Verificação e Validação

Verificação vs Validação

- Verificação:
  "Estamos construindo o produto corretamente?"
- O software deve estar de acordo com sua especificação.
- Validação:
  "Estamos construindo o produto certo?".
- O software deve fazer o que o usuário realmente deseja.

O processo V & V

- Deve ser aplicado a cada estágio do desenvolvimento de software
  - Vale tanto para verificação quanto validação
- Tem dois objetivos principais:
  - Descobrir problemas em um sistema;
    - Problema = sistema que não satisfaz sua especificação
  - Avaliar se o sistema é útil e usável ou não em uma situação operacional.

Objetivos de V& V

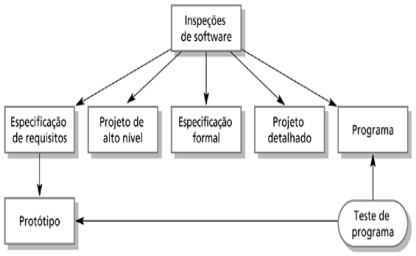
- Verificação e validação devem estabelecer confiança de que o software é adequado ao seu propósito.
- Isto NÃO significa completamente livre de defeitos.
- Ao invés disso, deve ser bom o suficiente para seu uso pretendido
  - Tipo de uso determinará o grau de confiança necessário.

V & V estática e dinâmica

- Inspeções de software. Análise de representações estáticas do sistema com o objetivo de descobrir problemas (verificação estática)
  - Pode ser suplementado por um documento baseado em ferramenta e análise de código.
- Teste de software. Relacionado ao exercício e à observação do comportamento do produto (verificação dinâmica)
  - O sistema é executado com dados de teste e seu comportamento operacional é observado.
- Outras técnicas: análise dinâmica, prototipação, entrevistas, cenários

V & V estática e dinâmica

Figura 22.1
Verificação e validação dinâmica e estática.



Testes de Programas

- Podem revelar a presença de defeitos, NÃO a ausência.
- Principal técnica de validação para requisitos não-funcionais
  - O software é executado para ver como se comporta.
- Devem ser usados em conjunto com a verificação estática para fornecer uma cobertura mais completa de V&V.

Tipos de teste

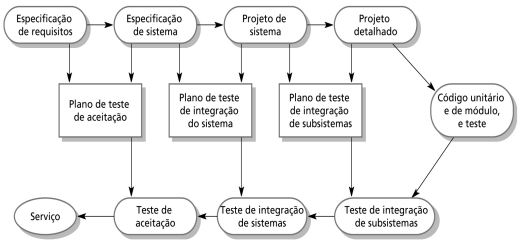
- Teste de validação
  - Pretende mostrar que o software atende as necessidades dos usuários;
  - Um teste bem sucedido é aquele que mostra que um requisito foi adequadamente implementado.
- Teste de defeitos
  - Testes projetados para descobrir defeitos de sistema;
  - Um teste de defeitos bem sucedido é aquele que revela a presença de falha em um sistema;
  - Abordado no Capítulo 23.

Teste e Depuração

- Testes e depuração e de defeitos são processos distintos
- Verificação e validação estão relacionados ao estabelecimento da existência de falhas em um programa
- Depuração está relacionado à localização e repararação dessas falhas

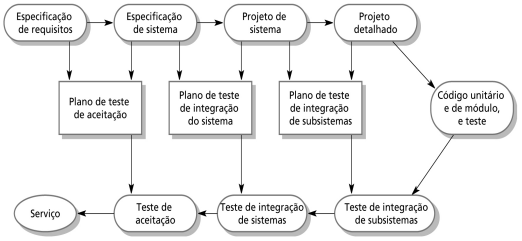
O Modelo V de Desenvolvimento

Figura 22.3 Plano de teste como ligação entre o desenvolvimento e os testes.



O Modelo V de Desenvolvimento

Figura 22.3 Plano de teste como ligação entre o desenvolvimento e os testes.



A Estrutura de um Plano de Testes

- Processo de teste
- Rastreabilidade de requisitos
- Itens testados
- Cronograma de testes
- Procedimentos de registro de testes
- Requisitos de hardware e de software
- Restrições

Inspeções de software

- Exame de um artefato de desenvolvimento para descobrir anomalias e defeitos
  - Técnica de verificação
  - Feitas por uma equipe, em uma reunião formal
- Não requerem a execução de um sistema,
  - Podem e devem ser usadas antes da implementação
- Podem ser aplicadas a qualquer artefato
- Têm se mostrado uma técnica efetiva para descobrir erros de programa
  - "*…according to statistics it will find up to 90% of the contained errors, if done properly.*"
    - (http://www.the-software-experts.de/e_dta-sw-test-inspection.htm)

Sucesso das inspeções

- Muitos defeitos diferentes podem ser descobertos em uma única inspeção.
  - Em teste, um defeito pode mascarar um outro, por isso, várias execuções são necessárias.
- Conhecimento sobre o domínio e sobre programação aumentam a eficácia
  - Revisores têm alta probabilidade de já ter visto os tipos de erros que normalmente surgem

Inspeções e testes

- Inspeções e testes são complementares
  - Inspeções => verificação
  - Testes => verificação e validação
- Ambos devem ser usados durante o processo de V & V.
- As inspeções podem verificar a conformidade com uma especificação
  - Não verificam a conformidade com os requisitos reais do cliente!
- As inspeções não podem verificar características de qualidade, tais como desempenho, usabilidade, etc.

Inspeções de Programas

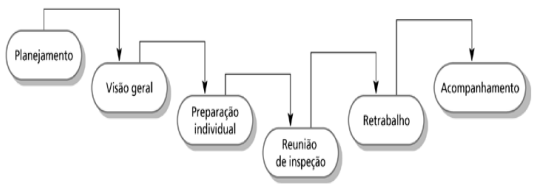
- Abordagem formalizada para revisões de artefatos
- Voltadas explicitamente para detecção de falhas (não correção).
- Falhas podem ser erros lógicos (por exemplo, uma variável não iniciada) ou não-conformidade com padrões.

Pré-condições para inspeção

- Uma especificação precisa deve estar disponível
- Os membros da equipe devem estar familiarizados com os padrões organizacionais
- O código sintaticamente correto ou outras representações do sistema devem estar disponíveis.
- Um *checklist* de erros deve ser preparado
- A gerência deve aceitar que a inspeção aumentará os custos no início do processo de software
- A gerência não deve usar inspeções para avaliar pessoal

O processo de inspeção

Figura 22.4 Processo de inspeção.



Procedimento de inspeção

- Visão geral do sistema apresentado para a equipe de inspeção
- Código e documentos associados são previamente distribuídos para a equipe de inspeção
- A inspeção ocorre e os erros descobertos são anotados
  - Alguns podem ser descobertos na análise individual
- Modificações são feitas para reparar os erros descobertos
- Uma nova inspeção pode ou não ser necessária

Papéis da inspeção

Tabela 22.1 Papéis no processo de inspeção

Papel	Descrição
Autor e proprietário	O programador ou projetista responsável por produzir o programa ou o documento. Ele é responsável pela correção de defeitos descobertos durante o processo de inspeção.
Inspetor	Encontra erros, omissões e inconsistências nos programas e documentos. Pode também identificar questões mais amplas fora do escopo da equipe de inspeção.
Leitor	Apresenta o código ou documento em uma reunião de inspeção.
Relator	Registra os resultados da reunião de inspeção.
Presidente ou moderador	Gerencia o processo e facilita a inspeção. Relata os resultados do processo ao moderador-chefe.
Moderador-chefe	Responsável pelos aprimoramentos do processo de inspeção, pela atualização da lista de verificação, pelo desenvolvimento de padrões etc.

Papéis da inspeção

Tabela 22.1 Papéis no processo de inspeção

Papel	Descrição
Autor e proprietário	O programador ou projetista responsável por produzir o programa ou o documento. Ele é responsável pela correção de defeitos descobertos durante o processo de inspeção.
Inspetor	Encontra erros, omissões e inconsistências nos programas e documentos. Pode também identificar questões mais amplas fora do escopo da equipe de inspeção.
Leitor	Apresenta o código ou documento em uma reunião de inspeção.
Relator	Registra os resultados da reunião de inspeção.
Presidente ou moderador	Gerencia o processo e facilita a inspeção. Relata os resultados do processo ao moderador-chefe.
Moderador-chefe	Responsável pelos aprimoramentos do processo de inspeção, pela atualização da lista de verificação, pelo desenvolvimento de padrões etc.

Checklists de Inspeção

- Um *checklist* de erros comuns deve ser usado para direcionar a inspeção.
- *Checklists* de erros são dependentes de linguagem de programação
  - Refletem os erros característicos com maior probabilidade de surgimento na linguagem
- Exemplos de itens da *checklist*: inicialização de variáveis, terminação de laços, etc.
- Inspeções também podem "executar" o sistema, através da análise passo-a-passo de seu código

Tabela 22.2 Verificações de inspeção

Classe de defeitos	Verificação de inspeção
Defeitos de dados	Todas as variáveis de programa são iniciadas antes que seus valores sejam usados? Todas as constantes foram denominadas? O limite superior de vetores deve ser igual ao tamanho do vetor ou Tamanho –1? Se são usados strings de caracteres, um delimitador é explicitamente atribuído? Existe alguma possibilidade de overflow de buffer?
Defeitos de controle	Para cada declaração condicional, a condição está correta? Cada loop está terminando corretamente? As declarações compostas estão corretamente delimitadas entre parênteses? Em declarações 'case', todos os casos possíveis são levados em conta? Se um comando 'break' é necessário após cada caso nas declarações 'case', ele foi incluído?
Defeitos de entrada/saída	Todas as variáveis de entrada são usadas? Todas as variáveis de saída têm valor atribuído antes de sua saída? Entradas inesperadas podem fazer com que os dados sejam corrompidos?
Defeitos de interface	Todas as chamadas de funções e de métodos têm o número correto de parâmetros? Tipos de parâmetros reais e formais se combinam? Os parâmetros estão na ordem correta? Se os componentes acessam memória compartilhada, eles têm o mesmo modelo de estrutura de memória compartilhada?
Defeitos de gerenciamento de armazenamento	Se uma estrutura ligada é modificada, todas as ligações foram corretamente reatribuídas? Se o armazenamento dinâmico foi usado, o espaço foi corretamente alocado? O espaço de memória é liberado depois de não ser mais necessário?
Defeitos de gerenciamento de exceções	Todas as condições possíveis de erro foram consideradas?

Taxa de Inspeção

- 500 declarações de código-fonte por hora durante a visão geral.
- 125 declarações de código fonte por hora durante a preparação individual.
- De 90 a 125 declarações por hora podem ser inspecionados durante a reunião de inspeção.
- A inspeção é, portanto, um processo dispendioso.
- A inspeção de 500 linhas custa aproximadamente 40 homem-hora de esforço – £2800 em valores da Grã-Bretanha (UK).

Análise Estática Automatizada

- Processamento de código fonte (ou *bytecode*)
- Varre o texto do programa e tenta descobrir condições potencialmente errôneas
  - Técnica de verificação
- São um suplemento, mas não um substituto, para as inspeções
- Podem ser usadas para aumentar a compreensão sobre um programa

Verificações de Análise Estática

Tabela 22.3 Verificação de análise estática automatizada

Classe de defeitos	Verificação de análise estática
Defeitos de dados	Variáveis usadas antes da inicialização Variáveis declaradas, mas nunca usadas Variáveis atribuídas duas vezes, mas nunca usadas entre atribuições Possíveis violações de limites de vetor Variáveis não declaradas
Defeitos de controle	Código inacessível Ramificações incondicionais em loops
Defeitos de entrada/saída	Variáveis geradas duas vezes sem tarefa de impedimento
Defeitos de interface	Tipo de parâmetro que não combina Número de parâmetro que não combina Resultados de funções não usadas Funções e procedimentos não chamados
Defeitos de gerenciamento de armazenamento	Ponteiros não atribuídos Aritmética de ponteiros

Tipos de Análise Estática

- Análise de fluxo de controle. Verifica laços com múltiplos pontos de saídas ou de entrada, encontra código inacessível, etc
- Análise de uso de dados. Detecta variáveis não iniciadas, variáveis que são declaradas mas nunca usadas, etc
- Análise de interface. Verifica a consistência das declarações de rotina e procedimentos e seus usos

Tipos de Análise Estática

- Análise de caminho. Identifica caminhos através do programa e estabelece as declarações executadas naquele caminho.
  - Pode também verificar se certo predicados são verdadeiros
  - Destaca as informações para inspeção ou revisão de código
- Muitos outros tipos de análises são possíveis!
- Limitações: escalabilidade, completude, precisão, excesso de informações

Análise Estática com o Lint

Figura 22.5
Análise estática com o Lint.

```
138% more lint_ex.c
#include <stdio.h>
printarray (Anarray)
int Anarray;
{
printf("%d",Anarray);
}
main ()
{
int Anarray[5]; int i; char c;
printarray (Anarray, i, c);
printarray (Anarray) ;
}

139% cc lint_ex.c
140% lint lint_ex.c

lint_ex.c(10): warning: c may be used before set
int_ex.c(10): warning: i may be used before set
printarray: variable # of args. lint_ex.c(4) :: lint_ex.c(10)
printarray, arg. 1 used inconsistently lint_ex.c(4) :: lint_ex.c(10)
printarray, arg. 1 used inconsistently lint_ex.c(4) :: lint_ex.c(11)
printf returns value which is always ignored
```

Uso de Análise Estática

- Particularmente valiosa quando uma linguagem tal como C, que tem tipagem fraca, é usada
  - Muitos erros não são detectados pelo compilador.
- Em linguagens como Java, que têm verificação tipo forte, muitos erros são detectados durante a compilação.
  - Análises mais sofisticadas ainda podem ser úteis, porém!