THE VALUE OF PERFORMANCE.

NORTHROP GRUMMAN

Reduce Development and Testing Time on Embedded Space Programs With Auto-Generated Code

Matthew Conte

Software Engineer

Northrop Grumman Electronic Systems

Abstract

Embedded software for space systems is one of the most expensive types of software to produce, mainly because of the rigorous testing involved. However, developers on the Northrop Grumman Space Systems Software team have observed that infrastructure code such as class definitions for interface messages, configuration files, and telemetry points is often repetitive, with each class differing only in parameters and implementation of common functions. It became clear that rewriting this repetitive code for each program and, sometimes, within a single program was taking developers' focus away from the true engineering problems.

To reduce time spent writing repetitive code, we have built up a suite of tools to parse human-readable documents such as Interface Control Documents (ICDs), configuration files, and telemetry definition spreadsheets and automatically generate C++ classes from these files. The use of these tools has improved code reuse and reduced the number of SLOCs that must be tested, resulting in cost and schedule benefits for our programs and freeing up developers' time to concentrate on the unique technical challenges of each program. This presentation will explain how we designed our code generation tools and discuss the many benefits of these tools to our products.

Agenda

- Who We Are
- Why Auto-Generate Code?
- Auto-Generation Tool Suite
- Evaluation of Our Auto-Gen Efforts
- Lessons Learned

Who We Are

NORTHROP GRUMMAN

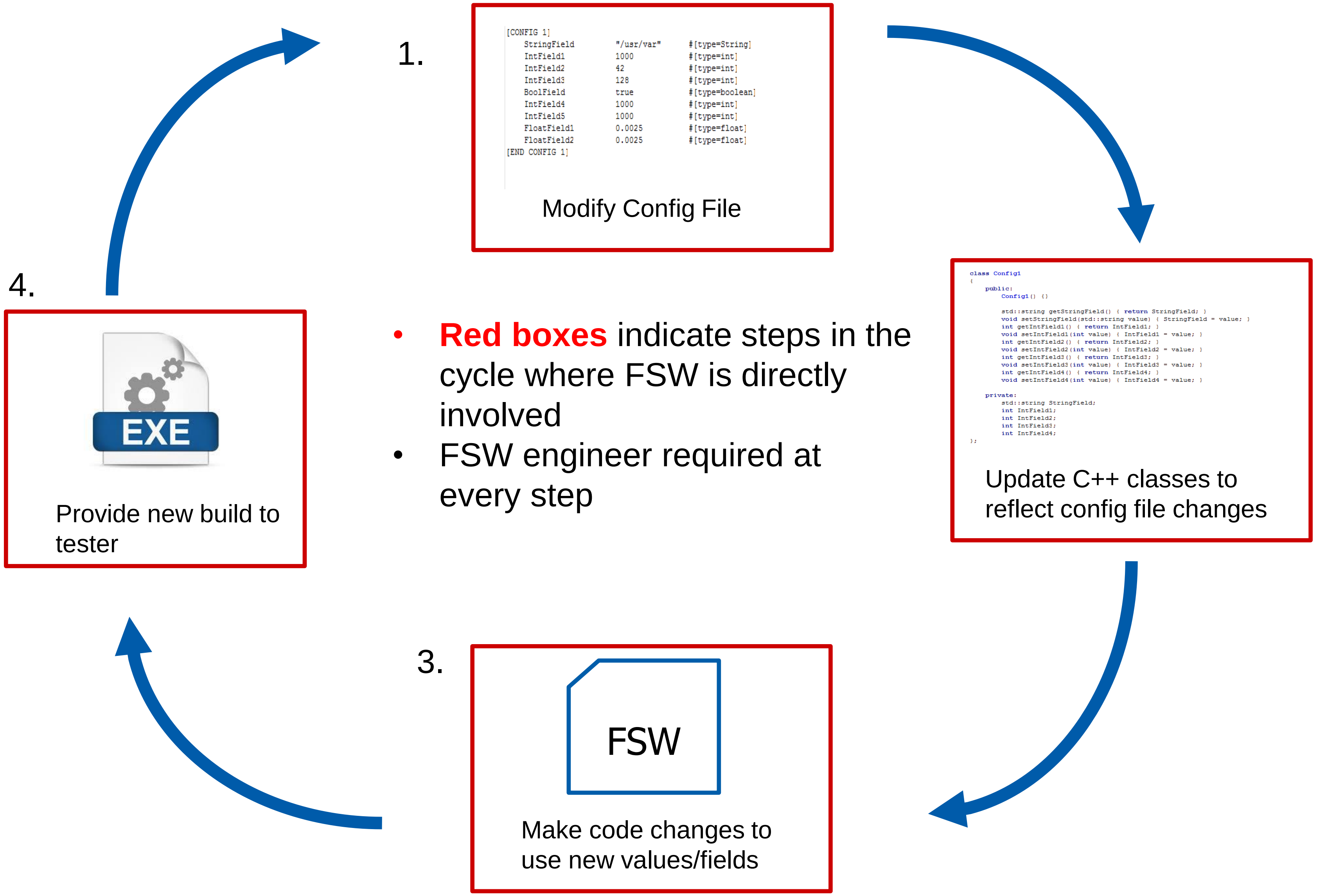
- Northrop Grumman Space Systems Software (SSSW) group
- Develop embedded FSW for space systems
- Maintain a test software (TSW) suite to control our integration lab equipment and exercise functions of the system
- Work with a variety of languages and tools
  - C++, Java, Tcl, Python
  - VxWorks Workbench

Program Background

- Our system is highly configurable
  - Configuration files
  - Telemetry definition files
  - Calibration files
- Need C++ code to read in these files at startup and provide the data to the relevant software components
- Dozens of each type of file
  - Writing and maintaining C++ classes for each individual file would be a large effort
- Settings change constantly during the testing phase of the program
  - If FSW group is bogged down in supporting test engineers, no time to continue development

FSW Development Cycle

NORTHROP GRUMMAN


1.
Modify Config File
2.
Update C++ classes to reflect config file changes
3.
FSW
Make code changes to use new values/fields
4.
EXE
Provide new build to tester

```
[CONFIG 1]
    StringField         "/usr/var"      #[type=String]
    IntField1           1000            #[type=int]
    IntField2           42              #[type=int]
    IntField3           128             #[type=int]
    BoolField           true            #[type=boolean]
    IntField4           1000            #[type=int]
    IntField5           1000            #[type=int]
    FloatField1         0.0025          #[type=float]
    FloatField2         0.0025          #[type=float]
[END CONFIG 1]
```

```
class Config1
{
    public:
        Config1() {}

        std::string getStringField() { return StringField; }
        void setStringField(std::string value) { StringField = value; }
        int getIntField1() { return IntField1; }
        void setIntField1(int value) { IntField1 = value; }
        int getIntField2() { return IntField2; }
        void setIntField2(int value) { IntField2 = value; }
        int getIntField3() { return IntField3; }
        void setIntField3(int value) { IntField3 = value; }
        int getIntField4() { return IntField4; }
        void setIntField4(int value) { IntField4 = value; }

    private:
        std::string StringField;
        int IntField1;
        int IntField2;
        int IntField3;
        int IntField4;
};
```

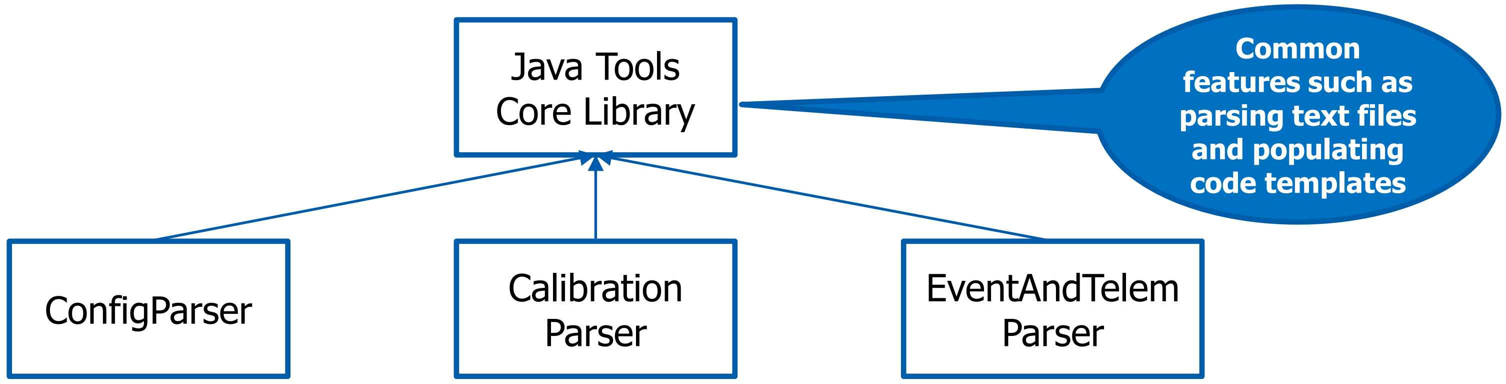
- **Red boxes** indicate steps in the cycle where FSW is directly involved
- FSW engineer required at every step

Why Auto-Generate Code?

- Engineers don’t want to do tedious work of maintaining dozens of configuration files
  - Especially not during Integration and Test (I&T) phase
  - Systems and test engineers update config files multiple times per day
  - Being bogged down with small updates distracts from critical development work
- Program management wants a shorter schedule and lower cost
  - Test the auto-gen tool once instead of manually-edited code every time it changes
  - Time spent writing tool up-front saves developers’ time later
  - Tools can be repurposed for later programs, promoting reuse

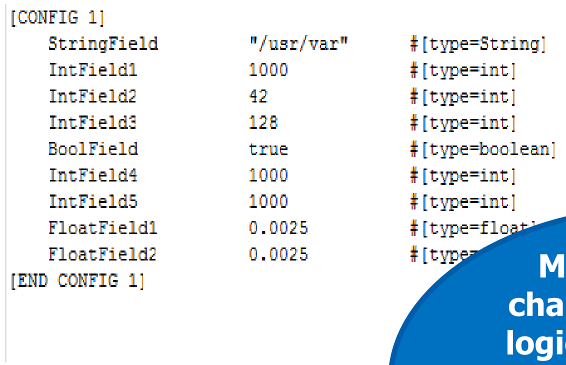
Auto-Generation Tool Suite

- Developed a suite of Java tools to generate code for several types of configuration files



- Inputs:
  - Spreadsheets and text files created by Systems and Test Engineers
  - C++ class templates
- Outputs:
  - C++ classes which are built into our flight code
    - Hold the data from the input files
    - Provide an interface for the rest of the system to access the data

Configuration Parser



```
[CONFIG 1]
    StringField         "/usr/var"       #[type=String]
    IntField1           1000             #[type=int]
    IntField2           42               #[type=int]
    IntField3           128              #[type=int]
    BoolField           true             #[type=boolean]
    IntField4           1000             #[type=int]
    IntField5           1000             #[type=int]
    FloatField1         0.0025           #[type=float
    FloatField2         0.0025           #[type
[END CONFIG 1]
```

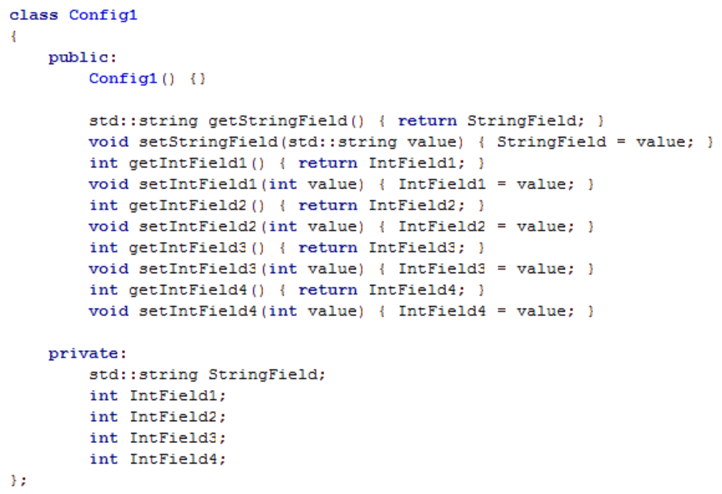
Engineer feeds config.txt into the Java tool

ConfigParser Tool

Manually coding changes to program logic is unavoidable. But these are now the only changes which involve FSW engineer.

Tool outputs C++ classes

Test Engineer crea
config.txt



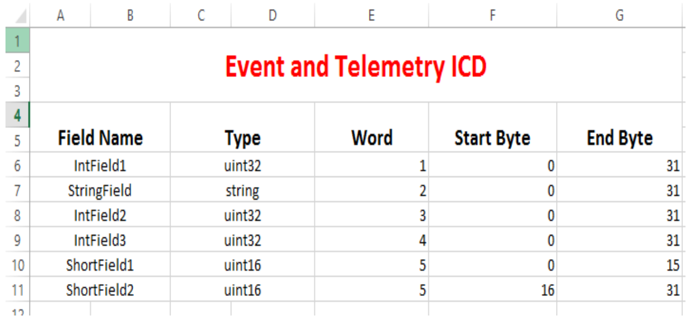
```
class Config1
{
    public:
        Config1() {}

        std::string getStringField() { return StringField; }
        void setStringField(std::string value) { StringField = value; }
        int getIntField1() { return IntField1; }
        void setIntField1(int value) { IntField1 = value; }
        int getIntField2() { return IntField2; }
        void setIntField2(int value) { IntField2 = value; }
        int getIntField3() { return IntField3; }
        void setIntField3(int value) { IntField3 = value; }
        int getIntField4() { return IntField4; }
        void setIntField4(int value) { IntField4 = value; }

    private:
        std::string StringField;
        int IntField1;
        int IntField2;
        int IntField3;
        int IntField4;
};
```

Software engineer makes required software changes and provides build to tester

Configuration Parser

- Role of the software engineering team is minimized in the process of updating configuration files for testing
  - New classes are automatically generated by the tool
  - Tool is trusted so testing not required on updated classes
- Quick turnaround compared to manually modifying and testing C++ code
  - Turn out build in minutes
  - Speed essential because of tight I&T schedule
- Less burden placed on software group to support testing
  - More time to develop actual flight code

Event and Telemetry Parser



Event and Telemetry ICD

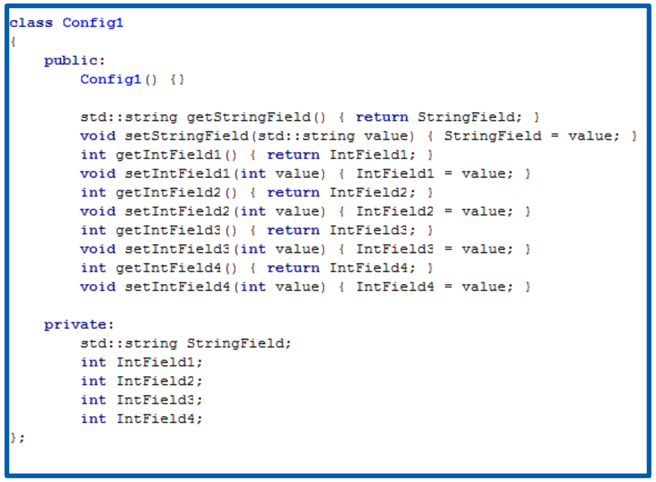
Field Name	Type	Word	Start Byte	End Byte
IntField1	uint32	1	0	31
StringField	string	2	0	31
IntField2	uint32	3	0	31
IntField3	uint32	4	0	31
ShortField1	uint16	5	0	15
ShortField2	uint16	5	16	31

Telemetry ICD created in collaboration with customer

Feed spreadsheet into EventAndTelemParser

EventAndTelemParser Tool

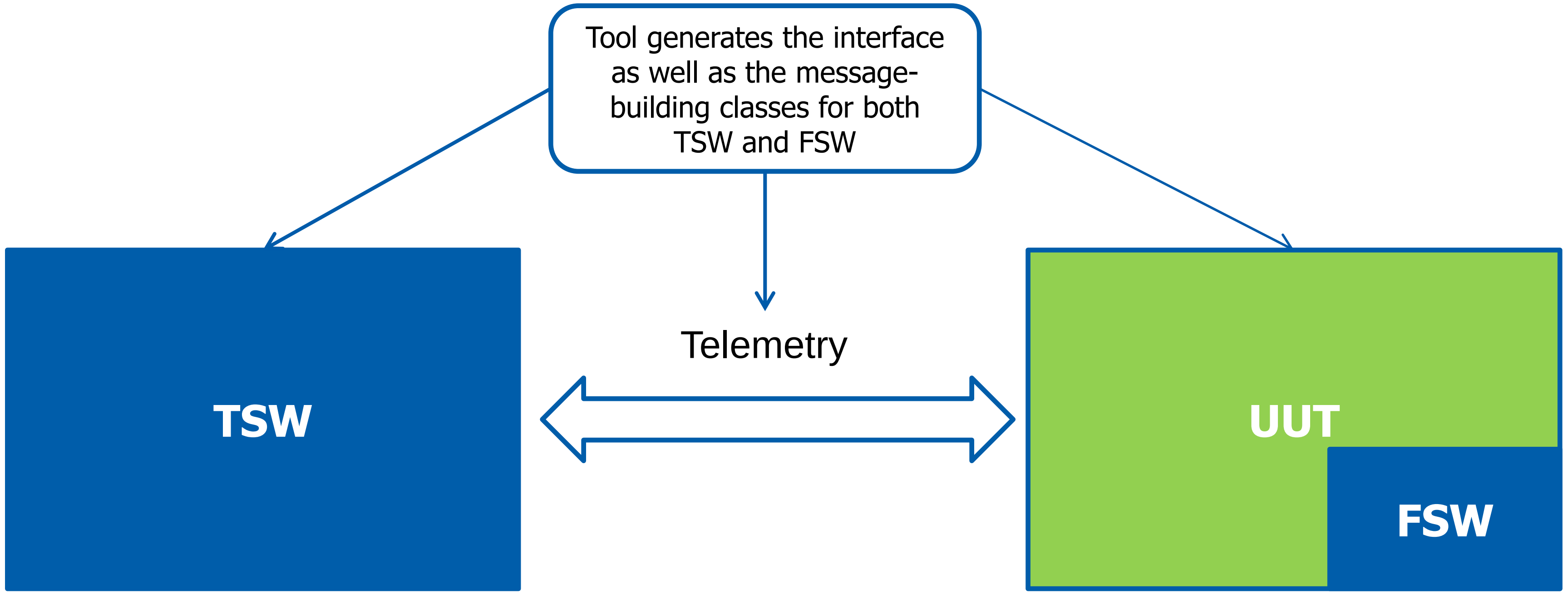
Outputs C++ classes and definition files for TSW suite



```
class Config1
{
    public:
        Config1() {}

        std::string getStringField() { return StringField; }
        void setStringField(std::string value) { StringField = value; }
        int getIntField1() { return IntField1; }
        void setIntField1(int value) { IntField1 = value; }
        int getIntField2() { return IntField2; }
        void setIntField2(int value) { IntField2 = value; }
        int getIntField3() { return IntField3; }
        void setIntField3(int value) { IntField3 = value; }
        int getIntField4() { return IntField4; }
        void setIntField4(int value) { IntField4 = value; }

    private:
        std::string StringField;
        int IntField1;
        int IntField2;
        int IntField3;
        int IntField4;
};
```

Event and Telemetry Parser

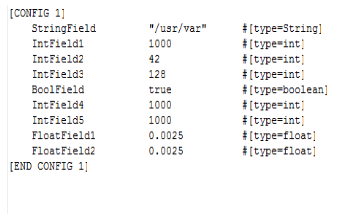
- System events and telemetry points are more complex than config files
- Definitions must be coordinated between FSW and TSW
  - TSW simulates sending/receiving telemetry from FSW
  - Cannot test unless both are in-sync


Tool generates the interface as well as the message-building classes for both TSW and FSW
Telemetry
TSW
UUT
FSW

FSW Development Cycle (with tool suite)

NORTHROP GRUMMAN

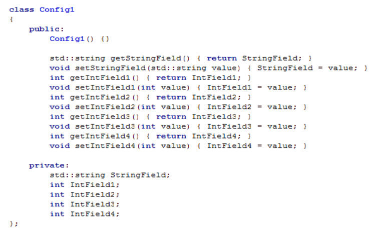
1.



```
[CONFIG 1]
    StringField         "/usr/var"      #[type=String]
    IntField1           1000            #[type=int]
    IntField2           42              #[type=int]
    IntField3           128             #[type=int]
    BoolField           true            #[type=boolean]
    IntField4           1000            #[type=int]
    IntField5           1000            #[type=int]
    FloatField1         0.0025          #[type=float]
    FloatField2         0.0025          #[type=float]
[END CONFIG 1]
```

Modify Config File

2.



```
class Config1
{
    public:
        Config1() {}

        std::string getStringField() { return StringField; }
        void setStringField(std::string value) { StringField = value; }
        int getIntField1() { return IntField1; }
        void setIntField1(int value) { IntField1 = value; }
        int getIntField2() { return IntField2; }
        void setIntField2(int value) { IntField2 = value; }
        int getIntField3() { return IntField3; }
        void setIntField3(int value) { IntField3 = value; }
        int getIntField4() { return IntField4; }
        void setIntField4(int value) { IntField4 = value; }

    private:
        std::string StringField;
        int IntField1;
        int IntField2;
        int IntField3;
        int IntField4;
};
```

Update C++ classes to reflect config file changes

3.

FSW

Make code changes to use new values/fields

4.

EXE

Provide new build to tester

- Removes FSW engineers from steps 1. and 2.
- Now only required for code changes in step 3.

How Has Auto-Generation Worked for Us?

- Software engineers enjoy the auto-generation approach
  - Less tedious work
  - Reduced testing time for code changes
  - More time to work on the true engineering challenges
- Still figuring out the development life-cycle for the tools
  - See Lessons Learned
- Has been a tougher sell to management
  - Need to be convinced to spend money on non-delivered code

Lessons Learned

- Rigorous testing of the tools should be performed well before I&T phase of the program
  - Time fixing bugs in tool is time that could be spent fixing integration issues
  - Sinking too much maintenance time into the tool defeats its purpose
- Develop early and reuse
  - This was the SSSW group's first foray into auto-generating config code
  - Developed tools from scratch side-by-side with FSW
- Keep tools up-to-date
  - Make change to tool as soon as new formats released
  - Don't let auto-gen files get "stale"; regenerate with new builds
- Maximize use of code templates
  - Easier to swap out than if the text of the CPP file is hard-coded in the tool
- Convince management of the value of investing time in auto-gen tools up-front

THE VALUE OF PERFORMANCE.

NORTHROP GRUMMAN