 正确性：最重要的质量指标



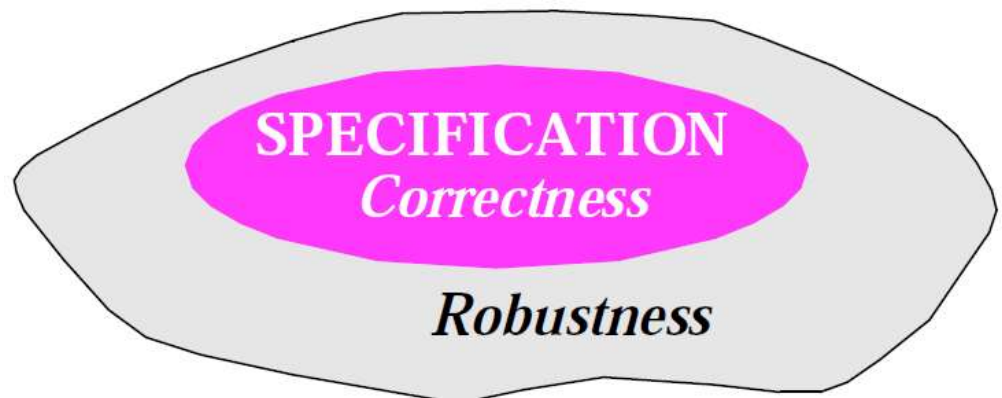
Assume a software system is developed in layers.
Each layer guarantees its correctness under the
assumption that its lower layer is also correct.
每一层保证自己的正确性，同时假设其下层是正确的

External 1: Correctness

- **Approaches of ensuring correctness: Testing and debugging**
测试和调试：发现不正确、消除不正确 ⇒ **Robustness (Chapter 6)**
- **Defensive programming such as typing and assertions**
防御式编程：在写程序的时候就确保正确性 meant to help build software that is correct from the start — rather than debugging it into correctness. ⇒ **Robustness (Chapter 6)**
- **Formal approach: “check”, “guarantee” and “ensure”**
形式化方法：通过形式化验证发现问题
 - Mathematical techniques for formal program specification and verification⇒ **Formal Language, Graduate courses**

External 2: Robustness

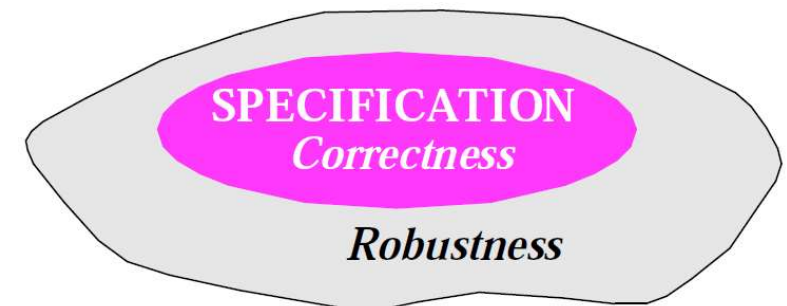
- **Robustness is the ability of software systems to react appropriately to abnormal conditions** 健壮性：针对异常情况的处理
 - Robustness complements correctness. 健壮性是对正确性的补充
 - Correctness addresses the behavior of a system in cases covered by its specification; 正确性：软件的行为要严格的符合规约中定义的行为
 - Robustness characterizes what happens outside of that specification. 健壮性：出现规约定义之外的情形的时候，软件要做出恰当的反应
- **Robustness** is to make sure that if such cases do arise, the system does not cause catastrophic events; it should produce appropriate error messages, terminate its execution cleanly, or enter a so-called “**graceful degradation**” mode.
- 健壮性：出现异常时不要“崩溃”



External 2: Robustness

- Robustness is concerned with “**abnormal case**”, which implies that the notions of normal and abnormal case are always relative to a certain specification “**normal**” 和 “**abnormal**” 是主观而非客观
 - An abnormal case is simply a case that is not covered by the specification 未被specification覆盖的情况即为 “异常情况”
 - If you widen the specification, cases that used to be abnormal become normal — even if they correspond to events such as erroneous user input that you would prefer not to happen 所谓的 “异常”，取决于spec的范畴
 - “Normal” in this sense does not mean “desirable”, but simply “planned for in the design of the software”.
 - Although it may seem paradoxical at first that erroneous input should be called a normal case, any other approach would have to rely on subjective criteria, and so would be useless.

⇒ **Exception handling (Chapter 6)**



External 3: Extendibility

- **Extendibility (可扩展性)** is the ease of adapting software products to changes of specification. 对软件的规约进行修改，是否足够容易？
- **The problem of extendibility is one of scale 规模越大，扩展起来越不容易**
 - For small programs change is usually not a difficult issue; but as software grows bigger, it becomes harder and harder to adapt.
 - A large software system often looks to its maintainers as a giant house of cards in which pulling out any one element might cause the whole edifice to collapse.
- **We need extendibility because at the basis of all software lies some human phenomenon and hence fickleness 为什么要扩展：应对变化**
 - Traditional approaches did not take enough account of change, relying instead on an ideal view of the software lifecycle where an initial analysis stage freezes the requirements, the rest of the process being devoted to designing and building a solution.

External 3: Extendibility

- Two principles are essential for improving extendibility:
 - *Design simplicity*: a simple architecture will always be easier to adapt to changes than a complex one. 简约主义设计
 - *Decentralization*: the more autonomous the modules, the higher the likelihood that a simple change will affect just one module, or a small number of modules, rather than triggering off a chain reaction of changes over the whole system. 分离主义设计

⇒ Chapter 3 (ADT and OOP)

⇒ Chapter 4/5 (Modularity and adaptability)

***Great Design
is great complexity
presented via simplicity***

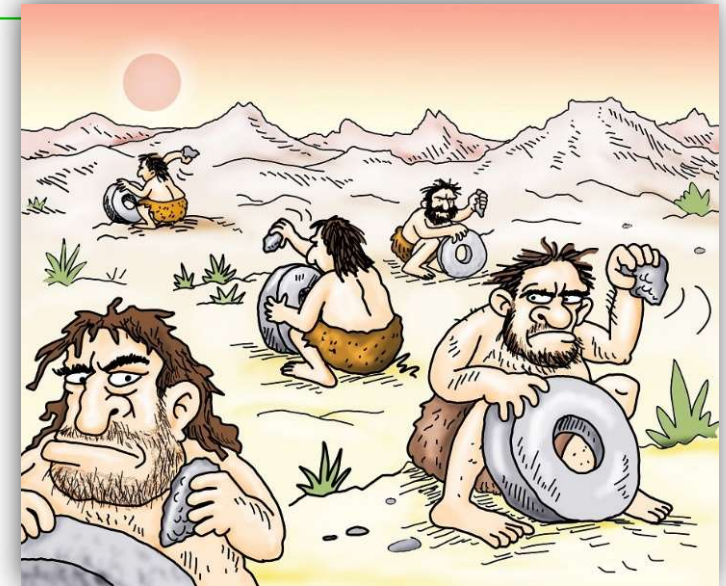
- M. Cobanli



External 4: Reusability

- **Reusability (可复用性)** is the ability of software elements to serve for the construction of many different applications. 一次开发，多次使用
- The need for reusability comes from the observation that software **systems often follow similar patterns**; it should be possible to exploit this **commonality** and avoid reinventing solutions to problems that have been encountered before. 发现共性
 - A reusable software element will be applicable to many different developments.
 - *Don't Repeat Yourself (DRY)*
 - *Don't Re-invent the Wheel*

⇒ Chapter 4 (Design for/with reuse)



External 5: Compatibility

- **Compatibility (兼容性)** is the ease of combining software elements with others. 不同的软件系统之间相互可容易的集成
- **Compatibility is important because we do not develop software elements in a vacuum (真空): they need to interact with each other.**
- **But they too often have trouble interacting because they make conflicting assumptions about the rest of the world.**
 - An example is the wide variety of incompatible file formats supported by many operating systems. A program can directly use another's result as input only if the file formats are compatible. (文件格式的兼容性)

External 5: Compatibility



- The key to compatibility lies in **homogeneity of design** (保持设计的同构性), and in agreeing on standardized conventions for inter-program communication.

The key to compatibility is standardization, especially standard protocols. 标准化

- **Approaches include:**
 - **Standardized file formats**, as in the Unix system, where every text file is simply a sequence of characters.
 - **Standardized data structures**, as in Lisp systems, where all data, and programs as well, are represented by binary trees (called lists in Lisp).
 - **Standardized user interfaces**, as on various versions of Windows, OS/2 and MacOS, where all tools rely on a single paradigm for communication with the user, based on standard components such as windows, icons, etc.
- More general solutions are obtained by defining **standardized access protocols** to all important entities manipulated by the software.

External 6: Efficiency

- **Efficiency** is the ability of a software system to place as few demands as possible on hardware resources, such as processor **time**, space occupied in internal and external **memories**, **bandwidth** used in communication devices.
- **Efficiency does not matter much if the software is not correct** (*“do not worry how fast it is unless it is also right”*). 性能毫无意义，除非有足够的正确性
 - The concern for efficiency must be balanced with other goals such as extendibility and reusability; 对性能的关注 要与 其他质量属性进行折中
 - Extreme optimizations make the software so specialized as to be unfit for change and reuse. 过度的优化导致软件不再适应变化和复用
- Algorithms, I/O, memory management, etc.

Abstract concepts for correctness of computation vs. Concrete implementation for performance through optimization

External 6: Efficiency

过早优化是万恶之源



We should forget
about small
efficiencies, say about
97% of the time:
premature
optimization is the
root of all evil.

Donald Knuth

External 7: Portability (可移植性)

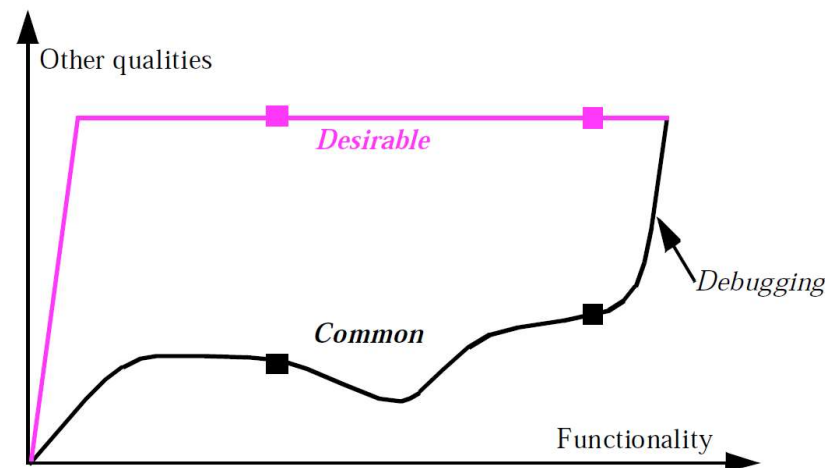
- **Portability** is the ease of transferring software products to various hardware and software environments. 软件可方便的在不同的技术环境之间移植
- Portability addresses variations not just of the physical hardware but more generally of the **hardware-software machine**, the one that we really program, which includes the operating system, the window system if applicable, and other fundamental tools. 硬件、操作系统

External 8: Ease of use

- **Ease of use (易用性)** is the ease with which people of various backgrounds and qualifications can learn to use software products and apply them to solve problems. (容易学、安装、操作、监控)
 - It also covers the ease of installation, operation and monitoring.
- How to provide detailed guidance and explanations to novice users, without bothering expert users who just want to get right down to business. (给用户详细的指南)
- **Structural simplicity**
 - A well-designed system, built according to a clear, well thought-out structure, will tend to be easier to learn and use than a messy one.
- **Know the user**
 - The argument is that a good designer must make an effort to understand the system's intended user community.

External 9: Functionality

- **Functionality is the extent of possibilities provided by a system.**
- *Featurism* (often “*creeping featurism*”) 程序设计中一种不适宜的趋势，即软件开发者增加越来越多的功能，企图跟上竞争，其结果是程序极为复杂、不灵活、占用过多的磁盘空间
 - The easier problem is the **loss of consistency** that may result from the addition of new features, affecting its ease of use. Users are indeed known to complain that all the “bells and whistles” of a product’s new version make it horrendously complex.
 - The more difficult problem is to avoid being so focused on features as to forget the other qualities (**Ignorance of overall quality**).



Osmond's curves

External 9: Functionality

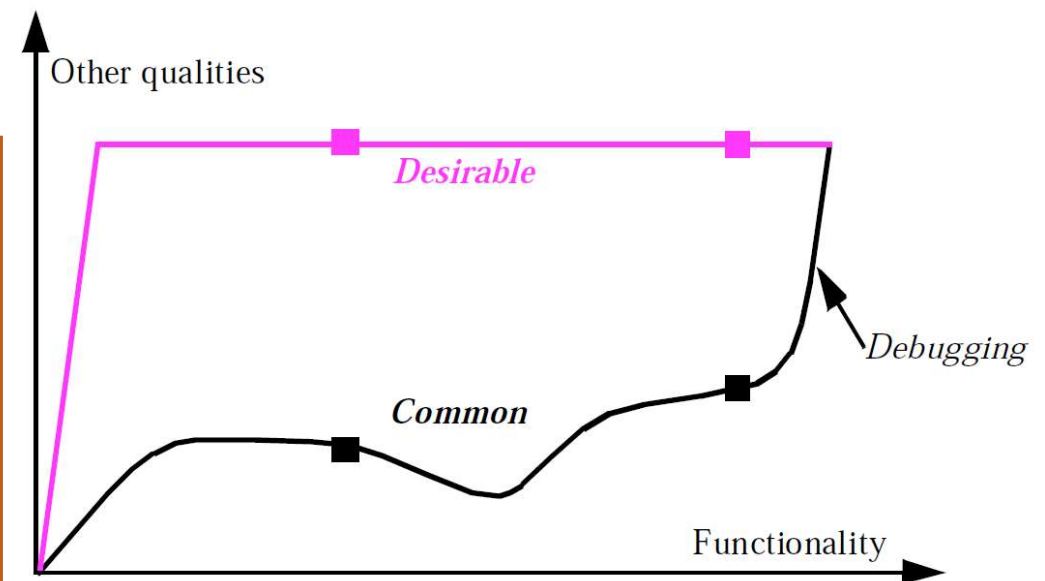
- What Osmond suggests (the color curve) is, aided by the quality-enhancing techniques of OO development, to maintain the quality level constant throughout the project for all aspects but functionality.
- You just do not compromise on reliability, extendibility and the like: you refuse to proceed with new features until you are happy with the features you have.

⇒ Chapter 2 (Agile, SCM)

Start with a small set of key features with all quality factors considered.

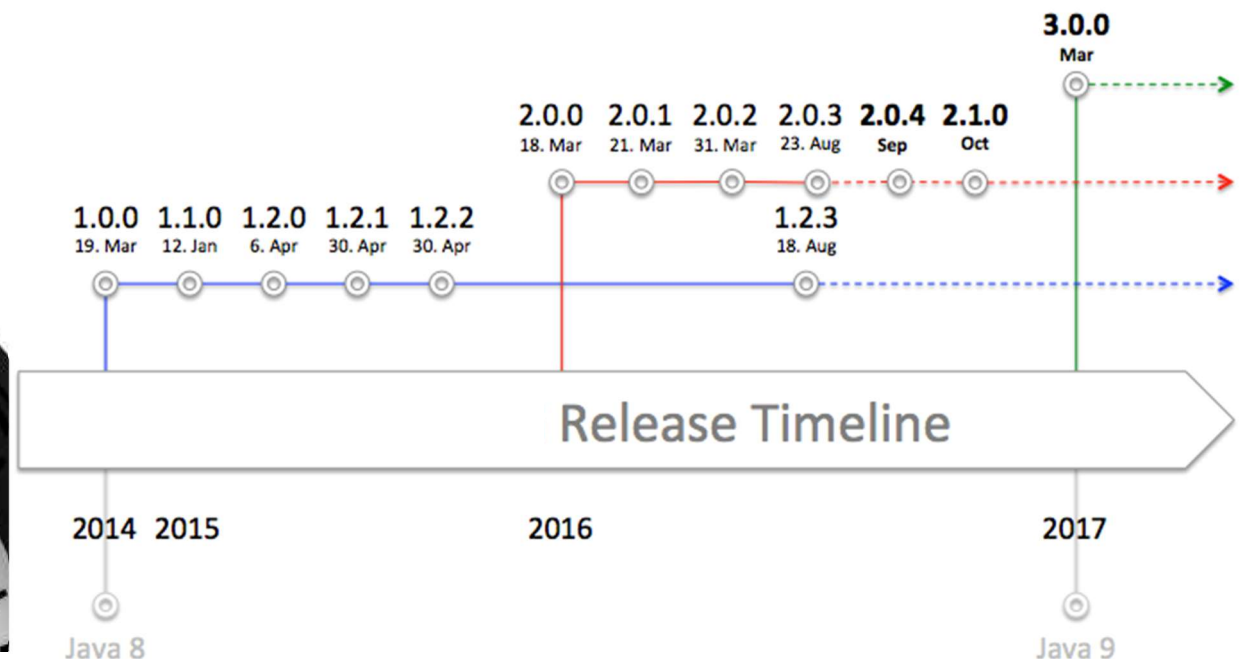
Add more features gradually during development process and guarantee the same quality as key features.

每增加一小点功能，都确保其他质量属性不受到损失



External 10: Timeliness

- **Timeliness (及时性)** is the ability of a software system to be released when or before its users want it.
- A great software product that appears too late might miss its target altogether.



External 10++: Other qualities

- **Verifiability** (可验证性) is the ease of preparing acceptance procedures, especially test data, and procedures for detecting failures and tracing them to errors during the validation and operation phases.
- **Integrity** (完整性) is the ability of software systems to protect their various components (programs, data) against unauthorized access and modification.
- **Repairability** (可修复性) is the ability to facilitate the repair of defects.
- **Economy** (经济性), the companion of timeliness, is the ability of a system to be completed on or below its assigned budget.

If you can't
MEASURE it
you can't **MANAGE** it.





(2) Internal quality factors

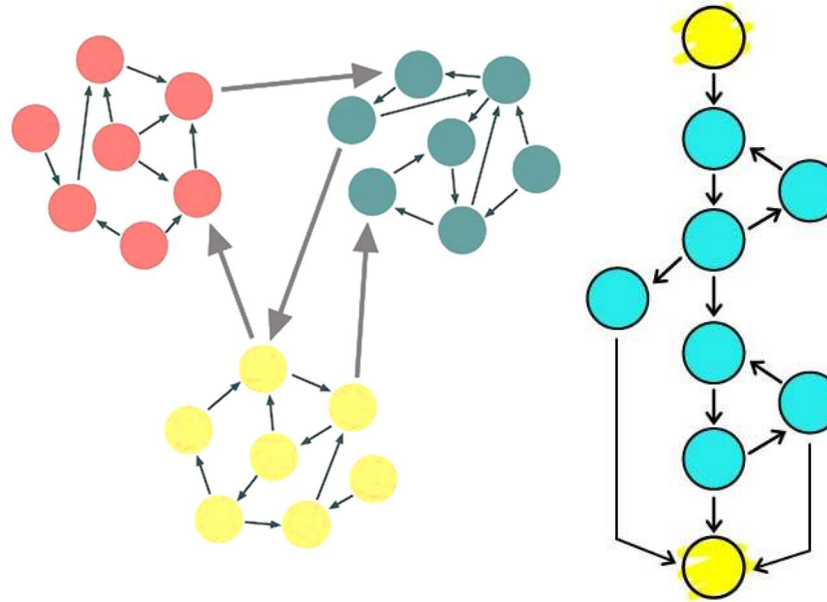


Internal quality factors

Complexity is the enemy of almost any external quality factors!

- Source code related factors such as Lines of Code (LOC), Cyclomatic Complexity, etc
- Architecture-related factors such as coupling, cohesion, etc
- Readability
- Understandability
- Clearness
- Size

system	Lines of code (LOC)
Netscape	17,000,000
Space shuttle	10,000,000
Linux	1,500,000
Windows XP	40,000,000
Boeing 777	7,000,000



Internal quality factors are usually used as partial measurement of external quality factors.



(3) Tradeoff between quality properties



Tradeoff between quality properties 折中

- How can one get *integrity* without introducing protections of various kinds, which will inevitably hamper *ease of use*?
- *Economy* often seems to fight with *functionality*.
- Optimal *efficiency* would require perfect adaptation to a particular hardware and software environment, which is the opposite of *portability*, and perfect adaptation to a specification, where *reusability* pushes towards solving problems more general than the one initially given.
- *Timeliness* pressures might tempt us to use “Rapid Application Development” techniques whose results may not enjoy much *extendibility*.

Integrity vs. ease of use
Economy vs. functionality
Efficiency vs. portability
Efficiency vs. reusability
Economy vs. reusability
Timeliness vs. extendibility

Tradeoff between quality properties

- **Developers need to make tradeoffs.**
 - Too often, developers make these tradeoffs implicitly, without taking the time to examine the issues involved and the various choices available; efficiency tends to be the dominating factor in such silent decisions.
 - A true software engineering approach implies an effort to state the criteria clearly and make the choices consciously.
 - 正确的软件开发过程中，开发者应该将不同质量因素之间如何做出折中的设计决策和标准明确的写下来
- **Necessary as tradeoffs between quality factors may be, one factor stands out from the rest: correctness.**
 - There is never any justification for compromising correctness for the sake of other concerns such as efficiency.
 - If the software does not perform its function, the rest is useless.
 - 虽然需要折中，但“正确性”绝不能与其他质量因素折中。

Key concerns of software construction


- All the qualities discussed above are important.

- 最重要的几个质量因素

But in the current state of the software industry, four stand out:

- *Correctness* and *robustness*: reliability
 - Systematic approaches to software construction
 - Formal specification
 - Automatic checking during development process
 - Better language mechanism
 - Consistency checking tools
- *Extendibility* and *reusability*: modularity

How OOP improves quality

- 
- **Correctness:** encapsulation, decentralization
 - **Robustness:** encapsulation, error handling
 - **Extendibility:** encapsulation, information hiding
 - **Reusability:** modularity, component, models, patterns
 - **Compatibility:** standardized module and interface
 - **Portability:** information hiding, abstraction
 - **Ease of use:** GUI components, framework
 - **Efficiency:** reusable components
 - **Timeliness:** modeling, reuse
 - **Economy:** reuse
 - **Functionality:** extendibility



2 Five key quality objectives of software construction



Quality considerations of this course

- Elegant and beautiful code \Rightarrow **easy to understand**, understandability
- Design for/with reuse \Rightarrow **cheap for develop**
- Low complexity \Rightarrow **ready for changes**, easy to extend
- Robustness and correctness \Rightarrow **safe from bug**, not error-prone
- Performance and efficiency \Rightarrow **efficient to run**



Removed



Chapter 4 Reusability



Chapter 5
Extendibility/Maintainability



Chapter 6
Robustness/Correctness



Removed



Chapter 3 ADT and OOP

Understandability

	Moment		Period	
	Code-level	Component-level	Code-level	Component-level
Build-time	代码的可理解性 （变量/子程序/ 语句的命名与构造 标准、代码布局 与风格、注释、 复杂度） Code Review; Walkthrough; Static Code Analysis; ADT/函数规约	构件/项目的可理解 性（包的组织、文 件的组织、命名空 间）	Refactoring	Version control
Run-time			Log Trace	

Reusability

	Moment		Period	
	Code-level	Component-level	Code-level	Component-level
Build-time	ADT/OOP; 接口与实现分离; 继承/重载/重写; 组合/代理; 多态; 子类型与泛型; OO设计模式	API design; Library; Framework (for/with reuse); Static linking		
Run-time		Dynamic linking		

Maintainability and Adaptability

	Moment		Period	
	Code-level	Component-level	Code-level	Component-level
Build-time	模块化设计; 聚合度/耦合度; SOLID; OO设计模式; Table-based programming; State-based programming; Grammar-based programming	SOLID; GRASP		SCM Version Control
Run-time				

Robustness

	Moment		Period	
	Code-level	Component-level	Code-level	Component-level
Build-time	Error handling; Exception handling; Assertion; Defensive programming; Test-first programming		Continuous Integration; Regression Testing	
Run-time	Unit/Integration Testing; Debug; Memory Dumping		Logging Tracing	

Performance

	Moment		Period	
	Code-level	Component-level	Code-level	Component-level
Build-time	代码调优; Design pattern			
Run-time	空间复杂性（内存管理性能）; 时间复杂性（I/O性能）; Memory dump; Garbage Collection (GC)	分布式系统	Performance profiling, analysis and tuning	并行/多线程



Summary



Summary

- **Quality properties of software systems**
 - External vs. internal quality factors
 - Important external quality factors
 - Tradeoff between quality factors
- **Five key quality objectives of software construction**
 - **Easy to understand:** elegant and beautiful code / understandability
 - **Ready for change:** maintainability and adaptability
 - **Cheap for develop:** design for/with reuse: reusability
 - **Safe from bugs:** robustness
 - **Efficient to run:** performance
- **Construction techniques to be studied in this course (classified by the orientation of five key quality objectives)**



The end

February 21, 2020