CS 4500
Software Development

Integration Testing

Ferdinand Vesely[1]

October 15, 2019

[1]Material based on *Code Complete* by Steve McConnell

Integration Testing

- We have some components that pass all unit tests
- We need to combine them together to form subsystems
- How do we integrate?
- How do we test?

Integration

Integration = combining separate software components into a single system

- Also: Combining software units into components
- Integrated components – added complexity of interactions
- Cascade of interdependencies
- If done poorly, problems can “explode” at once

Big-Bang Approach

Phased Integration

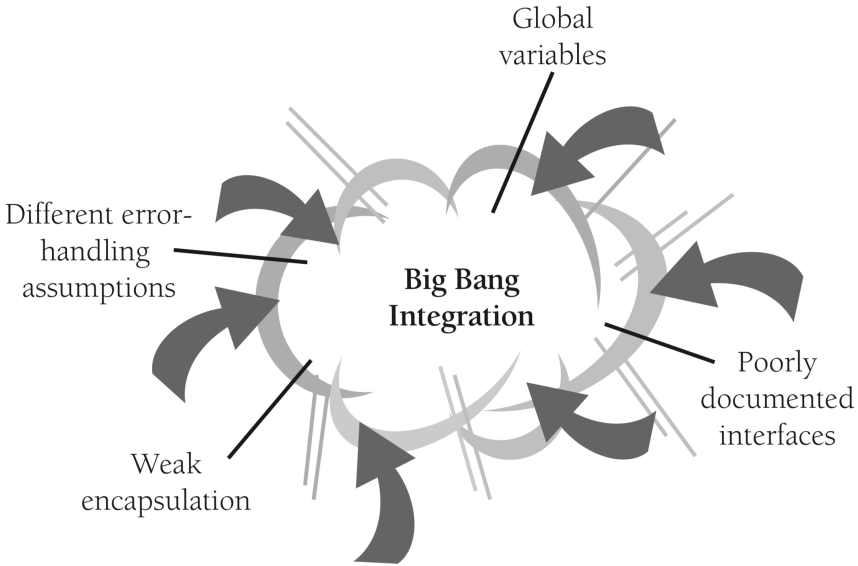
1. Unit development: Design, code, test, and debug each unit
2. System integration: Combine units into one big system
3. Test and debug the whole system

Big-Bang: Problems?

Big-Bang: Problems?

- After integration: new problems inevitably surface
- Causes could be anywhere
- All components are potential suspects
- Errors suddenly presented all at once
- Errors themselves might interact

Big-Bang: Problems?



Incremental Integration

"One piece at a time" approach

In general:

1. Develop a small, functional part of the system – skeleton
   - Thoroughly test and debug
   - Skeleton: attach the remaining parts of the system
2. Design, code, test, and debug a unit
3. Integrate new unit with the skeleton
   - Test & debug the combination
   - Ensure combination works before adding new components
4. Go back to 2 if components need to be added

Benefits of Incremental Integration

Benefits of Incremental Integration

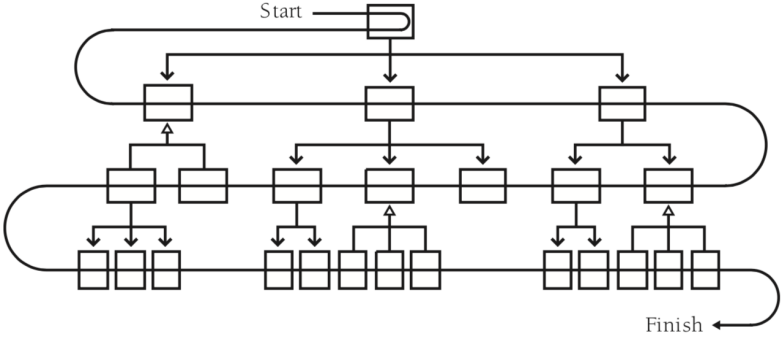
- Errors easier to locate and fix
- System success early
  - Always in a relatively working state
- Improved progress monitoring
- Units tested more fully

Incremental Integration Strategies

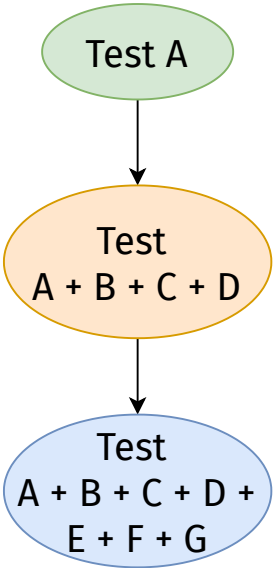
- Integration of some components – required before the integration of others
- Planning for integration – planning for construction
- Order of component construction has to support the order which they will be integrated

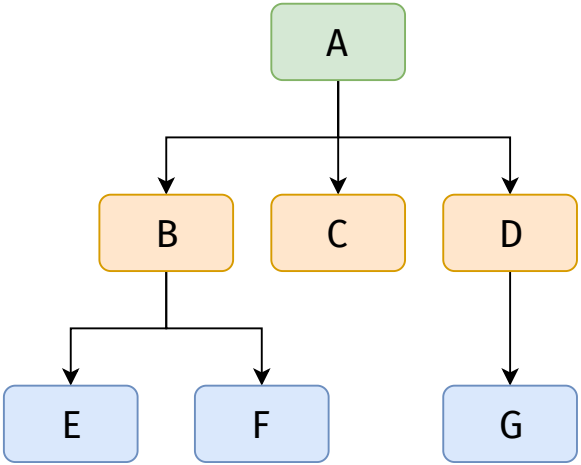
Top-Down Integration

- Unit/component at top of hierarchy written and integrated first
- Testing: *stubs* to exercise the higher placed components
- Stubs gradually replaced with actual units
- Important: Carefully specified interfaces
  - Avoid errors arising from subtle interactions



Top-Down





Pros / Cons

Pros

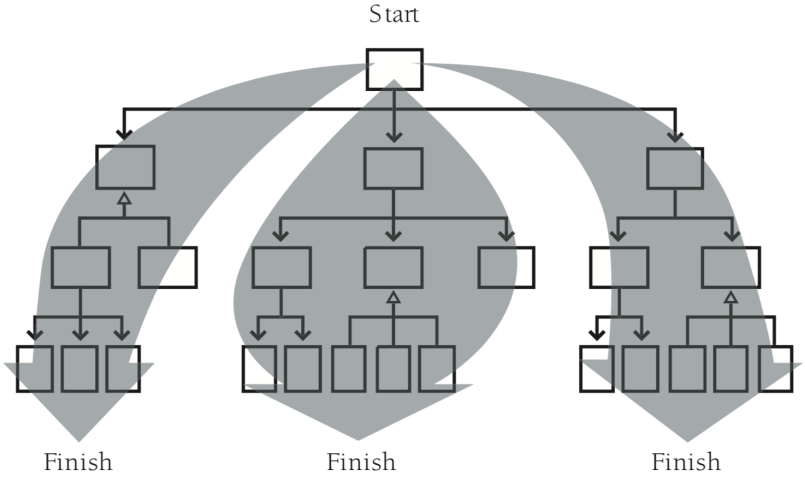
- Control logic of the system tested relatively early
  - Components at top of hierarchy exercised often – expose conceptual/design problems quickly
- Can complete a partially working system early
- Can begin implementing before low-level details are completed

Cons

- Tricky, low-level interfaces exercised last – can bubble up to the top
- Need to write *many* stubs
  - Stubs can contain errors
- Sometimes: What is the top?
- Pure top-down mostly doesn't make sense – hybrid approaches

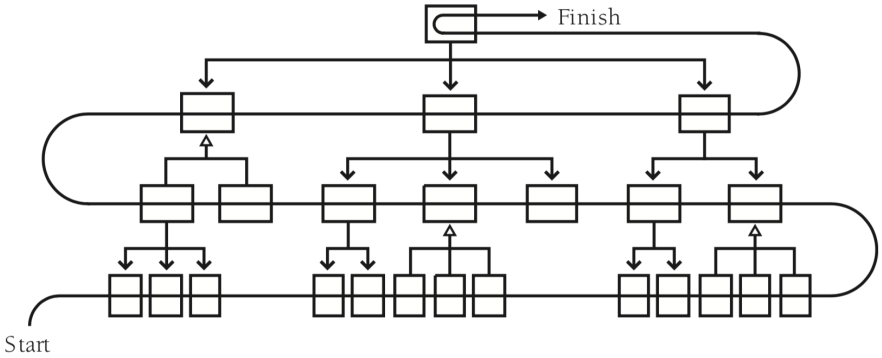
Top-Down Variant: Vertical Slice

- Work down in sections
- Fully flesh out a subsystem (functionality) before moving to the next

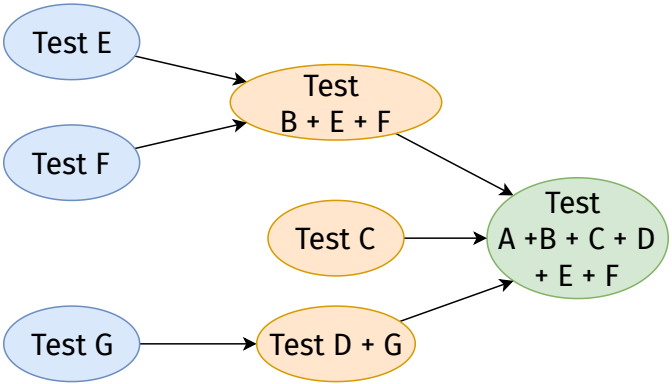


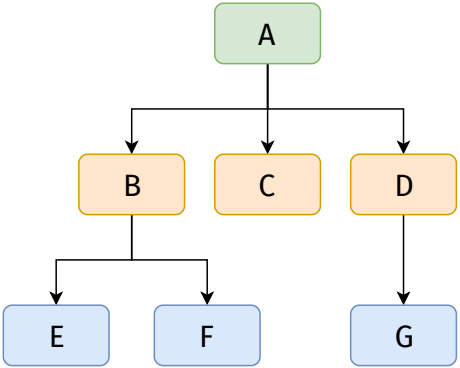
Bottom-Up Integration

- First: Implement and integrate components at bottom of hierarchy
- Add one component at a time
- Testing: *drivers* to exercise lower-level components
- Replace drivers with higher-level components as they are developed



Bottom-Up


Test E
Test F
Test
B + E + F
Test C
Test
A +B + C + D
+ E + F
Test G
Test D + G


A
B
C
D
E
F
G

Bottom-Up Integration

Pros

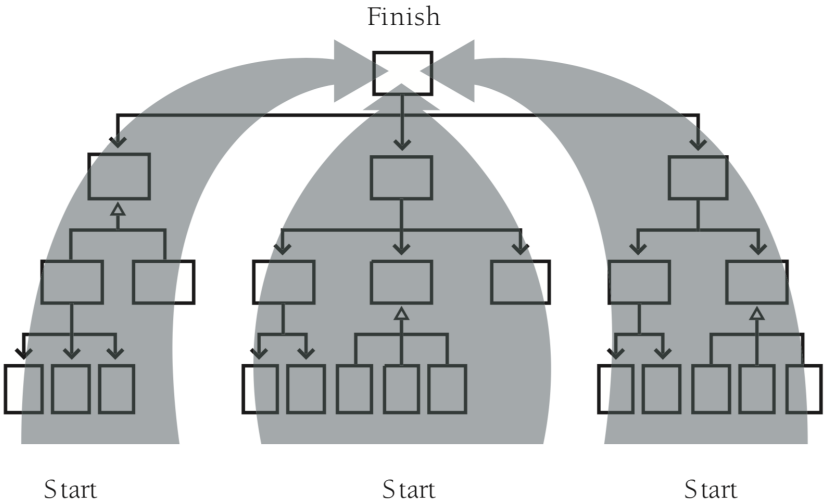
- Restricts possible source of error
  - The component being integrated
- Exercises potentially problematic interfaces early

Cons

- Integration of major high-level interfaces – last
- Conceptual design problems at higher levels:
  - discovered late
  - design changes: implementation/integration work might be discarded
- Design of the whole system – required before integration
  - Otherwise: might end up designing high-level components around problem in low-level ones
- Again, pure bottom-up often does not make sense

Bottom-Up Variant: Vertical slices

- Integrate subsystems bottom to up

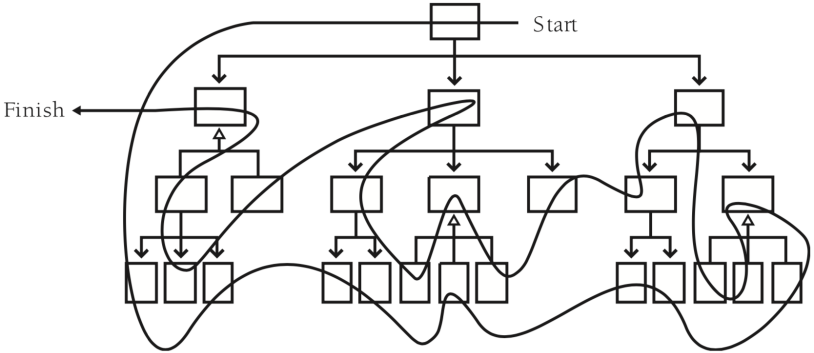


Problems with Both Top-Down and Bottom-Up

- Rigidity
- Not really reflecting practice
- Alternatives: Sandwich, Risk-oriented, Feature-oriented

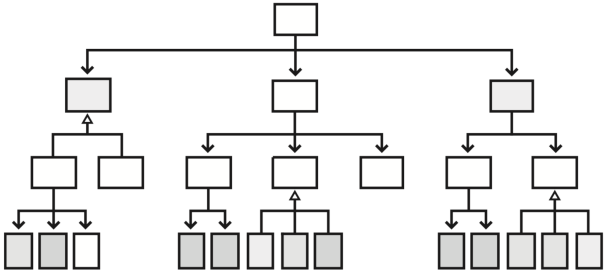
Sandwich Integration

- First: Integrate and test high-level components
- Then: Most important low-level components
- Finally: Integrate mid-level components



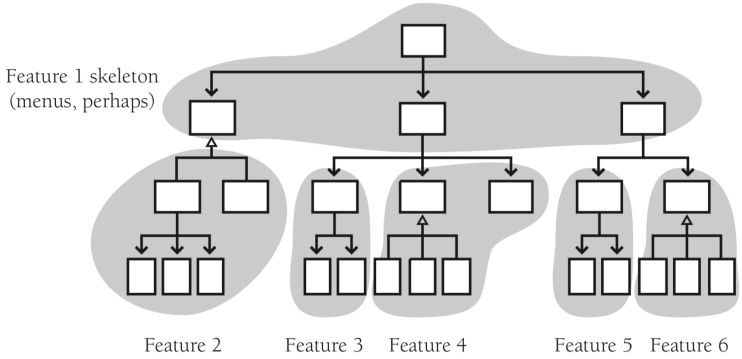
Risk-Oriented Integration

- “Hard part first” integration
- Identify level of risk associated with a component
- Implement most challenging first
- Usually top-level and bottom-level first



Feature-Oriented Integration

- Integrate a feature at a time
- Feature: identifiable function of the system
- Start with a skeleton (e.g., a menu system implementation)
- Add features to skeleton



Feature-Oriented Integration

- Feature might be bigger than a single unit/component
- Increment: may be bigger than a single component
  - Might reduce the certainty about location of errors
- Components – added as feature trees
- Integration easier if features relatively independent

Advantages

- Mostly eliminates scaffolding (stubs and drivers)
  - Skeleton might rely on some stubs
- Each new integrated feature: incremental addition to functionality
  - Evidence of progress
  - Functional software earlier

Daily Builds and “Smoke Tests”

Basically:

- Test integration frequently
- An executable is built every day[2]
- Perform a quick *smoke test* to see if the integrated program “smokes” when run
- Preferably automated

[2] As applicable

Daily Builds

- Daily build – “heartbeat” of a project
- Check for “broken” builds – strict enough to identify showstoppers, but does not draw attention to trivial defects
- Successful build:
  1. All relevant files compile
  2. Everything links
  3. Build passes the smoke tests
- Broken builds should be fixed immediately

Smoke Tests

- Exercise the entire system
- Quick set of tests to run daily
- Not necessarily exhaustive – should be capable of exposing major problems
- Ensure the daily build runs and is “sane”
- Needs to be kept current

Continuous Integration

- A step further: integrate and test continuously
- “Continuously” = every few hours, at least once a day
- Repository – development should happen in `master`
  - Branches are for experiments and bugfixes in older versions
- Everyone commits to `master` every day
- Load current release code, merge changes, run tests until 100% pass
- Every commit to `master` should be built
- State of `master` builds visible to everybody
- tooling: CI servers – detect build, run tests

Summary

- Integration testing: check if independently developed units work correctly when combined
- Approaches to incremental integration and integration testing: Top-down, botom-up, hybrid, …
- Various test doubles can be used to simulate behavior of dependencies
- Daily builds and smoke tests – ensure always a relatively working system
- Continuous integration – integrate every change