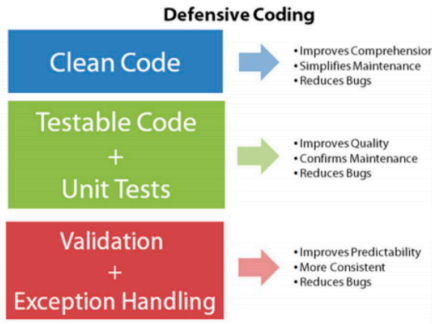
Defensive Programming

Defensive Programming 防御式编程

- A form of defensive design intended to ensure the continuing function of a piece of software under unforeseen circumstances. （保证 对程序的不可预见的使用，不会造成程序功能上的损坏
- Often used when high availability, safety or security is needed.
- Making sure that no warning produced by Static Analysis is a form of defensive programming



Defensive Programming:

- Redundant code(多余代码) is incorporated to check system state after modifications.
- (加入多余代码来进行检测？)
- Implicit assumptions (隐式假设) are tested explicitly.
- Risky programming constructs are avoided.

Risky programming constructs

- Pointers / Dynamic memory allocation / Floating-point numbers / Parallelism / Recursion / Interrupts

Defensive Programming Examples

- Use boolean variable not integer
- Test i <= n not i = = n (greater range)
- Assertion checking (e.g., validate parameters)
- Build debugging code into program with a switch to display values at interfaces
- Error checking codes in data (e.g., checksum or hash)

Maintenance

Most production programs are maintained by people other than the programmers who originally wrote them.

Fault Tolerance（容错能力）（错误检测？）

- General Approach: Failure detection→Damage assessment→Fault recovery→Fault repair
- N-version programming -- Execute independent implementation in parallel, compare results, accept the most probable.
- Basic Techniques: Report all errors for quality control
  - After error continue with next transaction (e.g., drop packet)
  - Timers and timeout in networked systems
  - User break options (e.g., force quit, cancel)
  - Error correcting codes in data
  - Bad block tables on disk drives
  - Forward and backward pointers in databases
- Backward Recovery:
  - Record system state at specific events (checkpoints). After failure, recreate state at last checkpoint.
  - Backup of files
  - Combine checkpoints with system log (audit trail of transactions) that allows transactions from last checkpoint to be repeated automatically.
  - Test the restore software!

Software Engineering for Real Time

- The special characteristics of real time computing require extra attention to good software engineering principles:
  - Requirements analysis and specification
  - Special techniques (e.g., locks on data, semaphores, etc.)
  - Development of tools
  - Modular design
  - Exhaustive testing
- Testing and debugging need special tools and environments
  - Debuggers, etc., cannot be used to test real time performance
  - Simulation of environment may be needed to test interfaces -- e.g., adjustable clock speed
  - General purpose tools may not be available

Agents and Components

- A large system will have many agents and components:
  - agent (people or external systems)
  - each is potentially unreliable and insecure
  - components acquired from third parties may have unknown security problems (COTS problem)
- The software development challenge:
  - develop secure and reliable components
  - protect whole system from security problems in parts of it

Techniques: Barriers（设置障碍）

- Place barriers that separate parts of a complex system:
  - Isolate components, e.g., do not connect a computer to a network
  - Firewalls
  - Require authentication to access certain systems or parts of systems
- Every barrier imposes restrictions on permitted uses of the system
  - Barriers are most effective when the system can be divided into subsystems with simple boundaries

Techniques: Authentication & Authorization

- Authentication establishes the identity of an agent: （身份验证：建立agent身份）
  - What the agent knows (e.g., password)
  - What the agent possess (e.g., smart card)
  - Where does the agent have access to (e.g., controller)
  - What are the physical properties of the agent (e.g., fingerprint)
- Authorization establishes what an authenticated agent may do: （授权）
  - Access control lists
  - Group membership

Techniques: Encryption

- Allows data to be stored and transmitted securely, even when the bits are viewed by unauthorized agents
  - Private key and public key
  - Digital signatures

Documentation

Javadoc Comments

```
/*
* Overall Description
…
* @param Parameter name, Description
* @return Description
* @throws Exception name, Condition under which the exception is thrown
```

```
7 */
```

- javadoc comments begin with /** and end with */
- In a javadoc comment, a * at the beginning of the line is not part of the comment text
- javadoc comments must be immediately before:
  - a class (plain, inner, abstract, or enum)
  - an interface
  - a constructor
  - a method
  - a field (instance or static)
- Anywhere else, javadoc comments will be ignored!

javadoc comments (but not other kinds of comments) are written in HTML
Javadoc comments should be written for programmers who want to use your code
Javadoc comments should not be written for:

- Programmers who need to debug, maintain, or upgrade the code （Internal (// or /*..*/) comments should be used for this purpose）
- People who just want to use the program

描述how to use，而不是internal working

Tags in doc comments

- In class and interface descriptions, use:
  - @author @version
- In method descriptions
  - @param @return @exception

Rules for writing summaries

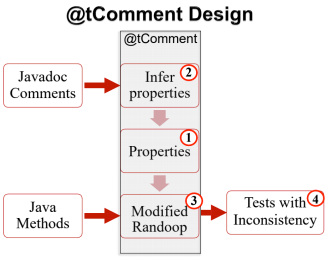
- For methods, omit the subject and write in the third-person narrative form（省略主语，用第三人称单数）
- Use the word this rather than “the” when referring to instances of the current class
- Do not add parentheses to a method or constructor name unless you want to specify a particular signature

Bugs and missing features

- TODO -- describes a feature that should be added
- FIXME -- describes a bug in the method
- XXX -- this needs to be thought about some more

Problems of Javadoc

- Outdated Code Comments
- Javadoc Comments can be Inconsistent with Code
  - An approach for testing comment-code inconsistency: @tComment (检查Javadoc Comments和Java Methods是否一致)
  - 两种inconsistency: correct code, incorrect comment / fault in code, correct comment



Software Reuse and Component- Based Software Engineering
大多数软件工程使用composition设计系统，并且专注于custom development of components
为了更低成本更快的提高软件质量，开始采用系统重用作为设计过程

Types of Software Reuse

- Application System Reuse

    - reusing an entire application by incorporation of one application inside another (COTS reuse) （合并？）
    - development of application families (e.g. MS Office)
- Component Reuse
    - components (e.g. subsystems or single objects) of one application reused in another application
- Function Reuse
    - reusing software components that implement a single well-defined function

Benefits of Reuse
- Increased Reliability
    - components already exercised in working systems
- Reduced Process Risk
    - less uncertainty in development costs
- Effective Use of Specialists
    - reuse components instead of people
- Standards Compliance
    - embed standards in reusable components
- Accelerated Development
    - avoid custom development and speed up delivery

Reuse Problems
- Increased maintenance costs
- Lack of tool support
- Pervasiveness of the “not invented here” syndrome
- Need to create and maintain a component library
- Finding and adapting reusable components

Generator-Based Reuse
- Program generators reuse standard patterns and algorithms
- Programs are automatically generated to conform to user defined parameters
- Possible when it is possible to identify the domain abstractions and their mappings to executable code
- Domain specific language is required to compose and control these abstractions

Types of Program Generators
- Applications generators for business data processing
- Parser and lexical analyzers generators for language processing
- Code generators in CASE tools
- User interface design tools

Component-Based Engineering
Component-Based Software Engineering
- CBSE is an approach to software development that relies on reuse
- CBSE emerged from the failure of object-oriented development to support reuse effectively
- Objects (classes) are too specific and too detailed to support design for reuse work
- Components are more abstract than classes and can be considered to be stand-alone service providers（object太具体 component更抽象一些）

Component Abstractions
- Functional Abstractions: component implements a single function (e.g. ln)
- Casual Groupings: component is part of a loosely related entities like declarations and functions
- Data Abstractions: abstract data types or objects
- Cluster Abstractions: component from group of cooperating objects
- System Abstraction: component is a self-contained system

Engineering of Component-Based Systems

Definition

- Component Qualification
  - candidate components are identified based on services provided and means by which consumers access them
- Component Adaptation
  - candidate components are modified to meet the needs of the architecture or discarded
- Component Composition
  - architecture dictates the composition of the end product from the nature of the connections and coordination mechanisms
- Component Update
  - updating systems that include COTS is made more complicated by the fact that a COTS developer must be involved

Commercial Off-the-Shelf Software (COTS)
商用现货（COTS）产品是包装解决方案，然后进行调整，以满足采购组织的需求，而不是调试定制或定制的解决方案
- COTS systems are usually complete applications library the off an applications programming interface (API)
- Building large systems by integrating COTS components is a viable development strategy for some types of systems (e.g. E-commerce or video games)

COTS Integration Problems
- Lack of developer control over functionality and performance
- Problems with component interoperability as COTS vendors make different user assumptions
- COTS vendors may not offer users any control over the evolutions of its components
- Vendors may not offer support over the lifetime of a product built with COTS components

Developing Components for Reuse
- Components may constructed with the explicit goal to allow them to be generalized and reused
- Component reusability should strive to
  - reflect stable domain abstractions
  - hide state representations
  - be independent (low coupling)
  - propagate exceptions via the component interface

Reusable Components
- Tradeoff between reusability and usability
  - generic components can be highly reusable
  - reusable components are likely to be more complex and harder to use
- Development costs are higher for reusable components than application specific components
- Generic components may be **less space-efficient and have longer execution times** than their application specific analogs

Domain Engineering
领域工程的主要目的是实现对特定领域中可复用成分的分析、生产和管理
- Domain analysis
  - define application domain to be investigated
  - categorize items extracted from domain
  - collect representative applications from the domain
  - analyze each application from sample
  - develop an analysis model for objects
- Domain model
- Software architecture development
- Structural model
  - consists of small number of structural elements manifesting clear patterns of interaction
  - architectural style that can be reused across applications in the domain
  - structure points are distinct constructs within the structural model (e.g. interface, control mechanism, response mechanism)
- Reusable component development
- Repository of reusable components is created

Structure Point Characteristics

- Abstractions with limited number of instances within an application and recurs in applications in the domain
- Rules governing the use of a structure point should be easily understood and structure point interface should be simple
- Structure points should implement information hiding by isolating all complexity contained within the structure point itself

Component-Based Development

- Analysis
- Architectural design
  - component qualification
  - component adaptation
  - component decomposition
- Component engineering
- Testing
- Iterative component update

Component Adaptation Techniques

- White-box Wrapping
  - integration conflicts removed by making code-level modifications to the code
- Grey-box Wrapping
  - used when component library provides a component extension language or API that allows conflicts to be removed or masked
- Black-box Wrapping
  - requires the introduction of pre- and post-processing at the component interface to remove or mask conflicts