



**"Skill-driven" Code Documentation**  
(Compatible with code generated from "Skilltree" design)  
**written by Avner Ben**  
Version 3  
21-July-2008

**Contents**

Contents .....	1
General .....	2
Location .....	3
Structure .....	4
Definitions.....	5
Methodical guidelines .....	11
Advanced .....	13
Appendix 1. C++ Example .....	14

## General

1. **Purpose.** This paper specifies a formal language for expressing in-code documentation – "Skill-Driven" documentation.
  - 1.1. "Skill-Driven" documentation suits any procedural/object-oriented language that has the "method" or "function" programming construct (e.g. C, C++, Python, C#, Java).
  - 1.2. Originally, "Skill-Driven" documentation specifies the syntax of the documentation generated with the code by the open *Skilltree*<sup>1</sup> compiler. In addition, this syntax applies naturally to code that is written manually (and independent of any design methodology), with the benefit of in-code documentation that is...
    - 1.2.1. ...highly *structured*,
    - 1.2.2. ...unambiguous,
    - 1.2.3. ...easy to maintain by *different* developers,
    - 1.2.4. ...requires no training to understand,
    - 1.2.5. ...links the code to the SRS (Software requirement Specification) and HLD (High Level Design).
2. **Scope.** This paper specifies *method* documentation. Other document-able items (not described here) are component, class, association, and code blocks in method.
3. **Terminology.** The word "method" refers here to both "member" function (in a class) and "global" function (outside any class, as in C).
4. **Typographical conventions.**
  - 4.1. **Code.** Code excerpts appear in non-proportional type. Multi-line examples appear in green.
  - 4.2. **Documentation keywords.** Keywords of the documentation language (e.g. "Input:") are capitalized here, for convenience. (An equally valid alternative is to use all capitals).

---

<sup>1</sup> Hence the qualifier "*skill-driven*".

## Location

1. A method descriptor consists of a sequence of commented lines above the method.
2. When the method is distributed between a header file and an implementation file, (as in C++)...
  - 2.1. The header file contains the complete descriptor.
  - 2.2. The implementation file (optionally) contains the brief descriptor. It must not *repeat* the complete descriptor

## Structure

1. A method descriptor consists of the following parts:
  - 1.1. "**Brief**" **descriptor** - consists of a single element:
    - 1.1.1. "Required skill". A single line in the "Required Skill Form".
    - 1.1.2. The brief descriptor is mandatory!
  - 1.2. (*Optionally*) "**Detailed**" **descriptor** - consists of multiple items in the *type-and-value* form, each item on a separate line.
    - 1.2.1. The detailed descriptor lines follow the brief descriptor line (without header or separator).
      - 1.2.2. By default, each detailed descriptor item consists of item type, followed by colon, and descriptor text. E.g.
 

```
// to report Attendance.
// - Input: Work Session data
// - Output: new Attendance
// - Todo: to handle attendance range that spans two days!
void reportAtnd(const WorkSessionData&);
```

 In this example, the method "reportAtnd" implements the required skill "to report Attendance" and is described by three items - "Input", "Output", and "Todo".
      - 1.2.3. There are three categories of detailed descriptor items, listed in the following sequence (without category headers):
        - 1.2.3.1. **Usage contract**. This is the very structured method descriptor. The usage contract consists of so many usage **rules**.
          - 1.2.3.1.1. There are three rule categories, whose item types use predefined keywords:
            - 1.2.3.1.1.1. **Transform** rules - item types **Input** and **Output**.
            - 1.2.3.1.1.2. **Transition** rules - item types **Start**, **End** and **Effect**.
            - 1.2.3.1.1.3. **Premises** - item type **Premise**.
          - 1.2.3.1.2. A usage rule consists of item type followed by colon, (optionally, between brackets) rule qualifiers), free-text descriptor, and (optionally, between brackets) invalidation clause.
            - 1.2.3.1.2.1. Available rule qualifiers are "**optional**", "**formal**", and "**generic**", subject to context.
            - 1.2.3.1.2.2. An **invalidation** clause consists of the keyword **Invalid**, (optionally) **condition** text, and (optionally) remedy.
              - 1.2.3.1.2.2.1. The remedy is separated from the Invalid keyword or condition by colon.
          - 1.2.3.1.3. Usage rules of the same item type may be grouped and sequence-numbered.
        - 1.2.3.2. (*Optionally*) **Unstructured** (free-text) descriptor items.
          - 1.2.3.2.1. For Doxygen compatibility, precede descriptor items (free, as well as usage rules) by hyphen.
        - 1.2.3.3. (*Optionally*) Formal **arguments** and **return** type descriptors.
          - 1.2.3.3.1. These lines are typically intended for documentation parsing programs (e.g. Doxygen, JavaDoc) and should follow the syntax required by them.

## Definitions

### 1. Required skill

#### 1.1. Form:

##### 1.1.1. To [strong verb] [rest of sentence].

E.g. "to report Attendance", "to allocate Message Chunk", "to identify collision between Game Entities", "to prepare Message for sending".

This is a short form of "The system shall be capable to [strong verb] [rest of sentence]", e.g. "The system shall be capable to identify collision between Game Entities".

##### 1.1.2. A required skill should contain a **single verb**, i.e. should suggest a single decision.

#### 1.2. Name uniqueness.

Required skills are *uniquely named* in product scope. It is not enough for a skill to be unique in the context of its entity. E.g. "to get the state of The Game" is correct. "To get state" (required of "The Game") - is incorrect.

#### 1.3. Internal references.

The names of *required entities* inside required skills (e.g. the receiver object) should be highlighted by capitalization, e.g. "to compute Roadmap statistics" (a skill of Roadmap), "to count pending Appointments in the Calendar" (involving the entities Appointment and Calendar), "to set receiver id in Message Header".

##### 1.3.1. Typically, the highlighted name is the receiver object's type: "to do something to an Object"

#### 1.4. Special skills.

Common functionality invites common skill verbs. This makes the design legible and frees the reader's attention to the problem domain. *Especially...*

To initialize [entity]	constructor
To finalize [entity]	destructor
To (implement) [rest of skill]	virtual function override
To iterate in [entity]	begin/end iterator-producing methods
To get [data] from [entity]	property getter
To set [data] in [entity]	property setter
To (remote) [rest of skill]	Proxy (pass through) skill

### 2. Usage contract:

#### 2.1. The usage contract is an unordered list of usage rules, grouped by category.

##### 2.1.1. Multiple items of the same category are **grouped** in the same line, ordered by sequence number, e.g.

```
// to punch attendance card.
// - Input: (1) Employee id, (2) date
//   (3) entry/exit code, (4) hour
// - Output: new Card Punch entry
```

##### 2.1.2. Continuation

lines follow naturally; there is no continuation prefix. The first documentation line - the skill - begins with "to". Proper documentation lines below begin with a word followed by colon, meaning either usage-rule type or free documentation item. Any other line must continue the line above, e.g.

```
// - Input: (1) Employee id, (2) date,
//   (3) entry/exit code, (4) hour
```

- 2.1.3. Usage-rule **emphasis**. Automated documentation tools (such as Doxygen) ignore line breaks and wrap the entire "detailed" documentation into a continuous paragraph. To make Doxygen print usage rules in an unordered list (as "bullets"), prefix them with hyphens. Also, to make Doxygen distinguish "brief" documentation (i.e. skill) from "detailed" documentation (i.e. usage contract etc.), terminate the skill by period (or blank line), as in...

```
/// to punch attendance card.
/// - Input: (1) Employee id, (2) date
///   (3) entry/exit code, (4) hour
/// - Output: new Card Punch entry
```

- 2.1.4. **Boolean conditions**. By default, consecutive rules are **and'ed**, meaning *all* rules must fire. When *any* one rule will do, they may be explicitly **or'ed**, e.g.

```
// to reclaim unused memory.
// - Start: (1) memory pool near full
//   or (2) explicit garbage collection request
//   or (3) system idle
```

- 2.1.5. Avoid complicated Boolean equations!

## 2.2. Item **adornments**.

- 2.2.1. **Optional** rules may be preceded by "(optional)" (also "option" and "optionally") meaning that the rule (e.g. input data) is not always relevant to the algorithm. For example, not all input is always used, not all output is always produced, not all start conditions must fire. A frequent example is a search method, which may or may not find the requested item, as in...

```
// - Input: key
//   Output: (optional) value
```

- 2.2.1.1. In grouped rules, the qualifier is merged with the sequence number, as in "(1, optional)".

- 2.2.2. Transform rules (only) may also be qualified by "(formal)".

- 2.2.2.1. A **formal** input rule means method **argument**, e.g. "Input: (formal) key".

- 2.2.2.1.1. The *programmatic* meaning of **optional parameter** is type dependent, e.g. zero, empty string, and null pointer. In Python, C and C++, this will be any type to which *operator not* may be applied.

- 2.2.2.2. A formal output rule means method **return**, e.g. "Output: (optional, formal) value".

- 2.2.2.2.1. The *programmatic* meaning of **optional return** is type dependent, e.g. null pointer (or the None object) meaning "not found" in a search method.

- 2.2.3. Input rules (only) may also be qualified by "(generic)", for generic template arguments.

- 2.2.3.1. "Generic" (template) parameter is yet another parameter<sup>2</sup>, (best, qualified by "type"). E.g.

```
// to parse basic type.
// - input: (1, Formal) Parsing Stream,
//   (2, Formal) token (not set), (3, Generic) token type
```

---

<sup>2</sup> For example, in a dynamic language (like Python), it may indeed be yet another parameter.

```
// - Output: (Formal) token (set)
template <class TokenType>
TokenType& parseBasicType(ParseStream&, TokenType&);
```

2.2.3.2. "Generic" does not apply to output (at least not in C++).

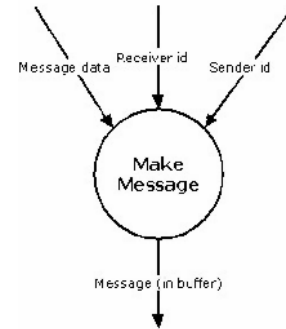
2.2.4. When the detailed descriptor contains explicit "param" and "return" items (e.g., Doxygen, JavaDoc), the *formal* transform rules are redundant. In this case, they may be merged with the respective item's descriptor.

**2.3. Rule categories.**

2.3.1.1. **Transform.** Transform rules define a required skill as a transform of *input* data, into *output* data, such as "to merge", "to separate", "to compute" etc., e.g.

```
// to prepare Message for sending.
// - Input: (1) Message Data,
// (2) sender id, (3) receiver id
// - Output: Message Block
```

In this case, the transform contract completely removes the need to describe the algorithm inside the method.



2.3.1.1.1. **Invalidation** (from transform rules).

2.3.1.1.1.1. What to expect when the input is invalid? E.g.

"Input: name (Invalid: error)" - when name is invalid (Not found? Illegal?), the standard error procedure is performed (e.g., exception).

"Input: entry id (Invalid - not found: insert entry)" - a blank entry is inserted for this id, if not found.

2.3.1.1.1.2. What to expect when the output may not be produced,

e.g. "Output: entry (Invalid: null)",

"Output: message (in buffer) (Invalid: message truncated)".

2.3.2. **Transition.** Transition rules define a required skill as asynchronous transition *signaling an event (for other skill)* instead of invoking them or synchronized by event (instead of being called). E.g.

```
// to terminate Rendezvous.
// - Effect: rendezvous owner signaled
// ...
// to free all waiting Consumers.
// - Start: Rendezvous owner signaled
```

2.3.2.1. Synchronization technology is irrelevant at this point, e.g. Synchronous function call, asynchronous message, polling.

2.3.2.2. "Start" rules tell when a thread of control enters reactive state. It is typical of "reactive" designs, e.g. the "Observer" pattern (also "Subscribe/Publish"). E.g.

```
// to refresh View.
// - Start: View observers notified
// ...
// to publish Document update.
// - Effect: View observers notified
```

2.3.2.2.1. The subscriber's skill "to refresh view" is triggered when the state of the system changes to "View observer's notified". One known source for this event is the publisher's skill "to publish Document update". (Exactly how the two are synchronized is

beyond the scope of either of them, e.g. synchronous function call, asynchronous message, etc. ).

- 2.3.2.3. "End" rules tell when a thread of control exits "wait" state (possibly entering reactive state). These occasions are rare in whole method descriptors (since "to wait" is seldom a significant external skill). Still, an occasional "end" rule may be essential, in order to inject sense to the documentation of an otherwise arbitrarily fragmentary design, e.g.

```
// to wait for other Participants to notify .
// - End: Rendezvous signaled
```

- 2.3.2.4. **Invalidation** (of transition rule). What to expect when the rule fails to fire.

- 2.3.2.4.1. What to expect when "end" rule fails to fire? (Typically, timeout).

- 2.3.2.4.2. What to expect when "start" rule or fails to fire? Typically, user-determined timeout through polling, E.g.

```
// to process Acknowledgement.
// - Start: Ack Message arrived
//   (Invalid - timeout: re-send)
// - Note: Ack timeout tested in each cycle
//   (if pending)
```

- 2.3.2.4.3. What to expect when "effect" rule fails to fire? (Typically, OS failure).

```
// to send message.
// - Effect: Message waiting in queue
//   (Invalid: system restart)
```

- 2.3.3. **Premises**. Axiomatic **premises** that are not immediately connected with any particular transform or transition rule and are taken for granted in the implementation of the required skill.

- 2.3.3.1. The contract has two sides:

2.3.3.1.1. Transforms and transitions are "signed" by the *receiver*.

2.3.3.1.2. Premises are "signed" by the *caller*.

- 2.3.3.2. **Invalidation** (of premise).

- 2.3.3.2.1. If the premise has no invalidation, then the user is thereby warned that failing to meet the premise (prior to invoking the method) will result in *undefined behavior* of the software product, e.g.

```
// to free all waiting Consumers .
// - Premise: Invoked from the Producer thread
```

There is no way to validate this during runtime; e.g.

```
// to identify long comment line.
// - Premise: Comments are not nested
// - Example: "/* ... /* ... */ ... */" will not count 2!
```

Otherwise, the result is undefined; e.g.

```
// to display window.
// - Premise: the window is fully constructed
```

Otherwise, the user sees nothing on the screen.

- 2.3.3.2.2. In some cases, the results *are* defined, but may or may not be *desirable*. e.g.

```
// to find entry.
// - Premise: Entries are unique
//   (Invalid: the first one found)
```

It is the caller's responsibility not to put duplicates in this table!

2.3.3.2.3. In some cases the algorithm actually validates the premise, optionally resulting in some reaction, which may be a predefined "error" procedure (such as abort, exception) or a corrective action (such as replacement by default value) , e.g.

```
// to print report.
// - Premise: Database is non-empty
//   (Invalid: "no data" standard message)
```

You will always see something on the screen, even if there is nothing to show!

2.3.3.2.4. Typical invalidation reactions:

2.3.3.2.4.1. **"Error"**. Project-standard error procedure, e.g. to throw exception.

2.3.3.2.4.2. **"Ignored"**. The method skips the input-related processing or returns prematurely (further processing rendered irrelevant due to processing pre-conditions not being met).

2.3.3.2.4.3. **Explicit reaction** specified.

2.3.3.2.4.4. The default reaction is "ignored".

2.4. An **invalidation** clause at the end of the rule (optionally) specifies a failure condition and remedy or other action .

2.4.1. Typical actions are:

2.4.1.1. **Error** – The product-specific standard error procedure, typically, to throw exception.

2.4.1.2. **Ignored** – the method returns prematurely.

2.4.1.3. An explicit **action**.

2.4.2. The default remedy is "ignored".

2.5. **Structured usage rules**. A usage rule consists of two parts: the name of the *object* that is used and its *state*. The state is within brackets. This allows transformations to be followed.

2.5.1. Structured output.

2.5.1.1. The skill is for **creating** an object: just `Output: object .`

2.5.1.2. The skill is for **changing** an existing object: `Output: object (state)`. The state must be significant, e.g. "ready", "open", "approved" etc. Some skills will only take objects at this state as input.

2.5.1.3. The skill is for **deleting** an existing object: `Output: object (state)`. Although deletion does not make any tangible product, it does create a state that potential users of the object must consider. The state may be more than just "deleted". It may involve the position of the object in the system, e.g. "closed" (referring to the resource that the object used to encapsulate).

2.5.1.4. *Example:*

```
// "File Wrapper" - to access disk file our way
Class FileWrapper
{
    // to initialize File-Wrapper.
    // - Input: (formal) file name
    // - Output: file (open
    FileWrapper(const char* fileName);
    // to initialize File Wrapper by copy.
    // - Input: (formal) File Wrapper
    // - Output: (optional) file (shared)
    FileWrapper(const FileWrapper& rhs);
```

```

// to finalize file-wrapper.
// - Output: (optional) file (closed)
~FileWrapper;
// to close wrapped file.
// - Output: (optional) file (closed)
close();
//...
};

```

- 2.5.1.5. The "constructor" (first function) always opens the file. At a later time, the file may be.
  - 2.5.1.5.1. The "copy-constructor" for another object (second function) increments the reference count and does not open a file. The file is shared (if open).
  - 2.5.1.5.2. The "destructor" (third function) decrements the reference - count and possibly closed the file (if it was open).
- 2.5.2. The input/output may refer to either the object used (or produced) or the store (or actor) that accepts it (or gives it).

*Both these cases are valid:*

```

// to register setup item.
// - Input: (1) key, (2) value
// - Output: setup entry (in The Registry)

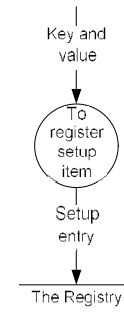
```

*and...*

```

// to register setup item.
// - Input: (1) key, (2) value
// - Output: The Registry (updated)

```



## Methodical guidelines

### 1. Required skill.

- 1.1. The **purpose** of the required skill is to map one *functional requirement* definition (e.g. from the SRS - Software Requirement Specification - document) to the software product and *back*.
    - 1.1.1. **Rationale**: Object-oriented design is about *collaboration* among entities – organizing teamwork among role-players, where each is characterized by the unique "*skill*" it donates to the team (a.k.a. its "secret") and is detailed by minor skills, expressed in (and inside) its methods. The latter are typically about initializing, finalizing, changing state, getting/setting properties and other detailed services. Each method must demonstrate that it fits in this scheme.
  - 1.2. The **required skill form**. Avoid dropping the prefix "to" from the required skill, such as using imperative statement (e.g. "publish Document update") or noun (e.g. "Document-update publishing"). The required skill is first of all a requirement - something that the product must be *capable to do* - and only secondarily a method descriptor.
  - 1.3. Required skills are **unique** in the entire product, to allow being reverse-engineered back into the SRS of the product, and thus become available for product owner – or peer - review. These parties make requirements of the entire product. On the contrary, the idea of collecting required skills under entities – i.e. methods under classes - is a programmatic design technique which is methodically sound, but may or may not appeal to the product owner.
  - 1.4. **Scope**. The required skill is part of the *problem domain*, even though it sits deep in the solution. It should state the exact donation of the software to the *required functionality* of the product. "to count pending Appointments in the Calendar" is a formula, but it also implements a tangible requirement from the product that the user may agree with. "to sum into the pending Appointment counter in the Calendar" requires some detective work to understand, because it is more solution - than problem - oriented. "to update calPendApptmtCounter" is outright programmatic Gibberish.
    - 1.4.1. Telling problem from solution may takes some intelligence, because software architectures typically feature layers of abstraction. E.g. since the problem domain of an *infrastructure* library is programmatic, its required skills are programmatic as well (e.g. "to store objects in a pool") and do not necessarily appeal to the original *product* owner. (The product owner, in this case, is another developer) . The required skills are (at least) *one level higher* than the code.
    - 1.4.2. The **objective** is to prevent the code from running astray and start serving its own requirements!
- ### 2. Usage contract.
- The usage contract is an extension of DBC ("Design By Contract" – after B. Meyer). It extends DBC by (1) classifying rule categories and (2) being on design (rather than programming) level.
- 2.1. The **objective** of extended DBC is to define the method's behavior *completely*, without giving away the secrets of its implementation (such as the algorithm inside it).
    - 2.1.1. The **premise** of extended DBC is that careful use of this syntax almost completely *eliminates* the need for both algorithmic and ambiguous, unstructured, free text descriptors.

2.1.2. The complete code documentation creates a virtual solution domain that insulates the user from the real solution . E.g. in...

```
// to post offline request.
// - Output: Pending request queue
// (with our request at the bottom)
```

...the system will behave as if it has a queue of pending requests, somewhere. Does it? It is not for us to tell!

2.2. The power of brevity. Do not scream; specify input and output!  
For example, The following piece of conventional documentation is screaming!

```
// This is the constructor (in case you didn't notice).
// WARNING! Pool memory must be allocated by the user!
// WARNING! We don't allocate it!
// Y O U H A V E B E E N W A R N E D ! ! !
```

```
Pool(void* memoryBlock);
```

The following version does it by specifying the fact of input.

```
// "to initialize Pool".
// - Input: pre-allocated memory block
Pool(void* memoryBlock);
```

2.2.1. Those who continue to walk by the dry spec will not be impressed by the screaming, either!

2.3. **Grouping usage-rule conditions.** Do not invest in complex rule systems involving nested and/or conditions. These are typically the sign of poorly-partitioned skills!

2.4. **Parameters and return** specification.

2.4.1. The documentation author has to decide whether to do specify formal input/output rules in the presence of param/return, or avoid the redundancy.

2.5. Free-text **descriptors**.

2.5.1. Free-text descriptors should be left for information that may not be expressed in the usage contract (and may not be left out), such as...

2.5.1.1. The **"official"** descriptor – use item name "Function:" e.g.

```
// to reclaim unused memory.
// - Function: The "garbage collection" algorithm
```

2.5.1.2. **TBD** (To Be Defined) item. Functionality as yet under analysis, "To-do" item. (Remember to remove when done!)

```
// to report Attendance.
// - Input: Work Session data
// - Output: new Attendance
// - Todo: to handle attendance that spans two days!
```

2.5.1.3. **Rationale** items.

2.5.1.3.1. Constraints that necessitate this (otherwise strange) logic.

```
// to initialize Window
// Output: Window (not ready)
// Note: First phase in 2-phase initialization
process!
```

2.5.1.3.2. Compiler/OS/Infrastructure/language issue workaround.

2.5.1.3.3. Where used (as originally intended).

```
// to tell storage size required by pool.
// - Used by: The Global Pool to allocate pool
storage
```

2.5.1.3.4. Performance/efficiency facts.

## Advanced

1. The following pointers are of interest to *Skilltree* users who update in-code documentation with the aim of reverse-engineering the design:
  - 1.1. **Coupling** analysis. Static analysis of the design is performed by matching the contents of complementary usage rules.
    - 1.1.1. **Data coupling** - matching the input of one required skill with the output of another.
    - 1.1.2. **Event coupling** - matching the start/end asynchronous trigger of one required skill with the effect of another.
    - 1.1.3. Coupling analysis ignores the *bracketed part* of the descriptor. This allows to specify stages in the transform and still enable coupling, e.g.

```
// to do step 1.  
// - Output: buffer (reset)  
// ...  
// to do step 2.  
// - Input: (1) buffer, (2) Message Block  
// - Output: buffer (with message block)
```

The two transforms apply to the same object, called "buffer".

- 1.2. Inside the method (not discussed here), each rule invalidation generates one "Invalid [usage rule type] [usage rule]" documentation line, followed by conditional statement (meaning premature exit from the method or block).
- 1.3. Some of the special skill verbs cited above (i.e. "to initialize...", "to finalize...", and "to iterate") are actually used by the Skilltree compiler. The rest are in planning.

## Appendix 1. C++ Example

The "Chunk memory-Pool" class:

```

001: // "Chunk Memory Pool" -
002: // -----
003: // to allocate fixed-sized memory chunks
004: // with efficient deallocation
005: // Method: Availability stack
006: // Efficiency: Maximum
007: class ChunkPool
008: {
009: public:
010:
011:     // to tell storage size required by chunk pool
012:     // - Input: (1, formal) number of chunks, (2, formal) chunk size
013:     // - Output: (formal) required memory block size
014:     // - Purpose: used by master pool to allocate overall storage
015:     static size_t requiredStorage(size_t size, size_t chunkSize)
016:     { return (size + sizeof(void*)) * chunkSize; }
017:
018:     // to initialize Chunk Pool.
019:     // - Input: (1, formal) pre-allocated storage block,
020:     //   (2, formal) number of chunks, (3, formal) chunk size
021:     // - Output: (1) memory block (divided to chunks),
022:     //   (2) available chunk stack (offering all,
023:     //   in order allocated)
024:     // - Premise: storage is sufficient
025:     ChunkPool(void* storage, size_t size, size_t chunkSize);
026:
027:     // to allocate memory chunk.
028:     // - Output: (1, formal) chunk pointer,
029:     //   (2) available chunk stack (offering previous)
030:     // - Premise: available (Invalid: error)
031:     void* AllocateChunk() throw(myException);
032:
033:     // to deallocate memory chunk.
034:     // - Input: (formal) returned chunk
035:     // - Output: Available chunk stack (updated with returned chunk)
036:     // Premise: (1) Chunk belongs here, (2) stack vacant
037:     void DeallocateChunk(void* chunkPtr) throw(myException);
038:
039:     // to get size from Chunk Pool.
040:     // - Output: (formal) number of chunks
041:     size_t getSize() { return size_; }
042:
043: private:
044:     ChunkPool(ChunkPool&);
045:
046: protected:
047:     // pool size (number of chunks) - for information
048:     size_t size_;
049:     // The available chunk stack
050:     void** chunkPtr_;
051:     // index used as top of stack
052:     size_t freeChunkCount_;
053: };

```