How detailed should a "detailed design" be? Enough detail to:

- ensure **appropriate function**
- resolve **ambiguities**
- enable **independent development**
- enable **unit testing**
Overall vs Detailed

Overall design: how modules "fit together".
Goal of overall design: maximize cohesion, minimize coupling
Detailed design: what does each module do?
Goal of detailed design: make implementation verifiable.
Boehm makes a careful distinction between "verification" and "validation".

**Verification**: "Are we making the **product right**?"
- I.e., are we conforming to **our own ideas** of what it should do?

**Validation**: "Are we making the **right product**?"
- I.e., are we conforming to **the customer's needs**?

Verification is about **correctness**, while validation is about **truth**.
The party line on verification: the detailed design should include

- **all interfaces** to each module in question,
- **preconditions**, **postconditions**, and **invariants** of each function

Why?
Software engineers wrestle a lot with "program correctness"

a module is **correct** if when used **as it is intended**, it works **as expected**.

it does not matter what it does **when misused**!

For a software developer, lots of bug reports start out, "I tried to do something the software says it won't do, and by George, it doesn't do it."

This is **not** a verification problem!
Program correctness for software engineers

**Preconditions:** what must be true before a function is called.

**Postconditions:** what will be true after a function is called, given that preconditions are met.

**Invariants:** things that remain true before and after each function call.

A set of preconditions, postconditions, and invariants for a function form a **contract** for a function.
Example: a linked list

function link-at-head

**Preconditions:**
L is a list, E is a new element

**Postconditions:**
L is modified so that E is now its head, with all other members preserved.

function unlink-from-head

**Preconditions:**
L is a list

**Postconditions:**
If L is non-empty, its head is removed and its other elements remain.
If L is empty, nothing happens.
Some facts about program correctness:

Except for the simplest programs, determining program correctness is **intractable**.

(It's also **undecidable**, but that refers to a machine with unlimited space!)

Limit of our knowledge: the "**principle of weakest preconditions**"
Weakest precondition analysis

Preconditions for a program form a set lattice. I.e., some are "weaker" than others.
A precondition X is stronger than a precondition Y if the set of all inputs matching X is a subset of the set of all inputs matching Y.

"x>1" is stronger than "x>0" and weaker than "x>2".
"x>1" and "y>2" have no relationship.

A program is correct iff the provable preconditions for the program are weaker than (or the same as) the stated preconditions.

"The principle of weakest preconditions"
Weakest precondition analysis in action

Weakest precondition analysis in action
start with **postconditions**
**work backward** one statement at a time.
When you get back past the beginning, you have
**provable preconditions**.
If these are weaker than or equivalent with **stated preconditions**, your program is **correct**.

Reference:
int inc(int x) {
    x = x + 1;
    return x;
}

Preconditions: x is any integer
Postconditions: return that integer plus 1

Analysis
int inc(int x) {
    // (2) { x = argument }
    x = x + 1;
    // (1) { x = argument + 1 }
    return x;
}

Thus
Given that result is argument + 1, derived preconditions are that input is argument, which is the same as stated preconditions.
Key to weakest precondition analysis: Hoare/Floyd Logic

A Hoare triple: \{before\} statement \{after\}

Several axioms:
- \{P\} nop \{P\}: if you do nothing, state doesn't change.
- \{P'\} x=y \{P\}: where P' is the result of starting with P and replacing x with y where it occurs.
  e.g., \{y=1\} x=y \{x=1\}
- \{P\} s1 \{Q\} and \{Q\} s2 \{R\}
  \-----------------------------
  \{P\} s1;s2 \{R\}
- \{P ^ C\} A \{Q\}, \{P ^ ¬C\} B \{Q\}
  \-----------------------------
  \{P\} if (C) then A else B \{Q\}
- \{C ^ I\} while (C) body \{¬C\}
  \-----------------------------
  \{I\} while (C) body; \{¬C ^ I\}
- \{P\} body \{Q\}, P' \rightarrow P, Q \rightarrow Q'
  \-----------------------------
  \{P'\} body \{Q'\}
The Achilles' heel of program correctness proofs: loops

- Cannot just jump over a loop in the same way.
- Must determine something that is a loop invariant, i.e., it is the same before and after the loop.
- The state before the loop is the loop invariant.
- Choosing a loop invariant is inductive, and equivalent to theorem proving.
catch-22: "damned if you do and damned if you don't"
preconditions and postconditions
suggest test cases.
in simple cases, can be proven.
in complex cases, proof may be impractical.
"There is no way to completely test a program."
The halting problem:
There is no way to predict whether a particular Turing machine will halt, other than to run it and observe it.

Is testing really the halting problem?

No, for an extremely counter-intuitive reason:
The proof of undecidability requires infinite state-space.
Any program running in a finite state-space can be analyzed; it is simply intractable to do so.

My favorite theory Ph.D. qualifying question: "Can a program running in a finite space be undecidable?"
Approaches to verification in the design phase

test cases: what should happen?
formal modeling: what should code "do"?
Fact: most algorithms are fairly simple speed requirements make them complicated. if we "leave out" the speed component, we can succinctly describe what should happen. if we "specify" internals in a suitably high-level language, we can show "what to do" without the "how".
The most prevalent formal modeling system: Z
pronounced "Zed"
a set-theoretic language
very high-level
intended for human consumption
primitives: set theoretic claims and logic.
Zed specifications

Zed types
N: integer
R: real number
{foo, bar}: enumerated set

Variable conventions
foo? : an input
bar! : an output
cat' : a resulting value of cat
Zed functions

Unlike normal programming languages, Zed "processes" are **assertions of state.**
Every storage object is a **function of its keys**
A function is a **set of associations.**

```plaintext
salary = { amy -> 500, george -> 200, frank -> 100}
```
sets salary to a function with elements
```plaintext
salary.amy = 500
salary.george = 200
salary.frank = 100
```

can augment functions
```plaintext
salary' = salary ⊕ { amy -> 600 } # give amy a raise
```
can remove elements of functions
```plaintext
salary' = { amy } ⊖ salary # omit amy from function salary.
```
Counter.x \( \Rightarrow \) Counter.x + 1

Changes container

Increment

\[ \Delta \text{Counter} \]

Counter.x' = Counter.x + 1
In Z, functions are just sets
  if \( F \) is a function
  \( \text{dom}(F) \) is its domain
  \( \text{rng}(F) \) is its range
An array is a function from integers to its container type!
A linked list is a function from its key type to its value type
All implementation details are lost!
Constraints and capacities

Wednesday, October 14, 2009
2:06 PM

Array

\[ \text{data} : \text{INDEX} \rightarrow \text{VALUE} \]
\[ \text{size} : \text{N} \]
\[ \text{INDEX} : \text{N} \]
\[ \text{VALUE} : \text{N} \]

INDEX \geq 0
INDEX < \text{size}

Set

\[ \text{Array} \]
\[ \text{ind}? : \text{N} \]
\[ \text{val}? : \text{N} \]

\[ \text{Array} \{ \text{data} \} ^{\oplus} \text{ind} \rightarrow \text{val} \]
\[ \text{md} \geq 0, \text{md} < \text{Array size} \]

add a value to a function, with override

\[ \text{hth element} : \]
\[ \text{Array.data} (n) \]
Purposes of Z

- Describe **what should be done** in a process without describing details of how
- Model changes as **assertions of state**.
- Allow **verification** of the assertions without knowing implementation details.
Location

lat: R
hem: & north, south
lon: R
dir: east, west

0 ≤ lat ≤ 90
0 ≤ lon ≤ 180
A map is a set of locations

Points: \( \text{NAME} \rightarrow \text{Loc} \)

NAME: soq chae
Loc: Location

Points is a set
\[ \{(\text{Name}, \text{Loc})\} \]

ordered pairs where there are unique Loc per Name.
Adding to a map

Wednesday, October 14, 2009
2:32 PM

Add Entry

\Delta \text{Map}

\text{name?}, seq, char

\text{loc?}, location

\text{map}.\text{Points}'

= \text{map}.\text{Points} \oplus \{ & \text{name?} \rightarrow \text{loc?} \}

\text{add name?} \& \text{dom} (\text{map}.\text{Points}) \Rightarrow \text{don't replace.}

Important:

Can control everything about this addition
Can say whether to duplicate entries or not.
Cannot say "how" to accomplish anything!
This disallows duplicate names!
Another method for exceptions

Add Entry

$\textbf{Map} \leftarrow$ does not change

(name?, seq_char, loc?, Location, error!, seq_char)

name? $\in \text{dom}(\text{Map Points})$

error! = "duplicate name not allowed"
The big deal of Z

- If one can express program steps as modifications to a state space, then
- although the steps themselves may be complex as set operations,
- the resulting program is just a sequence, and
- most loops are not represented in the sequence
- (other than the trivial overall loop)
- Thus Z specifications are easier to verify than those of a regular programming language....

- ... because of the details that Z omits!
Every Z specification corresponds to a Hoare/Floyd triple, where

| preconditions are statements in the constraints about input data. |
| postconditions are statements in the constraints about output data. |
| The operation is the whole Z clause. |

E.g.

\[
\begin{align*}
\Delta \text{Array} \\
\text{Ind}: \mathbb{N} \\
\text{Val}: \mathbb{N} \\
\text{Ind} \neq \text{Val} \\
\text{Ind} < \text{Array}.\text{Size} \\
\text{Array}' = \text{Array} \oplus \mathbb{N} \\
\text{Ind} \implies \text{Val} \rightarrow \text{Val} \end{align*}
\]
Z and design

Z allows one to model the intent of interfaces without giving details
One can express the semantics of a data storage medium without writing the code that enforces them.

Z strengths
- database transactions
- data integrity constraints
- defining tests

Z weaknesses
- doesn't model loops well
- doesn't model overall processes well
One easy use of Z: define tests
A Z specification gives the **conditions under which something should happen**.
If those conditions are not present, it **should not happen**.
This suggests **simple unit tests** from the Z specification itself.
Recall the array schema:

```
Array
  data: ind \rightarrow value
  ind?: N
  value?: N
  SIZE?: \mathbb{N}^+

\begin{align*}
  \text{md} &\geq 0 \\
  \text{md} &< \text{SIZE}
\end{align*}
```

```
set

\begin{align*}
\text{Array}\,\text{nd}?: N \\
\text{val}?: N
\end{align*}
```

```
\begin{align*}
\text{md} &\geq 0 \\
\text{md} &< \text{Array}\,\text{SIZE}
\end{align*}
```

\[
\text{Array}\,\text{data} = \text{Array}\,\text{data} \oplus \text{md} \Rightarrow \text{val}?
\]

So, how to test an array?
- case 1: \(0 \leq \text{ind} < \text{Array}\,\text{SIZE}\)
- case 2: \(\text{ind} < 0\)
- case 3: \(\text{ind} \geq \text{Array}\,\text{SIZE}\)

This is a simple example of **equivalence partitioning** (that we will study later).
Summary so far:

detailed design has several roles
to limit communication between groups
to circumscribe behavior so that there are no mistakes in implementation
to suggest unit tests of individual modules
to aid in verification of the final result
The detailed design document should inform implementors of:
the public interfaces required for modules.
planned interactions between work of different teams

The detailed design document should not describe software interactions within a team/module.
private interfaces used only within the module.
Container
Data

Constraints

$A \oplus B$ : new function with overrides for
$B$

$A \sqcap B$ : new function with values of $A \leq \text{dom}(B)$
omitted.

$N$: integers
$R$: real numbers

$f : x \rightarrow y$ function

$\Delta x$: $x$ changes
$x \times$: $x$ does not change