Modularity

Guidelines for design
in any programming language

© Gunnar Gotshalks

Modular Software

- Software constructed as assemblies of small pieces

© Gunnar Gotshalks

Modular Software – 2

- Software constructed as assemblies of small pieces
  - **Each piece encompasses the data and operations necessary to do one task well**

© Gunnar Gotshalks

Modular Software – 3

- Software constructed as assemblies of small pieces
  - **Each piece encompasses the data and operations necessary to do one task well**
- Modular software ==> maintainable software
  - **Uses divide and conquer principle**

© Gunnar Gotshalks

Modular Software – 4

- Software constructed as assemblies of small pieces
  - » **Each piece encompasses the data and operations necessary to do one task well**
- Modular software ==> maintainable software
  - » **Uses divide and conquer principle**
- Meyer:
  - » **To achieve extendibility, reusability, compatibility, need modular software and methods to produce modular software**

© Gunnar Gotshalks

Modular Software – 5

- Software constructed as assemblies of small pieces
  - **Each piece encompasses the data and operations necessary to do one task well**
- Modular software ==> maintainable software
  - **Uses divide and conquer principle**
- Meyer:
  - **To achieve extendibility, reusability, compatibility, need modular software and methods to produce modular software**
- In OO design — non-OO design
  - **Module ≡ Class** — **Module ≡ File**

© Gunnar Gotshalks

Issues in Modular Design

- Information hiding

Issues in Modular Design – 2

- Information hiding
  - **Client sees only interface**

© Gunnar Gotshalks

Issues in Modular Design – 3

- Information hiding
  - **Client sees only interface**

- Independence

© Gunnar Gotshalks

Issues in Modular Design – 4

- Information hiding
  - » **Client sees only interface**
- Independence
  - » **Each module implements a separable part of the whole**

© Gunnar Gotshalks

Issues in Modular Design – 4

- Information hiding
  - » **Client sees only interface**
- Independence
  - » **Each module implements a separable part of the whole**
  - » **Modules have small, simple interfaces**

© Gunnar Gotshalks

Issues in Modular Design – 5

- Information hiding
  - » **Client sees only interface**
- Independence
  - » **Each module implements a separable part of the whole**
  - » **Modules have small, simple interfaces**
  - » **High interaction between modules is usually symptomatic of a bad modular design**

© Gunnar Gotshalks

Cohesion & Coupling

- Key ideas for measuring modularity

© Gunnar Gotshalks

Cohesion & Coupling – 2

- Key ideas for measuring modularity
  - **Cohesion**
    - **How "self contained" is a module**

© Gunnar Gotshalks

Cohesion & Coupling – 3

- Key ideas for measuring modularity
  - » **Cohesion**
    - > **How "self contained" is a module**
    - > **No extraneous bits & pieces**

© Gunnar Gotshalks

Cohesion & Coupling – 4

- Key ideas for measuring modularity
  - » **Cohesion**
    - > **How "self contained" is a module**
    - > **No extraneous bits & pieces**
  - » **Coupling**
    - > **How dependent modules are on each other**

© Gunnar Gotshalks

Cohesion & Coupling – 5

- Key ideas for measuring modularity
  - » **Cohesion**
    - > **How "self contained" is a module**
    - > **No extraneous bits & pieces**
  - » **Coupling**
    - > **How dependent modules are on each other**
    - > **Size of the common interface**

Do you want high or low cohesion and coupling?

© Gunnar Gotshalks

Cohesion & Coupling – 5

- Key ideas for measuring modularity
  - » **Cohesion**
    - > **How "self contained" is a module**
    - > **No extraneous bits & pieces**
  - » **Coupling**
    - > **How dependent modules are on each other**
    - > **Size of the common interface**

Want high cohesion and low coupling

© Gunnar Gotshalks

Criteria for Modularity

- Want a modular design method satisfying
  - **Decomposability**

Criteria for Modularity – 2

- Want a modular design method satisfying
  - » **Decomposability**
  - » **Composability**

© Gunnar Gotshalks

Criteria for Modularity – 3

- Want a modular design method satisfying
  - » **Decomposability**
  - » **Composability**
  - » **Understandability**

© Gunnar Gotshalks

Criteria for Modularity – 4

- Want a modular design method satisfying
  - » **Decomposability**
  - » **Composability**
  - » **Understandability**
  - » **Continuity**

© Gunnar Gotshalks

Criteria for Modularity – 5

- Want a modular design method satisfying
  - » **Decomposability**
  - » **Composability**
  - » **Understandability**
  - » **Continuity**
  - » **Protection**

© Gunnar Gotshalks

Criteria for Modularity – 6

- Want a modular design method satisfying
  - **Decomposability**
  - **Composability**
  - **Understandability**
  - **Continuity**
  - **Protection**
- Without these, we cannot produce modular software

© Gunnar Gotshalks

Decomposability

- Decomposition
  - » **Break a problem into sub-problems connected by simple structures**

© Gunnar Gotshalks

Decomposability – 2

- Decomposition
  - » **Break a problem into sub-problems connected by simple structures**
    - > **Minimize communication between sub-problems**

© Gunnar Gotshalks

Decomposability – 3

- Decomposition
  - » **Break a problem into sub-problems connected by simple structures**
    - > **Minimize communication between sub-problems**
    - > **Permit further work to proceed separately on each sub-problem**

© Gunnar Gotshalks

Decomposability – 4

- Decomposition
  - **Break a problem into sub-problems connected by simple structures**
    - **Minimize communication between sub-problems**
    - **Permit further work to proceed separately on each sub-problem**
  - Example
    - **See slides on top down design**

© Gunnar Gotshalks

Composability

- Composition
  - **Produce software from reusable plug and play modules**

© Gunnar Gotshalks

Composability – 2

- Composition
  - » **Produce software from reusable plug and play modules**
  - » **Composed software is itself a reusable module**

© Gunnar Gotshalks

Composability – 3

- Composition
  - » **Produce software from reusable plug and play modules**
  - » **Composed software is itself a reusable module**
  - » **Reusable modules work in environments different from the ones in which they were developed**

© Gunnar Gotshalks

Composability – 4

- Composition
  - » **Produce software from reusable plug and play modules**
  - » **Composed software is itself a reusable module**
  - » **Reusable modules work in environments different from the ones in which they were developed**

  - » Examples
    - > **Using pipe in the Unix shell to combine Unix commands**

© Gunnar Gotshalks

Composability – 5

- Composition
  - » **Produce software from reusable plug and play modules**
  - » **Composed software is itself a reusable module**
  - » **Reusable modules work in environments different from the ones in which they were developed**
  - » Examples
    - > **Using pipe in the Unix shell to combine Unix commands**
    - > **See slides on abstract data types and bottom-up design**

© Gunnar Gotshalks

Decomposability and Composability

- Composability and decomposability are independent and often at odds

© Gunnar Gotshalks

Decomposability and Composability – 2

- Composability and decomposability are independent and often at odds
  - **Top down design favours generating modules that fulfill specific requirements**

© Gunnar Gotshalks

Decomposability and Composability – 3

- Composability and decomposability are independent and often at odds
  - **Top down design favours generating modules that fulfill specific requirements**
    - **Unsuitable for composition**

© Gunnar Gotshalks

Decomposability and Composability – 4

- Composability and decomposability are independent and often at odds
  - **Top down design favours generating modules that fulfill specific requirements**
    - **Unsuitable for composition**
  - **Bottom up design favours general modules that are too general**

© Gunnar Gotshalks

Decomposability and Composability – 5

- Composability and decomposability are independent and often at odds
  - » **Top down design favours generating modules that fulfill specific requirements**
    - > **Unsuitable for composition**
  - » **Bottom up design favours general modules that are too general**
    - > **When combined generate inefficient systems – in size and speed**

© Gunnar Gotshalks

Decomposability and Composability – 6

- Composability and decomposability are independent and often at odds
  - » **Top down design favours generating modules that fulfill specific requirements**
    - > **Unsuitable for composition**
  - » **Bottom up design favours general modules that are too general**
    - > **When combined generate inefficient systems – in size and speed**
- Both top down – decomposition – and bottom up – composition are required

© Gunnar Gotshalks

Decomposability and Composability – 7

- Composability and decomposability are independent and often at odds
  - » **Top down design favours generating modules that fulfill specific requirements**
    - > **Unsuitable for composition**
  - » **Bottom up design favours general modules that are too general**
    - > **When combined generate inefficient systems – in size and speed**
- Both top down – decomposition – and bottom up – composition are required
  - » **Trick is to know when and how to best use both methods**

© Gunnar Gotshalks

Understandability

- Understandable
  - » **Minimize need to understand module context**

Understandability – 2

- Understandable
  - » **Minimize need to understand module context**
    - > **Know or examine as few other modules as possible**

© Gunnar Gotshalks

Understandability – 3

- Understandable
  - » **Minimize need to understand module context**
    - > **Know or examine as few other modules as possible**
    - > **Very important for maintenance**

© Gunnar Gotshalks

Continuity

- The smaller the change in specification, the fewer the number of modules that must be changed (edited) and if possible compiled

© Gunnar Gotshalks

Continuity – 2

- The smaller the change in specification, the fewer the number of modules that must be changed (edited) and if possible compiled
  - » **Example**
    - > **Use of symbolic constants – need to change value in one place but requires recompilation of every module using the constant**

© Gunnar Gotshalks

Understandability and Continuity

- Related to coupling and cohesion

A module should do one thing well

Modular Protection

- Confine abnormal run time errors to one or a very few modules

Modular Protection – 2

- Confine abnormal run time errors to one or a very few modules
- Avoid propagation of error conditions to neighbouring modules

© Gunnar Gotshalks

Modular Protection – 3

- Confine abnormal run time errors to one or a very few modules

- Avoid propagation of error conditions to neighbouring modules
  - » **Example**
    - > **Validate input before propagating it to other modules**

© Gunnar Gotshalks

Modular Protection – 4

- Confine abnormal run time errors to one or a very few modules

- Avoid propagation of error conditions to neighbouring modules
  - **Example**
    - **Validate input before propagating it to other modules**

- Exceptions in languages like C++ and Java can be used in an undisciplined manner leading to violations of protection

© Gunnar Gotshalks

Modular Protection – 5

- Confine abnormal run time errors to one or a very few modules
- Avoid propagation of error conditions to neighbouring modules
  - » **Example**
    - > **Validate input before propagating it to other modules**
- Exceptions in languages like C++ and Java can be used in an undisciplined manner leading to violations of protection
  - » **Exceptions raised in one part of the system should not be handled by a remote part of the system**

Design Rules to Ensure Modularity

- We have seen criteria for modular software development

© Gunnar Gotshalks

Design Rules to Ensure Modularity – 2

- We have seen criteria for modular software development
- From them we can deduce the following rules that can help establish the properties we want in our designs

© Gunnar Gotshalks

Design Rules to Ensure Modularity – 3

- We have seen criteria for modular software development
- From them we can deduce the following rules that can help establish the properties we want in our designs

  » **Direct Mapping rule**

Design Rules to Ensure Modularity – 4

- We have seen criteria for modular software development
- From them we can deduce the following rules that can help establish the properties we want in our designs

  » **Direct Mapping rule**

  » **Few interfaces rule**

© Gunnar Gotshalks

Design Rules to Ensure Modularity – 5

- We have seen criteria for modular software development
- From them we can deduce the following rules that can help establish the properties we want in our designs

  » **Direct Mapping rule**

  » **Few interfaces rule**

  » **Small interfaces rule**

© Gunnar Gotshalks

Design Rules to Ensure Modularity – 6

- We have seen criteria for modular software development
- From them we can deduce the following rules that can help establish the properties we want in our designs

  » **Direct Mapping rule**

  » **Few interfaces rule**

  » **Small interfaces rule**

  » **Explicit interfaces rule**

© Gunnar Gotshalks

Design Rules to Ensure Modularity – 7

- We have seen criteria for modular software development
- From them we can deduce the following rules that can help establish the properties we want in our designs
  - » **Direct Mapping rule**
  - » **Few interfaces rule**
  - » **Small interfaces rule**
  - » **Explicit interfaces rule**
  - » **Information Hiding rule**

© Gunnar Gotshalks

Direct Mapping Rule

Correspondence

The structure used in implementing a software system should remain compatible with the structure used in modeling the system

© Gunnar Gotshalks

Direct Mapping Rule – 2

Correspondence

The structure used in implementing a software system should remain compatible with the structure used in modeling the system

- Software design involves addressing needs in a problem domain

© Gunnar Gotshalks

Direct Mapping Rule – 3

Correspondence
The structure used in implementing a software system should remain compatible with the structure used in modeling the system

- Software design involves addressing needs in a problem domain
- Have to understand the problem **AND** its domain, then formulate a solution

© Gunnar Gotshalks

Direct Mapping Rule – 4

Correspondence
The structure used in implementing a software system should remain compatible with the structure used in modeling the system

- Software design involves addressing needs in a problem domain
- Have to understand the problem **AND** its domain, then formulate a solution
- Model our solution in some notation (we use BON)

© Gunnar Gotshalks

Direct Mapping Rule – 5

Correspondence
The structure used in implementing a software system should remain compatible with the structure used in modeling the system

- Software design involves addressing needs in a problem domain
- Have to understand the problem AND its domain, then formulate a solution
- Model our solution in some notation (we use BON)
- Need a clear mapping from the proposed solution (in BON) to program source text

© Gunnar Gotshalks

Direct Mapping Rule – 7

Correspondence
The structure used in implementing a software system should remain compatible with the structure used in modeling the system

- Software design involves addressing needs in a problem domain
- Have to understand the problem AND its domain, then formulate a solution
- Model our solution in some notation (we use BON)
- Need a clear mapping from the proposed solution (in BON) to program source text
- Arises from **continuity** and **decomposability**

© Gunnar Gotshalks

Few Interfaces Rule

Every module should communicate with as few others as possible

© Gunnar Gotshalks

Few Interfaces Rule – 2

Every module should communicate with as few others as possible

- Restrict the number of communication channels between modules

Few Interfaces Rule – 3

Every module should communicate with as few others as possible

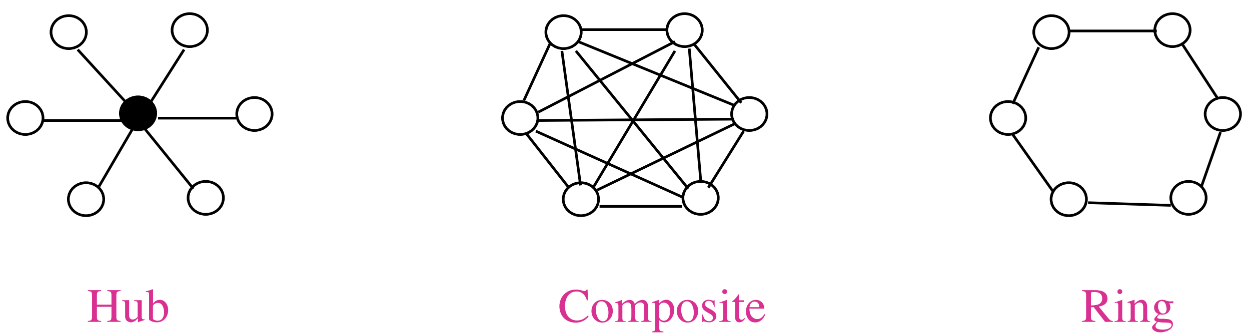
- Restrict the number of communication channels between modules
- Arises from **protection**, **continuity**, **composability**, **decomposability** and **understandability**

© Gunnar Gotshalks

Few Interfaces Rule – 4

Every module should communicate with as few others as possible

- Restrict the number of communication channels between modules
- Arises from **protection**, **continuity**, **composability**, **decomposability** and **understandability**



© Gunnar Gotshalks

Small Interfaces Rule

If two modules communicate, they should exchange as little information as possible

Small Interfaces Rule – 2

If two modules communicate, they should exchange as little information as possible

- Also known as **weak coupling**

© Gunnar Gotshalks

Small Interfaces Rule – 3

If two modules communicate, they should exchange as little information as possible

- Also known as **weak coupling**
- Relates to the size of connections rather than their number

© Gunnar Gotshalks

Small Interfaces Rule – 4

- Historical bad idea: Fortran COMMON block
  - » **COMMON block1 A[75], B[25]**
  - » **COMMON block1 C[50], D[50]**
    - > **View memory in two different ways!**

© Gunnar Gotshalks

Small Interfaces Rule – 5

- Local variables via Algol-60 block structure

var i

Access all variables in outer block
i := i + 5

© Gunnar Gotshalks

Explicit Interfaces Rule

Whenever two modules A and B communicate, this must be obvious from the text of A or B or both

© Gunnar Gotshalks

Explicit Interfaces Rule – 2

Whenever two modules A and B communicate, this must be obvious from the text of A or B or both

- Conversation is limited to a few participants and only a few words

Explicit Interfaces Rule – 3

Whenever two modules A and B communicate, this must be obvious from the text of A or B or both

- Conversation is limited to a few participants and only a few words
- Conversations are **loud** and **public**

© Gunnar Gotshalks

Explicit Interfaces Rule – 4

Whenever two modules A and B communicate, this must be obvious from the text of A or B or both

- Conversation is limited to a few participants and only a few words
- Conversations are **loud** and **public**
- Really important with respect to **understandability**

© Gunnar Gotshalks

Explicit Interfaces Rule – 5

Whenever two modules A and B communicate, this must be obvious from the text of A or B or both

- Conversation is limited to a few participants and only a few words
- Conversations are **loud** and **public**
- Really important with respect to **understandability**
- Worry about procedure **parameters** as well as **shared data**

© Gunnar Gotshalks

Information Hiding Rule (Parnas 72)

The designer of every module must select a subset of properties as the official information about the module, to be made available to client modules

© Gunnar Gotshalks

Information Hiding Rule (Parnas 72) – 2

The designer of every module must select a subset of properties as the official information about the module, to be made available to client modules

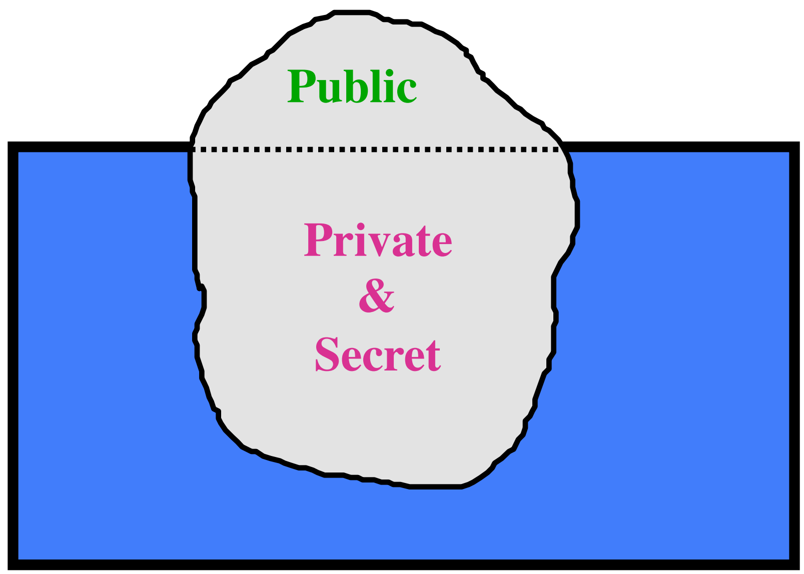
- Only **some**, but **not all** of the module's properties are public; the rest are secret

© Gunnar Gotshalks

Information Hiding Rule (Parnas 72) – 3

The designer of every module must select a subset of properties as the official information about the module, to be made available to client modules

- Only **some**, but **not all** of the module's properties are public; the rest are secret
- Public ≡ **interface**



© Gunnar Gotshalks

Software Construction Principles

- **Linguistic Modular Units Principle**
  - » **Modules must correspond to syntactic language units**

© Gunnar Gotshalks

Software Construction Principles – 2

- Linguistic Modular Units Principle
  - » Modules must correspond to syntactic language units
    - > Example: in C can include files that begin or end with a partial if…then…else statement

© Gunnar Gotshalks

Software Construction Principles – 3

- **Linguistic Modular Units Principle**
  - » **Modules must correspond to syntactic language units**
    - > **Example: in C can include files that begin or end with a partial if…then…else statement**

- **Self-Documenting Principle**
  - » **Module designers should make all information about the module part of the module itself**

© Gunnar Gotshalks

Software Construction Principles – 4

- Linguistic Modular Units Principle
  - » Modules must correspond to syntactic language units
    - > Example: in C can include files that begin or end with a partial if…then…else statement

- Self-Documenting Principle
  - » Module designers should make all information about the module part of the module itself
    - > Ideally all program text, diagrams, mathematics and explanations are in one file

© Gunnar Gotshalks

Software Construction Principles – 5

- **Uniform Access Principle**
  - » **All module services should be available through a uniform notation, which does not betray whether they are implemented through storage or computation**

Software Construction Principles – 6

- Uniform Access Principle
  - All module services should be available through a uniform notation, which does not betray whether they are implemented through storage or computation
  - Allow implementer to make space-time tradeoffs

© Gunnar Gotshalks

Software Construction Principles – 7

- **Uniform Access Principle**
  - » All module services should be available through a uniform notation, which does not betray whether they are implemented through storage or computation
  - » Allow implementer to make space-time tradeoffs
- **Single Choice Principle**
  - » Whenever a system must support a set of alternatives, one and only one module in the system should know their exhaustive list

© Gunnar Gotshalks

Software Construction Principles – 8

In real projects
A module needs to be both open and closed!

© Gunnar Gotshalks

Software Construction Principles – 9

In real projects
A module needs to be both open and closed!

- Open-Closed Principle
  - Open
    - Available for extension – add new features

© Gunnar Gotshalks

Software Construction Principles – 10

In real projects
A module needs to be both open and closed!

- Open-Closed Principle
  - » Open
    - > Available for extension – add new features
  - » Closed
    - > Available for client use – stable in spite of extensions

© Gunnar Gotshalks

Software Construction Principles – 11

In real projects
A module needs to be both open and closed!

- Open-Closed Principle
  - » Open
    - > Available for extension – add new features
  - » Closed
    - > Available for client use – stable in spite of extensions
- We are never done with extensions

© Gunnar Gotshalks

Software Construction Principles – 12

In real projects
A module needs to be both open and closed!

- Open-Closed Principle
  - » Open
    - > Available for extension – add new features
  - » Closed
    - > Available for client use – stable in spite of extensions
- We are never done with extensions
- We must make modules available to others

© Gunnar Gotshalks

Software Construction Principles – 13

In real projects
A module needs to be both open and closed!

- **Non OO languages**
  - » **Close when stability is reached, reopen when necessary**

© Gunnar Gotshalks

Software Construction Principles – 14

In real projects
A module needs to be both open and closed!

- Non OO languages
  - » Close when stability is reached, reopen when necessary
  - » But need to reopen all the clients as well

© Gunnar Gotshalks

Software Construction Principles – 13

In real projects
A module needs to be both open and closed!

- **Non OO languages**
  - » **Close when stability is reached, reopen when necessary**
  - » **But need to reopen all the clients too**
  - » **Inheritance offers a solution to this problem**

© Gunnar Gotshalks

Software Construction Principles – 14

In real projects
A module needs to be both open and closed!

- **Non OO languages**
  - » **Close when stability is reached, reopen when necessary**
  - » **But need to reopen all the clients too**
  - » **Inheritance offers a solution to this problem**
    - > **But only with multiple inheritance**